IEICE TRANSACTIONS

# on Information and Systems

**VOL. E105-D NO. 12**
**DECEMBER 2022**



A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

IEICE TRANS. INF. & SYST., VOL.E105–D, NO.12 DECEMBER 2022

PAPER Special Section on Forefront Computing

# Holmes: A Hardware-Oriented Optimizer Using Logarithms

**Yoshiharu YAMAGISHI**[†], **Tatsuya KANEKO**[†], ***Student Members***, **Megumi AKAI-KASAYA**[††,†††], ***and*** **Tetsuya ASAI**[††], ***Members***

**SUMMARY** Edge computing, which has been gaining attention in recent years, has many advantages, such as reducing the load on the cloud, not being affected by the communication environment, and providing excellent security. Therefore, many researchers have attempted to implement neural networks, which are representative of machine learning in edge computing. Neural networks can be divided into inference and learning parts; however, there has been little research on implementing the learning component in edge computing in contrast to the inference part. This is because learning requires more memory and computation than inference, easily exceeding the limit of resources available for edge computing. To overcome this problem, this research focuses on the optimizer, which is the heart of learning. In this paper, we introduce our new optimizer, hardware-oriented logarithmic momentum estimation (Holmes), which incorporates new perspectives not found in existing optimizers in terms of characteristics and strengths of hardware. The performance of Holmes was evaluated by comparing it with other optimizers with respect to learning progress and convergence speed. Important aspects of hardware implementation, such as memory and operation requirements are also discussed. The results show that Holmes is a good match for edge computing with relatively low resource requirements and fast learning convergence. Holmes will help create an era in which advanced machine learning can be realized on edge computing.

***key words:*** *optimizer, edge computing, neural network, nonvolatile memory, quantization*

## 1. Introduction

Edge computing, which has gained recognition in recent years, is a technology that completes information processing immediately without the need to connect to a powerful server computer through a network [1]. This has several advantages, including being unaffected by the network environment in which it is used and reducing the load on the cloud, which significantly increases every year. Edge computing is also an excellent option in terms of security because it prevents leakage of confidential information.

By contrast, edge computing has a weakness in that the available resources are limited. To overcome this weakness, many researchers have focused on nonvolatile memory [2], [21]. Nonvolatile memory performs well with low power consumption as it does not require electricity, except when in use, which helps implement the "inference part of the neural network" in edge computing. However, nonvolatile memory has the disadvantage of consuming a large amount of power during memory access. This makes it incompatible with operations that require repeated data writes, such as neural network training. Recently, research has been conducted on hardware for the "learning part", in contrast with the "inference part" [3]–[9]. In addition, recent studies on the learning have struggled to achieve high performance and low power consumption, and many limitations remain [5]–[9]. For example, Kaneko et al. included hardware perspectives, such as "quantization bit limit," "low resource," and "power consumption reduction" when designing neural networks by using edge computing [1]. Kaneko et al. believe that a future challenge is to improve the accuracy degradation caused by edge computing-oriented limitations, such as quantization bit limitation. Although not included in their paper, in this study, the "increase in the number of required training sessions" was also identified as a problem.

Here, the focus is on the fact that machine learning theory was not designed for edge computing. When implementing machine learning in edge computing, some major constraints must be considered. Forcing an implementation of a theory that is not designed for edge computing (e.g., using high-level synthesis) often results in more resource requirements and easily exceeds the constraints [10]. Particular attention is paid to the optimizer, which is the core of the learning algorithm in machine learning.

In this study, a new method, the optimizer Holmes is introduced. This method was developed by considering the characteristics and strengths of the hardware, and its theory is different from that of existing optimizers.

## 2. Optimizer Review

The optimizer is an algorithm that searches for the optimal state via multiple modifications to a structure [11]. In a neural network, an optimizer is used to reduce the difference between the inferred and the correct value, which can be considered as the heart of the learning algorithm. Therefore, the optimizer algorithm is responsible for most of the operations in the neural network and has a large impact on the accuracy and number of training sessions. Therefore, optimizers have attracted the attention of many researchers,

Manuscript received December 11, 2021.
Manuscript revised March 12, 2022.
Manuscript publicized May 11, 2022.

[†]The authors are with Graduate School of Information Science and Technology, Hokkaido University, Sapporo-shi, 060–0814 Japan.
[††]The authors are with Faculty of Information Science and Technology, Hokkaido, Hokkaido University, Sapporo-shi, 060–0814 Japan.
[†††]The authors are with Graduate School of Engineering, Suita-shi, 565–0871 Japan.

DOI: 10.1587/transinf.2022PAP0001

and new optimizers are constantly being invented [12]–[18].

### 2.1 Gradient Descent

Gradient descent is an optimizer that can be considered as the prototype of many optimizers. The update equation of gradient descent is given by,

$$w_t = w_{t-1} - \eta \nabla_w L_{(w_{t-1})}, \tag{1}$$

where $w$ is the weight; $t$ is the number of iterations; $\eta$ is the learning rate; and $L$ is the loss function. It is a simple theory that by differentiating the loss function $L_{(w_{t-1})}$, the direction of the steepest descent (the direction that is considered to have the optimal solution) is obtained, and the weights are modified.

### 2.2 Momentum

The momentum optimization method adds the concept of the law of inertia to gradient descent [13], [14]. Gradient descent moves down the gradient in small constant steps, whereas momentum optimization changes the speed of gradient traversal depending on the previous gradient. Thus, in general, momentum optimization converges faster than gradient descent. The update equation of momentum optimization is given by,

$$\begin{aligned} m_t &= \beta m_{t-1} - \eta \nabla_w L_{(w_{t-1})}, \\ w_t &= w_{t-1} + m_t, \end{aligned} \tag{2}$$

where $m$ is a momentum vector. The momentum vector is the sum of the previous gradients, that is, the momentum optimization method uses the gradient as the acceleration rather than the velocity; $\beta$, a hyperparameter with a value between 0 and 1, prevents the momentum vector from becoming too large.

### 2.3 RMSProp

RMSProp is an optimizer that focuses on the learning rate. Learning rate is a hyperparameter related to the speed of decreasing gradient [15]. If the learning rate is too small, the learning process will be slow to complete, and will get trapped in a local solution. By contrast, if the value of the learning rate is too large, the progress is fast; however, the solution tends to oscillate, and the accuracy becomes unstable. Furthermore, because the optimal value varies depending on the problem to be solved, the adjustment of the learning rate has been a problem for many researchers. The update equation of RMSProp is given by,

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta) \nabla_w L^2_{(w_{t-1})}, \\ w_t &= w_{t-1} - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla_w L_{(w_{t-1})}, \end{aligned} \tag{3}$$

where $v$ is the sum of the squares of the past gradients. By using the sum of squares of past gradients, the learning rate is automatically adjusted. In addition, in gradient descent, the learning rate is multiplied equally over the entire gradient, whereas in RMSProp, v varies according to the size of the individual elements of the gradient. Therefore, the accuracy is generally high; $\beta$, a hyperparameter with a value between 0 and 1, prevents v from becoming too large (i.e., the learning rate becoming too small).

## 3. Proposed Algorithm

The new optimizer, Holmes, is inspired by the momentum optimization method and is theoretically designed with edge computing in mind. There are three major novelties in Holmes. First, quantization, which was previously considered the main source of reduced accuracy, is used in an ingenious way to improve accuracy. Second, the goal is to use few resources and low power, which makes the algorithm appropriate for implementation in edge computing. Third, Holmes has a property similar to RMSProp in that it automatically adjusts the amount of learning rate reduction depending on the magnitude of individual values, which is not present in previous momentum optimization methods.

Originally, Holmes' theory was an attempt to qualitatively mimic some of the calculations of the momentum optimization method using quantization. Specifically, the hyperparameter $\beta$ (which accepts values between 0 and 1) used in the momentum optimization method is multiplied by the momentum vector to make the momentum vector slightly smaller. In Holmes, quantization is substituted for the operation of slightly reducing the momentum vector value. Thus, Holmes can be formulated as:

$$\begin{aligned} m_t &= 2^{\lfloor \log_2(m_{t-1}) \rfloor} - \eta \nabla_w L_{(w_{t-1})}, \\ w_t &= w_{t-1} + m_t. \end{aligned} \tag{4}$$

Comparing this equation to Eq. (2), the part of $\beta m_{t-1}$ is $2^{\lfloor \log_2(m_{t-1}) \rfloor}$, and it is observed that one hyperparameter is lost. This part represents the "logarithmic quantization".

Fundamentally, quantization reduces the expressive power of a number, and the more quantization is used, the lower is the accuracy. However, by effectively combining different types of quantization, Holmes is able to improve the accuracy. Holmes quantization consists of three elements: a limitation on the number of quantization bits, fixed-point number, and logarithmic quantization.

### 3.1 Fixed-Point Number

There are two typical methods for representing decimals in hardware: fixed-point numbers and floating-point numbers. In fixed-point numbers, the memory for storing integers and memory for storing decimals are separated in advance. The memory used for storage consists of a sign part, an integer part, and a decimal part. By contrast, in floating-point numbers, there is a memory for storing numerical values and a memory for storing the position of the decimal point, and these data are used to construct numerical values during numerical calculations. The memory for storing consists of a

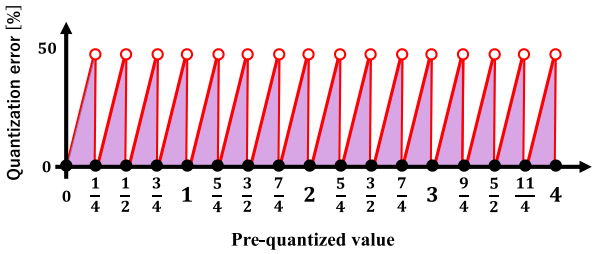


**Fig. 1** Percentage of quantization error in relation to the value prior to quantization when the number of quantization bits is limited to two decimal places in a fixed-point number. The possible values after quantization are located at equal intervals, indicating that the percentage of error is between 0% and 50%.

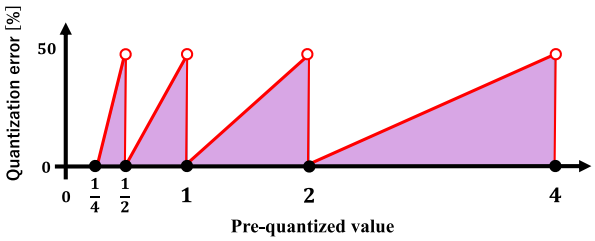


**Fig. 2** Percentage of quantization error in relation to the pre-quantized value when logarithmic quantization is used. The acceptable values after quantization are not evenly spaced; the larger the absolute value, the wider is the interval. The percentage of error falls between 0% and 50%.

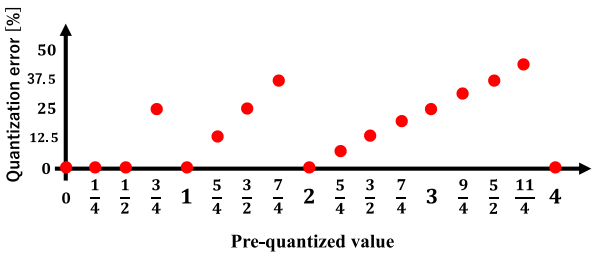


**Fig. 3** Percentage of quantization error in relation to the value prior to the logarithmic quantization performed after limiting the number of quantization bits to two decimal places in a fixed-point number. It is observed that when the absolute value is small, the percentage of error that becomes zero is high, and as the absolute value increases, the percentage of error that becomes zero decreases.

sign part, an exponent part, and a mantissa part. Floating-point numbers need to be constructed every time a numerical value is calculated. They need to be converted into exponents and mantissae every time they are stored in memory. However, fixed-point numbers do not require such conversions, and as they do not need to store the position of the decimal point, they require less memory. Fixed-point numbers are often used in edge computing, where the resources consumed and the execution time are directly proportional to cost and performance.

### 3.2 Limitation on the Number of Quantization Bits

In edge computing, in which available resources are limited, maintaining the details of the data is a major issue. If the values to be retained are too detailed, the memory requirement will get inordinately large. Coarser representation of values can reduce memory requirements. However, coarser values also lead to a loss in expressive power, which leads to a decrease in the accuracy of the neural network. Many researchers have struggled with this problem: "I want to reduce the number of memory requirements as much as possible, but I also want to guarantee a certain level of accuracy." In Fig. 1, the ratio of quantization error to the pre-quantized value is displayed (averaging to 25%) with the number of quantization bits limited to fixed-point numbers.

### 3.3 Logarithmic Quantization

Logarithmic quantization in this paper means approximation to the nearest power of two below the pre-quantized value. For example, a pre-quantized value of 6 is converted to 4 ($= 2^2$), and a pre-quantized value of 0.1 is converted to 0.0625 ($= 2^{-4}$). This conversion is performed in a software-oriented approach as follows:

1. Take the logarithm with a base of 2.
2. Round down to the nearest decimal point.
3. Substitute the value into the exponent of 2 and multiply it by the power.

This conversion may seem complicated, but when you consider the characteristics of the hardware (all values are stored in memory as binary numbers), it is very easy to perform, as shown below.

1. Search for the first place where 0 and 1 are flipped, starting from the upper bits.
2. Assign 1 to the lower bit side of this place, and set everything after that to 0.

The ratio of the quantization error to the pre-quantized value is shown in Fig. 2 and averages to 25%.

### 3.4 Quantization of Holmes

As shown in Fig. 1, when bit limiting quantization is performed under fixed-point numbers, the interval between possible values is equal. However, as shown in Fig. 2, in the case of logarithmic quantization, the smaller the absolute value, the narrower is the interval between possible values, and the larger the absolute value, the wider is the interval between possible values. In both cases, there is no correlation between the magnitude of the value before quantization and percentage quantization error. The percentage of quantization error with logarithmic quantization performed after limiting the quantization bit under fixed-point numbers, is displayed in Fig. 3. The results of the analysis carried out separately for each interval, are presented in Table 1. The larger the value before quantization, the larger the percentage of error caused by logarithmic quantization. By performing this operation on the momentum vector, Holmes achieved the most important feature, which is automatic ad-

**Table 1** Average of the quantization error percentage for each of the pre-quantized values divided into intervals. It is observed that as the absolute value becomes larger, the average value of the error percentage increases. In this table, the number of quantization bits consists of three integer bits and two decimal bits. Therefore, any value greater than $2^3$ is converted to $2^3$; thus, the percentage error cannot be calculated.

| Pre-quantized value (P) | Average value of quantization error [%] |
|---|---|
| $P < 0$ | 0 |
| $0 \leq P < 2^{-2}$ | 0 |
| $2^{-2} \leq P < 2^{-1}$ | 0 |
| $2^{-1} \leq P < 2^{0}$ | 12.5 |
| $2^{0} \leq P < 2^{1}$ | 18.8 |
| $2^{1} \leq P < 2^{2}$ | 21.9 |
| $2^{2} \leq P < 2^{3}$ | 24.2 |
| $2^{3} \leq P$ | – |

justment of the amount of reduction according to the size of individual values.

## 4. Evaluation

Holmes was evaluated in two aspects: "How does it move down the gradient as an optimizer?" and "How does the convergence speed change when it is applied to neural networks?" For the former question, the optimization of the three-hump camel function and Rosenbrock function were examined. For the latter, the speed of learning was verified on a commonly used dataset, MNIST.

Gradient descent and momentum optimization methods were selected for the former question and mini-batch stochastic gradient descent (mini-batch SGD) and momentum optimization methods for the latter for comparison [12]. The mini-batch SGD is a learning method in which the entire training data are divided into smaller portions (mini-batches); losses are calculated for each mini-batch; and back-propagation is performed. The reason for selecting mini-batch SGD was to facilitate the comparison with the previous study by Kaneko et al [1]. The reason for selecting the momentum optimization method was that it is the optimizer on which Holmes' idea was based. However, RMSProp was not used for evaluation because it was concluded that RMSProp and other optimizers are not suitable for edge computing as they require considerable resources to be implemented [15], [17], [18].

### 4.1 Function Optimization

The performance of each optimizer was evaluated using two functions (the three-hump camel function and the Rosenbrock function). The definition of the three-hump camel function is defined as

$$f(x, y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2. \quad (5)$$

The Rosenbrock function is defined as

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2], \quad (6)$$

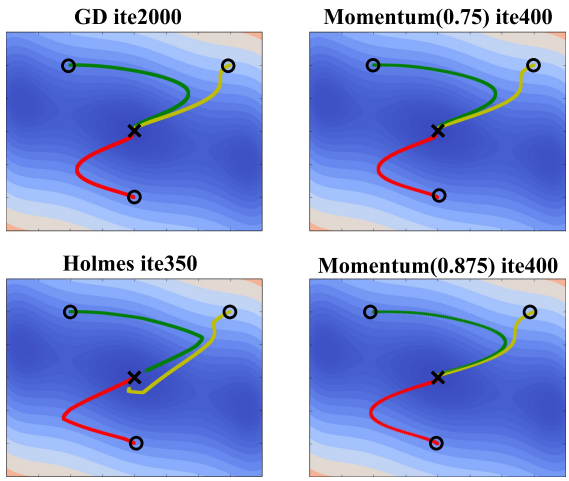


**Fig. 4** Three-hump camel function, starting at location ◯ and searching for the lowest point(location ×). Red, green, and yellow represent the paths of the search with different starting positions. Under these conditions, it is observed that gradient descent reached the optimal solution for all three color paths simultaneously, approximately after 4000 iterations. Holmes with the red path reached the solution after 350 iterations. The momentum optimization method reached the solution for all three color paths simultaneously approximately after 400 iterations, irrespective of whether the hyperparameter $\beta$ was 0.75 or 0.875.

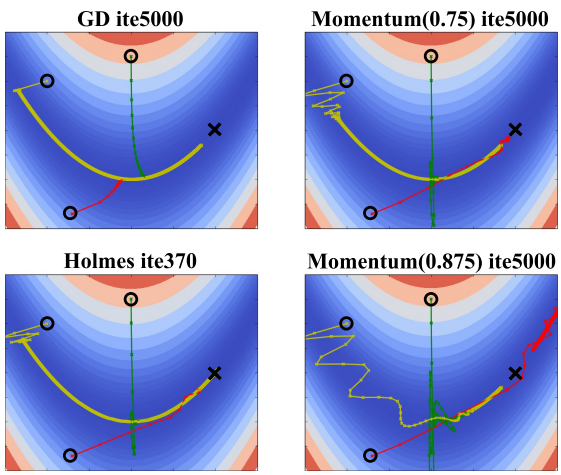


**Fig. 5** Rosenbrock function, starting from location ◯ and searching for the lowest value (location ×). Red, green, and yellow represent the search paths with different starting positions. Under these conditions, Holmes was able to reach the optimal solution after 370 iterations, but the gradient descent and momentum optimization methods were not able to reach the solution even after 5000 iterations. In addition, when the hyperparameter $\beta$ of momentum optimization method was 0.875, the movement was found to be too large to be stable.

where n is a number of dimensions; in this case $n = 2$.

Figures 4 and 5 show the path and the number of iterations (maximum iterations = 5000) to reach the optimal solution using the gradient descent, momentum optimization method, and Holmes. Red, green, and yellow indicate the paths followed by the search with different starting positions. The momentum optimization method was investigated with two different hyperparameters, 0.75 and 0.875.

**Table 2** Common hyperparameters when comparing the accuracy of three optimizers: Holmes, momentum optimization method, and mini-batch SGD.

| | |
|---|---|
| Activation function | Sigmoid |
| Mini-batch size | 32 |
| Network size | input 784, hidden 128, output 10 |
| Dataset | MNIST dataset |
| Learning rate | 0.25 |
| Number of quantization bits | 16bit (sign 1, integer 2, decimal 13) |

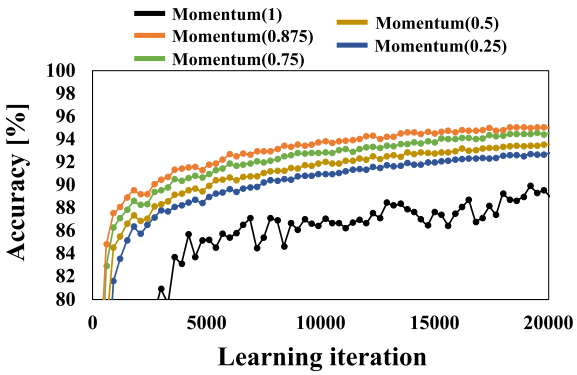


**Fig. 6** Variation in accuracy when changing the hyperparameter $\beta$, which is specific for the momentum optimization method. The other hyperparameters are listed in Table 2. The data points were plotted after every 300 iterations. The values in parentheses in the legend indicate the magnitude of $\beta$. The highest accuracy was obtained when $\beta = 0.875$.

In Fig. 4, on the three-hump camel function, gradient descent used 2000 iterations to reach the optimal solution, while Holmes and momentum optimization methods did not display much difference for the number of iterations. However, in Fig. 5, for the Rosenbrock function, Holmes was significantly faster than the gradient descent and momentum optimization methods. Holmes was faster than the gradient descent and momentum optimization methods in removing the learning oscillation making it very stable compared to the momentum optimization method with a large $\beta$.

### 4.2 Beta of the Momentum Optimization Method

The momentum optimization method used for the comparison has a specific hyperparameter $\beta$. First a search was conducted for the $\beta$ that is most suitable for the MNIST dataset used in this study. Hyperparameters other than $\beta$ are shown in Table 2. Because our objective is implementation in edge computing, parameters that are easy to express were selected by combining the powers of two. As displayed in Fig. 6, $\beta = 0.875$ is the most appropriate for the MNIST dataset. This value was used in subsequent evaluations.

### 4.3 Loss of Training Data

Subsequently, the difference in convergence speed of mini-batch SGD, Holmes, and momentum optimization methods were compared using "loss per iteration", where loss is the difference between the outputs of the neural network and the teacher data of the training data.

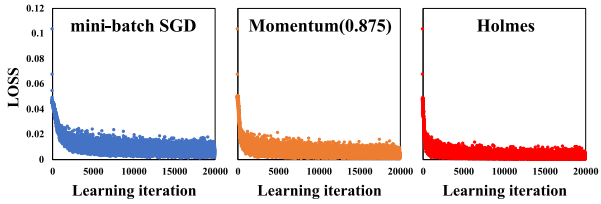


**Fig. 7** Comparison of the loss changes for the three optimizers: Holmes, momentum optimization method, and mini-batch SGD. It is observed that the speed at which the loss decreases is faster for Holmes, momentum optimization method, and mini-batch SGD, in that order.

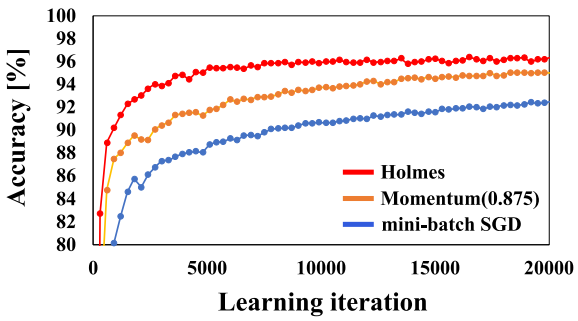


**Fig. 8** Comparison of the accuracy of the three optimizers: Holmes, momentum optimization, and mini-batch SGD. It it observed that Holmes achieves higher accuracy than the momentum optimization method and mini-batch SGD. The hyperparameter $\beta$ of the momentum optimization method was selected to be the one with the highest accuracy among the results of the search shown in Fig. 6. The other hyperparameters are displayed in Table 2. Data points are plotted after every 300 iterations.

In Fig. 7, the loss is almost the same at the beginning of the training, but as training progresses, the loss is different for each optimizer. It is observed that the speed of loss reduction is faster for Holmes, momentum optimization, and mini-batch SGD, in that order. However, it cannot be claimed that Holmes is the best optimizer because the training data loss is small. This alone does not prove that the performance can match that of the new data used along with the training data (high generalization performance).

### 4.4 Accuracy of Test Data

Finally, the differences in the generalization performance of mini-batch SGD, Holmes, and momentum optimization methods were compared using the "accuracy of test data." After every 300 iterations, the training (for updating internal parameters such as weights and momentum vectors) was stopped and a test dataset different from the training dataset was used to obtain the percentage of correct answers.

In Fig. 8, it can be observed that Holmes outperformed the mini-batch SGD and momentum even when being validated on the test dataset. The difference in performance is particularly noticeable when the number of learning iterations is small. In nonvolatile memory, where the number of memory accesses has a significant impact on the power consumption, it is extremely encouraging to observe that the number of learning iterations before convergence is low.

This demonstrates that Holmes is a good match for edge computing.

## 5. Discussion

In the evaluation section, the "number of memory accesses for nonvolatile memory," which is the most power-intensive part of the edge computing was evaluated. In this section, Holmes is compared with momentum optimization, RMSProp, and Adam with respect to other important aspects of edge computing, such as memory requirements and required operations [18].

### 5.1 Required Memory

The value after logarithmic quantization can be expressed only with the information of "sign" and "the position where 0 and 1 are flipped in the number." This means that in the momentum optimization method, the amount of memory required to store the momentum vector is $w \times$ bit, where bit is the number of quantization bits, whereas, in Holmes, it is $\log_2(w \times \text{bit})$. However, RMSProp does not have a momentum vector, and it needs to store the value $v$, which is the sum of the squares of the past gradients. The memory required to hold the value of the gradient squared is $2 \times w \times$ bit. On the other hand, Adam must store both the momentum vector and the value $v$. Therefore, the memory requirements are smaller for Holmes, momentum optimization, RMSProp, and Adam, in that order.

### 5.2 Required Operations

Here, the required operations are considered. Table 3 shows the number of operations required when each optimizer is used with respect to the mini-batch SGD. Holmes has more phases to add momentum vectors to the weights, whereas the momentum optimization method has more phases to add momentum vectors to the weights and multiply the momentum vectors by the hyperparameter $\beta$. However, because addition and multiplication are also used in mini-batch SGD, Holmes and momentum optimization methods can use the same operators.

By contrast, RMSProp and Adam have more square root operations and divisions along with addition and multiplication. Square root operations and divisions are not used in mini-batch SGD. Implementing square root and division in hardware requires a lot of resources and time and is not suitable for edge computing with limited resources [19].

**Table 3** Comparison of resource requirements of Holmes, momentum optimization method, RMSProp, and Adam with respect to mini-batch SGD. The comparison is made in three categories: required memory, required operations, and number of hyperparameters. Each cell shows the increase compared to mini-batch SGD.

| | Memory | Operations | Hyperparameter |
|---|---|---|---|
| mini-batch SGD | – | – | – |
| Holmes | $\log_2(w \times \text{bit})$ | addition | ±0 |
| Momentum | $w \times$ bit | addition<br>multiplication | +1 |
| RMSProp | $2 \times w \times$ bit | addition<br>multiplication<br>square root<br>division | +1 |
| Adam | $3 \times w \times$ bit | addition<br>multiplication<br>square root<br>division | +2 |

### 5.3 Hyperparameters

Hyperparameters are parameters that must be manually set by the user. As the appropriate value changes depending on the problem to be solved, the more hyperparameters there are, the more the burden on the user increases [20]. The $\beta m_{t-1}$ part of the momentum optimization method is $2^{\lfloor \log_2(m_{t-1}) \rfloor}$ in Holmes, and $\beta$ is missing. It is observed that Holmes has one less hyperparameter than the momentum optimization method. In addition, RMSProp does not have a momentum vector but uses another hyperparameter to prevent the learning rate from becoming extremely small. Adam also has a momentum vector and the same hyperparameters as RMSProp.

Therefore, momentum optimization and RMSProp have one hyperparameter, whereas Adam has two hyperparameters, and Holmes has even fewer hyperparameters.

### 5.4 Characteristic Parts

The momentum optimization method and Holmes both use momentum vectors. In the momentum optimization method, the entire momentum vector is multiplied by a constant value $\beta$ to prevent the momentum vector from becoming extremely large. Similarly, in Holmes, there is an operation to prevent the momentum vector from becoming extremely large; however, rather than multiplying by a constant value, Holmes combines logarithmic quantization with a limit on the number of quantization bits under a fixed-point number. This operation does not simply remove a small amount of momentum vector, but the percentage of reduction varies according to the size of the elements in the momentum vector. This method of "adjusting each element according to its size" is also used in RMSProp. However, the target of the automatic adjustment is the momentum vector in Holmes, and the learning rate in RMSProp. In addition, RMSProp achieves automatic adjustment by dividing the learning rate by the square root of the sum of the squares of the gradients. Adam combines the properties of both the momentum optimization method and RMSProp.

### 5.5 Hardware Design for Backpropagation

Because the previous study by Kaneko et al. [1] included a hardware version of the mini-batch SGD, that configuration was adopted to construct a block diagram that adapts the momentum optimization method and Holmes. Compared

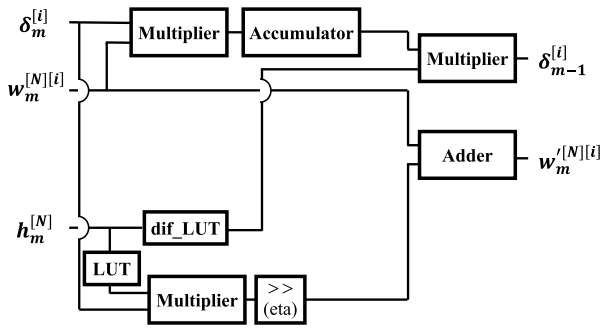


**Fig. 9** Mini-batch SGD's update module for backpropagation is rewritten based on previous study by Kaneko et al.; $h$ is the output of the previous layer; $\delta$ is the error gradient for the previous layer; and >> is a bit shift substitution for the learning rate. LUT contains an activation function in the form of a lookup table, and dif_LUT contains the derivative of the activation function in the form of a lookup table.

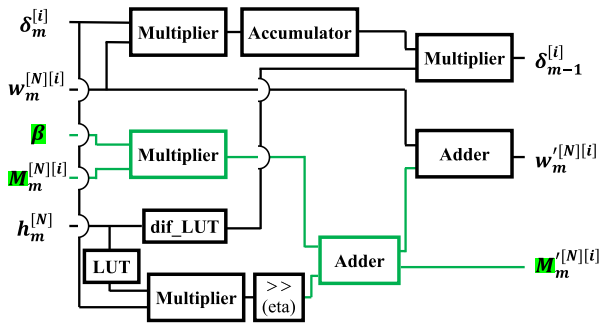


**Fig. 10** The update module of the momentum optimization method for backpropagation. The green area shows the additional operation compared to Fig. 9.

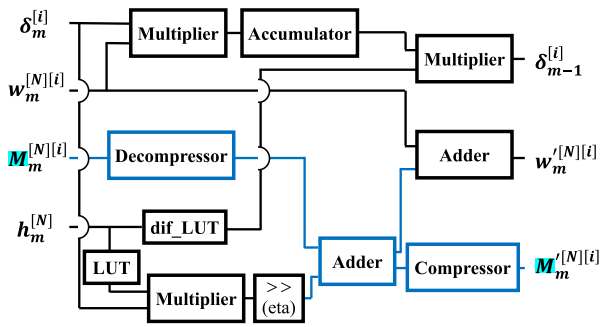


**Fig. 11** Holmes' update module for backpropagation. The blue area shows the additional operation compared to Fig. 9. The compressor converts the input value into the minimum information necessary to represent the value after logarithmic quantization. The decompressor converts that information into a value.

with the previous study shown in Fig. 9, the momentum optimization method is shown in Fig. 10. It adds memory to store the momentum vector, hyperparameter $\beta$, and multiplication and addition phases. The size of the memory used to store the momentum vector is denoted by $w \times$ bit. Holmes also has additional memory to store the momentum vector, as illustrated in Fig. 11; however, because it is compressed by Holmes quantization, the amount of memory required for storage is reduced to $\log_2(w \times \text{bit})$. In addition, no hyperparameters are added, but instead, a compressor and an expander using logarithmic quantization. The compressor is the first place where 0 and 1 are inverted, starting from the high-order bit, and information is output on the "most high-order bit (sign bit)" and the "position on the lower bit side of the place where 0 and 1 are inverted (inversion position)." When the code bit is 0, the decompressor outputs a numerical value in which only the inversion position is set to 1, and when the code bit is 1, it outputs a numerical value in which everything above the inversion position is 1, and everything below the inversion position and the inversion position is 0.

## 6. Conclusion

Various optimizers have been devised to improve the performance of neural networks. However, only a few of these can be applied to edge computing. One of the reasons is that the optimizer theory is not designed with hardware in mind. When implementing an optimizer for edge computing, there are considerations of "what type of computation hardware constraints," such as "how many quantization bits should be limited," and "what circuitry should be used," and these constraints are extensive. In other words, most existing optimizers require considerable resources (memory, computation, power, etc.) during implementation, which easily exceeds the constraints of edge computing.

Our new optimizer, Holmes, is designed for edge computing. Compared to other optimizers, it requires less memory, fewer operations, and fewer hyperparameters, making it more compatible with dedicated hardware, such as FPGAs and ASICs. In addition, by reversing the weakness of the hardware implementation, which is the "decrease in accuracy and learning convergence speed as a result of the quantization bit limitation," and by using the numerical expressions unique to hardware, dramatically improved convergence speed is obtained with Holmes.

## Acknowledgments

This study was supported by a JSPS Grant-in-Aid for JSPS Fellows and a Grant-in-Aid for Scientific Research on Innovative Areas [18H05288] from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan.

### References

[1] T. Kaneko, K. Orimo, I. Hida, S. Takamaeda-Yamazaki, M. Ikebe, M. Motomura, and T. Asai, "A study on a low power optimization algorithm for an edge-AI device," NOLTA, vol.10, no.4, pp.373–389, 2019.

[2] R. Bez and A. Pirovano. "Non-volatile memory technologies: emerging concepts and new materials," Materials Science in Semiconductor Processing, vol.7, no.4-6, pp.349–355, 2004.

[3] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA based neural network accelerator," arXiv preprint, arXiv:1712.08934, 2017.

[4] A. Ankit, I.E. Hajj, S.R. Chalamalasetti, G. Ndu, M. Foltin, R.S. Williams, P. Faraboschi, W.-M.W. Hwu, J.P. Strachan, K. Roy, and

D.S. Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," In: ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, pp.715–731, 2019.
[5] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "LNPU: A 25.3TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16," In: 2019 IEEE International Solid-State Circuits Conference (ISSCC) Session 7.7, pp.142–144, 2019.
[6] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C.-Y. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S.-H. Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P.-F. Lu, B. Curran, L. Chang, and K. Gopalakrishnan, "A scalable Multi-TeraOPS deep learning processor core for AI training and inference," In: 2018 Symposium on VLSI Circuits Digest of Technical Papers C4-2, pp.C35–C36, 2018.
[7] D. Han, J. Lee, J. Lee, and H.-J. Yoo, "A 1.32 TOPS/W energy efficient deep neural network learning processor with direct feedback alignment based heterogeneous core architecture," In: 2019 Symposium on VLSI Circuits Digest of Technical Papers C24–3, pp.C304–C305, 2019.
[8] C. Kim, S. Kang, D. Shin, S. Choi, Y. Kim, and H.-J. Yoo, "A 2.1TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression," In: 2019 IEEE International Solid-State Circuits Conference (ISSCC), session 7.4, pp.136–138, 2019.
[9] T. Kaneko, H. Momose and T. Asai, "An FPGA accelerator for embedded microcontrollers implementing a ternarized backpropagation algorithm," In: 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp.1–8, 2019.
[10] K. Wakabayashi, "Use of high-level synthesis to generate hardware from software - Another alternative general-purpose program-executing mechanism to the CPU," IEICE, vol.6, no.1, pp.37–50, July 2012.
[11] D. Choi, C.J. Shallue, Z. Nado, et al., "On empirical comparisons of optimizers for deep learning," ICLR 2020 Conference Blind Submission, June 2020.
[12] L. Bottou, "Large-scale machine learning with stochastic gradient descent," COMPSTAT'10, pp.177–186, Aug. 2010.
[13] B.T. Polyak, "Some methods of speeding up the convergence of iteration methods," USSR Computational Mathematics and Mathematical Physics, vol.4, no.5, pp.1–17, 1964.
[14] N. Qian, "On the momentum term in gradient descent learning algorithms," Neural Networks, vol.12, no.1, pp.145–151, 1999.
[15] T. Tieleman and G. Hinton, Lecture 6.5–RmsProp: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural Networks for Machine Learning, 2012.
[16] Y. Nesterov, "A method for solving the convex programming problem with convergence rate O($1/k^2$)," Doklady AN USSR, vol.269, pp.543–547, 1983.
[17] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," Journal of Machine Learning Research, vol.12, pp.2121–2159, 2011.
[18] D.P. Kingma and J. Ba, "Adam: A method for stochastic optimization," In: The International Conference on Learning Representations (ICLR), 2015.
[19] H. Mitome, P.M. Yan, and R. Ishii, "Hardware Design of Dividers and It's Evaluation," The Institute of Electrical Engineers of Japan, C, A publication of Electronics, Information and Systems Society, vol.116, no.5, pp.534–539, 1996.
[20] B. Neyshabur, R. Tomioka, and N. Srebro, "In search of the real inductive bias: On the role of implicit regularization in deep learning," In: International Conference on Learning Representations (ICLR), 2015.
[21] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," IEEE, vol.91, no.4, pp.489–502, April 2003.

**Yoshiharu Yamagishi** received B.E. in electronic engineering from Hokkaido University, Sapporo, Japan, in 2020. He is currently pursuing his Masters degree at the Graduate School of Information Science and Technology, Hokkaido University. His current research interests are in machine learning in edge computing, mainly in reinforcement learning and neural network optimizers.

**Tatsuya Kaneko** received the B.E. and M.E. degrees from Hokkaido University, Sapporo, Japan, in 2018 and 2020, respectively. He is currently pursuing a Ph.D at the same university. His current research interests include a hardware-aware training algorithm and its hardware architecture for an edge device.

**Megumi Akai-Kasaya** completed her Ph.D. in physical chemistry from Osaka University in 1997. She has been a professor at the Faculty of Information Science and Technology at Hokkaido University since 2020 and has concurrently become a professor at the Graduate School of Science, Osaka University, in 2021. Her scientific interests include nonequilibrium (or dynamic) systems, carrier transport in nanostructured soft materials, and development of new functional devices utilizing nanoscale physical properties.

**Tetsuya Asai** received the B.S. and M.S. degrees in electronic engineering from Tokai University, Hiratsuka, Japan, in 1993 and 1996, respectively, and a Ph.D. degree from Toyohashi University of Technology, Toyohashi, Japan, in 1999. He is currently a professor with the Graduate School/Faculty of Information Science and Technology, Hokkaido University, Sapporo, Japan. His current research interests include developing intelligent integrated circuits and their computational applications, emerging research architectures, deep learning accelerators, and very large-scale device-aware neuromorphic integrations.