https://doi.org/10.31449/inf.v49i1.7145 Informatica **49** (2025) 53–60 **53**

Learning Algorithm for LesserDNN, a DNN with Quantized Weights

Masashi Takemoto[1], Yasutake Masuda[1], Jingyong Cai[2] and Hironori Nakajo[2]
[1]BeatCraft, Inc. Tokyo Japan
[2]Tokyo University of Agriculture and Technology Tokyo, Japan
E-mail: lesser@beatcraft.com, masuda@beatcraft.com, kkkluoruo@hotmail.com, nakajo@cc.tuat.ac.jp

Keywords: Deep neural network, machine learning, simulated annealing, weight quantization

Received: September 12, 2024

This paper presents LesserDNN, a model that uses a set of floating-point values {-1.0, -0.5, -0.25, -0.125, -0.0625, 0.0625, 0.125, 0.25, 0.5, 1.0} as quantized weights, and a new learning algorithm for the proposed model. In previous studies on deep neural networks (DNNs) with quantized weights, because DNNs employ the gradient descent method as their learning algorithm, quantized weights were applied only during the inference stage. Due to differentiability properties, quantized weights cannot be used when the gradient descent method is applied during training. To address this issue, we devised an algorithm based on simulated annealing. Since simulated annealing has no differentiability requirements, LesserDNN can utilize quantized weights during training. With the use of quantized weights and this simulated annealing-based algorithm, the learning process becomes a combinatorial problem. The proposed algorithm was applied to train networks on the MNIST handwriting dataset. The tested models were trained with the simulated annealing-based algorithm and quantized weights, achieving the same level of accuracy as gradient descent-based comparison methods. Additionally, we conducted tests using the CIFAR-10 dataset, and achieved the good results to demonstrate the algorithm. Thus, LesserDNN has a simple design and small implementation scale because backpropagation is not applied. Moreover, this model achieves a high accuracy

Povzetek: LesserDNN je model globoke nevronske mreže z utežmi, kvantiziranimi na nabor vrednosti -1,0; -0,5; -0,25; -0,125; -0,0625; 0,0625; 0,125; 0,25; 0,5; 1,0. Za učenje tega modela so razvili algoritem, ki temelji na simuliranem žarjenju, saj gradientni spust zaradi lastnosti diferencirljivosti ni primeren za kvantizirane uteži. Ta pristop omogoča uporabo kvantiziranih uteži že med učenjem.

1 Introduction

Deep neural networks (DNNs) have contributed substantially to the development of machine learning and various machine learning-based applications. To improve accuracy, deeper networks, more complex structures, and more neurons have been introduced in various models. However, these large models are not suitable for deployment in small devices. To address this issue, weight quantization approaches have been introduced. To reduce model size while maintaining accuracy, less precise weights are used only for inference, while training still requires full-precision weights. Weight quantization successfully reduces the memory requirements and computational costs of DNNs while maintaining the same accuracy as full-precision weight DNNs. Lower precision weights are not applied during the DNN training process because of the learning algorithm, namely, the gradient descent method. To fully utilize less precise weights, a new learning algorithm must be developed.

Therefore, we devised LesserDNN, a novel approach that uses a set of floating-point values {- 1.0, -0.5, -0.25, -0.125, -0.0625, -0.0625, 0.125, 0.25, 0.5, 1.0} as quantized weights instead of arbitrary values such as continuous floating-point numbers, where w$\leqq$1.0 and w$\geqq$-1.0. (w: a weight). Moreover, we developed a new learning algorithm. We implemented a framework that utilizes these quantized weights during both the training and inference stages; thus, LesserDNN can build arbitrary networks of freely stacked layers containing neurons with quantized weights.

In conventional DNNs, the weights are set as arbitrary real numbers between -1.0 and 1.0, and infinite combinations of these weights are possible during training. On the other hand, in contrast to traditional DNNs, the learning process of LesserDNN is formulated as a combinatorial problem because the number of combinations of quantized weights is finite.

Although the number of combinations is mathematically finite, it is intractable for typical computer systems; thus, it is impossible to search for the optimal solution in a round-robin fashion within a practical time frame. Therefore, we devised a learning algorithm using simulated annealing (SA). SA imitates the most settled arrangement of heated molecules as they cool, with the weights representing the molecules and the number of selections representing the temperature. Randomly selected weights are changed and updated only when these changes improve the results. The

combinations are evaluated according to the difference between the current result and the correct answer. The optimization proceeds by iterating; thus, this difference decreases, and the global optimum is eventually reached. The background of this study is described in Section 2. LesserDNN is explained in Section 3, and the details of the algorithm are presented in Section 4. The experiments conducted on the MNIST handwriting dataset are explained in Section 5, and the results are reported in Section 6, The results of additional experiments on the CIFAR-10 dataset is explained in Section 7. Then, we have discussions in Section 8.

The code of LesserDNN is available online at `https://github.com/BeatCraft/LDNN`, and the code for experiments also is available online at `https://github.com/BeatCraft/LDNN-mnist`.

2 Background

Weight quantization has been proposed as a method of increasing the inference speed while reducing memory footprint. Weight quantization involves statistically analyzing trained DNNs and replacing values. In previous studies, Vanhoucke et al. [1] initially applied weight quantization to reduce the computational burden of DNNs. In each layer, the weights were normalized to a signed integer in the range of -127 to 127, and the activations were quantized as 8-bit integers. As a result, the total memory footprint of the improved network was approximately 3 or 4 times smaller than that of the original network. Compared with DNNs, networks that applied quantization exhibited recognizable improvements in terms of speed while maintaining accuracy.

Similar to the initial study, many weight quantization studies have applied dynamic range quantization; however, there are some notable studies on fixed-point weight quantization. Courbariaux et al. [4] introduced BinaryConnect, in which the weights are aggressively reduced to a single bit (-1 or 1). Because of these binarized weights, approximately two-thirds of the multiplication operations can be replaced by addition and substitution operations. As the calculations are greatly simplified, the training speed is 3 times faster, yet the accuracy is reduced by only 19%. To improve the accuracy of BinaryConnect, Li et al. [5] developed ternary weight networks. These networks added zero as a binarized weight and introduced a threshold-based ternary function to transform full-precision weights to ternary weights. The threshold value for the ternary weights was determined by optimizing the threshold-based ternary function. This function minimizes the Euclidian distance between full-precision weights and ternary weights with a scaling factor. Since ternary weight networks require 2-bit storage for each weight unit, their model compression rates are higher than those of full-precision weight models. The experimental results showed that when compared with full-precision weight models, the accuracies of ternary weight networks were reduced by 0.4% or less.

The results of weight quantization experiments have been remarkable thus far; however, previous studies have had difficulty testing weight quantization models with quantized weights only. In general, these studies statistically analyze trained DNNs and replace values; thus, training is initially performed with full-precision weights. The full-precision weights are replaced by quantized weights after training, and these quantized weights applied only during the inference stage to retain high accuracy.

Vanhoucke et al. [1] applied quantization in pretrained networks. Courbariaux et al. [4] and Li et al. [5] employed binary and ternary weights (lower precision weights) in forward and back propagation and applied full-precision weights in the gradient method. In these weight quantization studies, quantized weights were not fully used during training.

Weight quantization approaches require full-precision weights because the gradient descent method is used the optimization method in the training process. Because of differentiability properties, DNNs are not differentiable with respect to quantized weights. As the changes in the weights are large, the gradient descent method cannot accurately determine the gradient and thus cannot differentiate changes in the weights. Thus, quantized weights cannot be used in the gradient descent method. To address this issue, this paper applies a non-gradient-based approach, namely, an SA-based method.

SA is known as a combinatorial optimization method. In 2017, Mousavi et al. [3] applied SA to an artificial neural network (ANN). In their empirical analysis, the ANN/SA model outperformed the ANN model. However, as discussed in the paper, the ANN/SA model had large time costs. As the weights are expressed as floating-point numbers, the number of combinations in the model is enormous and practically infinite, and the computations cannot be completed within a practical time range.

The combinatorial optimization method also faces this issue. For instance, as an implementation limitation, the number of combinations is varied between -1.0 and 1.0 in increments of 0.01, which is considered the precision level of the weights. The number of combinations is 200^n, where n is the number of weights in the model. For example, a small DNN with 10000 weights contains 200^{10000} combinations. Considering the large number of combinations, small DNNs with medium-precision weights have reduced time costs while retaining the large computational power required to complete the training process. In typical experiments, small DNN models have more than 1000 weights, and the number of combinations easily surpasses this example.

To address this issue, quantized weights are introduced. The quantized weights are designed as a set of sequential numbers, where the current number is divided by 2 to determine the subsequent number, such as {-1.0, -0.5, -0.25, -0.125, -0.0625, 0.0625, 0.125, 0.25, 0.5, 1.0}, because we found that logarithmic quantization of the weights achieved

better classification results than linear quantization of the weights in a previous study [2]. This approach reduces the level of precision of each weight from 200 to 10.

Thus, when quantized weights are used, the total number of combinations in the DNN is 10^{10000}. While this is still a large number, it is 20^{10000} times smaller than 200^{10000}, and current computer systems can likely handle this number of combinations. When an SA-based algorithm is applied to a DNN, a set of quantized weights can be used for training because the gradient method is not used as the optimizer.

3 LesserDNN

LesserDNN includes layers, neurons, and weights and has basic mechanisms such as inference, activation functions, loss functions, and batches. LesserDNN is similar to existing DNNs, except that LesserDNN has no backpropagation. These characteristics and other differences, such as the lack of bias, and how batches are handled, are explained in this chapter.

The output of a neuron in a DNN is calculated by adding a $bias$ to the summation of the product, as shown in (1), where x is the input value and w are the weights; then, the output value is passed to an activation function such as ReLU (2). ReLU and leaky ReLU (3) can be selected as activation functions in each layer. In (1), y is the output of the neuron, and a in (3) is a coefficient that depends on the actual network applied in a given problem.

$$y = \sum_{i=1}^{n}(x_i \cdot w_i) + bias \tag{1}$$

$$f(x) = \begin{cases} x & (x \geqq 0) \\ 0 & (x < 0) \end{cases} \tag{2}$$

$$f(x) = \begin{cases} x & (x \geqq 0) \\ a \cdot x & (x < 0) \end{cases} \tag{3}$$

The bias has two functions: adjusting the output range of the neuron and ensuring that the output value is differentiable. Ensuring that a value is differentiable means guaranteeing that that value is not set to zero. LesserDNN has no bias since there is no need to consider whether the output of the neuron is differentiable since the gradient descent method is not used for training and layer normalization is applied. Layers containing neurons use a function to normalize the output. Inference is executed by performing calculations sequentially starting with the input layer. A layer receives the output of the previous layer, executes calculations and passes the results to the next layer. This basic propagation process is repeated until the output layer is reached. In the output layer, the softmax function may be applied. For classification problems, the output of the final layer is the probability. The softmax function adjusts the individual values in the output layer to ensure that the sum of the outputs is equal to 1.0.

The loss function is a function that is used to determine the magnitude of the discrepancy between the correct and predicted values. LesserDNN selects either the mean squared error (MSE) or the cross entropy (CE) to evaluate the model. The MSE is defined in (4), where n is the number of data points, y is the correct value, and $\hat{y}$ is the predicted value. In classification problems, n is the number of classes. The CE is defined in (5), where $p(x)$ is the correct probability and $q(x)$ is the predicted probability.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4}$$

$$H(p,q) = -\sum_{x} p(x) \cdot \log q(x) \tag{5}$$

The calculations are performed in each layer, and basic functions are implemented to take advantage of hardware systems such as GPUs or multicore CPUs. LesserDNN adapts CUDA [10] and OpenCL [9] to support GPUs. CUDA is an advanced computation library for NVIDIA [8] GPUs. OpenCL is an open standard for parallel programming on systems with different types of hardware, such as multicore CPUs, FPGAs, and GPUs. The input data for training and testing are grouped in two-dimensional arrays and treated as batches. The inference results on the batched input data are evaluated in bulk, and the average of the results is considered the optimization improvement for that batch.

A batch can be assigned to multiple processes by adapting a message passing library (MPI). An MPI is a standard for distributed-memory parallel processing, and Open MPI [7] is available as an implementation. Open MPI supports data transfer and synchronization between processes not only on single computers but also on multiple computers connected in a network. Thus, very large batches that consume considerable working memory can be handled.

4 Learning algorithm

The learning algorithm of LesserDNN includes triple nested loops, challenge loops, cooling loops, and a main loop, as shown in Fig. 1. The challenge loops determine which weights to update, and the cooling loop is crucial for applying the SA function in the learning algorithm. The cooling loop controls the temperature, which represents the number of weights to update during each iteration. Then, the main loop iterates to the next cooling loop. More iterations are required for more difficult problems.

4.1 Challenge loop

The challenge loop is the innermost loop and the core of the algorithm as shown in Fig. 2, and the pseudocode is shown in Algorithm 1. It includes four basic functions: selecting

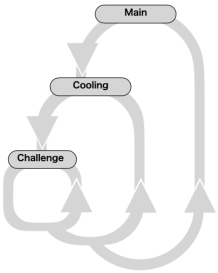

Figure 1: The triple nested loops

the weights, updating the network, evaluating the network, and reversing changes.

Before the loop, all weights in the LesserDNN network are initialized by randomly selecting a value in the set {-1.0, -0.5, -0.25, -0.125, -0.0625, -0.0625, 0.125, 0.25, 0.5, 1.0}. Inference is performed on the training data, and the difference between the current result and the correct answer is calculated as the loss function, Λ using either the MSE or CE. The loop takes a positive integer as a variable, and the number of weights to handle during each iteration is N. Then, N weights are randomly selected and changed, and the loss function, λ, is obtained. Since the training data include batches with multiple data points, multiple values are obtained. The average of these values is determined as the evaluation result for that batch. If λ is smaller than Λ,, the changed weights are maintained; otherwise, the changes are discarded. The loop stores ρ, which is the ratio of the number of successful updates to the total number of trials. The minimum number of iterations is set to 50 by default, and the loop is terminated when the hit rate falls below 1%.

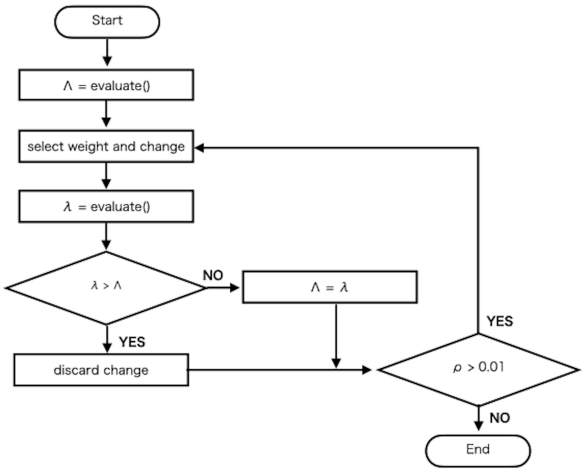

Figure 2: Flowchart of the challenge loop

Algorithm 1 Challenge Loop

```
1:  function challenge(Λ, N)
2:      cnt ← 0
3:      hit ← 0
4:      ρ ← 1.0
5:      while ρ > 0.01 do
6:          select(N)          ▷ select N weights randomly
7:          update()
8:          λ ← evaluate()
9:          if λ < Λ then
10:             hit ← hit + 1
11:             Λ ← λ
12:             keep the changes
13:         else
14:             discard the changes
15:         end if
16:         cnt ← cnt + 1
17:         ρ ← hit/cnt
18:     end while
19:     return Λ
20: end function
```

4.2 Cooling loop

The cooling loop controls N in the iterations of the challenge loop, as shown in Fig. 3, and the pseudocode is shown in Algorithm 2. N is the number of weights to be updated at once in the challenge loop. N is calculated according to the total number of weights M and the temperature in the cooling loop K.

The maximum value of temperature, θ is also calculated according to M. In the cooling loop, K decreases from θ to 0 as the temperature of the heated material decreases over time. In the cooling loop, K decreases as N decreases. We set the maximum number of weights to be changed during each iteration to 1%, and the maximum temperature, θ, was calculated as $\theta = \log_{\epsilon}(0.01 \cdot M)$, where ϵ is set to 2 as a default value.

Therefore, the number of weights to be changed at once at temperature k is $N = \epsilon^k$. ϵ is a hyperparameter that controls the convergence speed. The larger the value of ϵ is, the more rapidly the optimization process proceeds; thus, the possibility of falling into a local optimum increases. The appropriate range for ϵ is 1.1 to 2.0. The cooling loop starts at θ and is repeated until K reaches 1.

4.3 Main loop

The main loop is a loop that simply iterates the cooling loop, as shown in Algorithm 3. The number of iterations varies depending on the complexity of the problem. The number of iterations also depends on the number of training samples (batch size). Because LesserDNN is an SA-based method, the number of iterations is a hyperparameter.

In LesserDNN, the cycle of select(), update(), and evaluate() in the challenge loop must be performed sequentially. The computational costs of the main loop and cooling loop

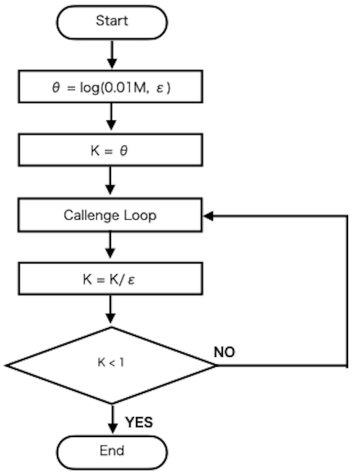

Figure 3: Flowchart of the coolong loop

Algorithm 2 Cooling Loop

```
function Cooling
    M ← total number of the weights
    ϵ ← 2
    θ ← log_ϵ(0.01 · M)
    Λ ← evaluate()
    K ← θ
    while K >= 1 do
        N ← int(K)
        Λ ← challenge(Λ, N)
        K ← K/ϵ
    end while
end function
```

are low and may not change with the complexity of the problem or the batch size. The evaluation function, which includes the inference and loss functions, has the largest computational cost. The evaluation function can be parallelized by assigning divided batches to multiple processes, thus allowing LesserDNN to maintain the cycle in a single sequence and distribute the computational costs to as many processes as the hardware allows.

Algorithm 3 Main Loop

```
function Main(n)                ▷ n : number of iterations
    for i < n do
        cooling()
    end for
end function
```

5 Experiments

To validate the learning algorithm and examine the characteristics of LesserDNN, we conducted experiments using an MNIST dataset.

We constructed a network with the same layer and neuron configuration in TensorFlow and performed the same experiments for comparison. The performance was evaluated with stochastic gradient descent (SGD) and adaptive moment estimation (Adam) as backpropagation algorithms. The SGD algorithm is a first-order iterative optimization algorithm for determining the local minimum of a differentiable function. The strategy involves determining the steepest descent in a large or infinite space. By repeatedly applying the strategy, the algorithm eventually finds a local minimum. Adam was developed by Kingma et al. [6]. Adam is a stochastic gradient-based optimization method that calculates the exponential average of the gradient and the squared gradient and adapts the learning rate for each weight in the neural network. The hyperparameters control the decay rates of these moving averages. The moving averages are estimated according to the mean of the uncertain variance of the gradient.

The MNIST dataset is a well-known classification example in machine learning that contains 60000 training images and 10000 test images. These images are 28×28 pixel 8-bit grayscale images. All the images contain handwritten figures, namely, the digits 0 to 9, shown as Fig. 4. DNNs are trained with the training images and evaluated with the test images. A very basic structure, namely, a fully connected network with 2 hidden layers, was used. The input layer had 784 neurons to transform the 28×28 8-bit grayscale image to 784 floating-point values in the range of 0.0 to 1.0. The output layer had 10 neurons corresponding to the 10 classes.

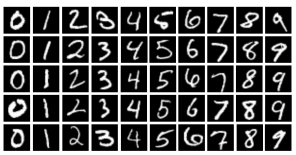

Figure 4: Examples of MNIST Dataset

5.1 Network size

The number of neurons in the two hidden layers was initially set to 256 and then changed to 128, 64, and 32.

5.2 Number of iterations

The number of iterations was set to 100. The accuracy and CE of the network, which has two hidden layers with 256 neurons, were determined with a learning algorithm that was iterated 100 times.

5.3 Training batch size

The networks with two 256-neuron hidden layers were trained on MNIST datasets with different numbers of images: 250, 500, 750, 1000, 2000, 3000, 4000, 5000, 10000,

20000, 40000, and 60000. The number of iterations was set to 100 for all configurations.

5.4 Base of temperature, ϵ

ϵ is a hyperparameter of LesserDNN that controls the speed of the temperature descent during each iteration of the SA-based learning algorithm. The networks with two 256-neuron hidden layers were trained with different ϵ values, including 2.00, 1.50, 1.25, and 1.10, to assess how this value influences training.

The number of iterations was set to 100 for all networks.

6 Results

6.1 Network size

Table 1 shows the results of training LesserDNN models with different network sizes and the results of equivalent networks in TensorFlow for comparison.

6.2 Number of iterations

Fig. 5 shows the changes in the accuracy and CE as the learning algorithm is iterated 100 times in the 256 network.

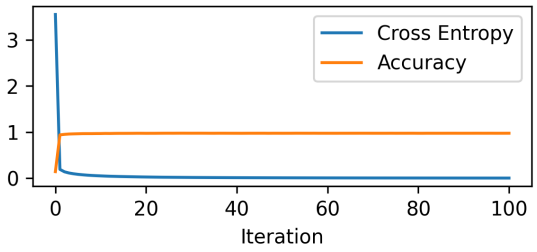

Figure 5: Changes in the accuracy and CE versus the number of iteration on the MNIST dataset

6.3 Training batch size

Table 2 shows the accuracies of networks trained with 250, 500, 750, and 1000 training images and the results of TensorFlow for comparison.

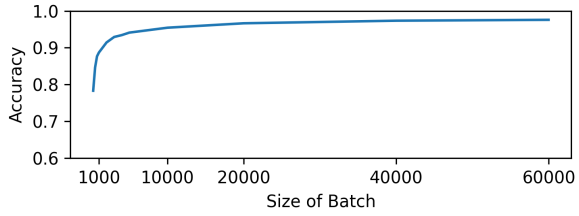

Figure 6: Changes in accuracy versus batch size for MNIST

6.4 Base of temperature, ϵ

Fig. 7 shows how the accuracy changes with the parameter, ϵ. The range of the Y axis varies from 0.90 to 1.00 because differences were observed only in this range.

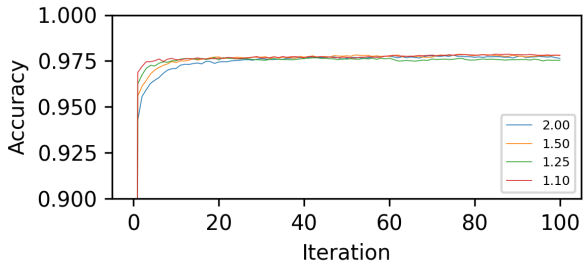

Figure 7: Accuracy versus ϵ

7 Additional experiments with the CIFAR-10 dataset

CIFAR-10 is a well-known image classification dataset with 60000 32×32 color images in 10 classes, including objects like airplanes, cars, and animals, shown as Fig. 8. Divided into 50000 training and 10000 test images, it is widely used to benchmark image recognition models due to the challenges posed by its diverse object orientations, lighting, and backgrounds.

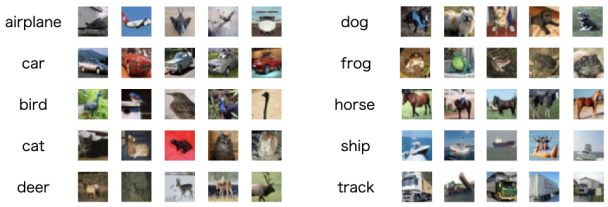

Figure 8: Examples of CIFAR-10 Dataset

DNNs with two hidden layers, each containing 64, 128, 256, and 512 neurons, were configured in both LesserDNN and TensorFlow for comparison. The accuracies achieved with the CIFAR-10 dataset were then compared between the two implementations, shown as Table3. In the TensorFlow model, features such as batch normalization and data augmentation were not used to ensure a direct comparison of the optimization algorithms.

8 Discussion

The experimental results showed that the SA-based learning algorithm can be applied to train LesserDNN, a DNN with quantized weights. Moreover, the fully connected

Table 1: Different network sizes for MNIST

	32	64	128	256
LesserDNN	0.9537	0.9675	0.9731	0.9776
TensorFlow/SGD	0.9618	0.9699	0.9734	0.9753
TensorFlow/Adam	0.9721	0.9789	0.9830	0.9848

Table 2: Different batch sizes for MNIST

	250	500	750	1000
LesserDNN	0.7832	0.8463	0.8766	0.8870
TensorFlow/SGD	0.6552	0.7689	0.8216	0.8347
TensorFlow/Adam	0.8082	0.8614	0.8850	0.8985

Table 3: Different network sizes for CIFAR-10

	64	128	256	512
LesserDNN	0.483200	0.4857	0.4706	0.4795
TensorFlow/SGD	0.4962	0.513	0.5231	0.5319
TensorFlow/Adam	0.5019	0.518	0.5333	0.5504

model achieved an accuracy comparable to that of existing DNNs on a classification task with the MNIST dataset.

The MNIST experimental results showed that LesserDNN has almost equivalent performance to TensorFlow, and we confirmed a decrease in accuracy of less than 1%. Furthermore, we confirmed that the accuracy decreased because the network size and batch size were reduced; however, even in such cases, considerable accuracy was maintained, indicating that a tradeoff between computational complexity and accuracy can be established.

When the batch size and number of iterations were fixed, the accuracy decreased slightly depending on the size of the network, as shown in Table 1. TensorFlow/SDG and TensorFlow/Adam both showed similar results. Thus, LesserDNN exhibits similar characteristics to DNNs.

Fig. 5 shows that no overfitting occurred in the experiments. After a certain number of iterations, the accuracy changed only slightly, although the loss function kept decreasing. The algorithm appears to switch between semi-optimal solutions of weights. The iterations should be terminated when the accuracy no longer increases. It is important that the method for determining the appropriate batch size be clear since the number of iterations is the hyperparameter of the LesserDNN model.

In Fig. 6, as the batch size varied from 60000 to 1000, the accuracy of LesserDNN decreased gradually from 97% to 88%. Then, when the batch size was 500, the accuracy decreased sharply to 84% and decreased further to 78% when the batch size was 250. The accuracy of TensorFlow decreased in the same trend with the both SDG and Adam. LesserDNN showed considerably equimbarent performance with TensorFlow.

ϵ moderates the learning rate. This parameter controls the numbers of neurons that are modified during each iteration in the cooling loop and is a major factor in determining the learning rate. A higher learning rate narrows the search direction, while a lower learning rate allows the search to proceed without narrowing the range of possibilities. Fig. 7 shows that the maximum accuracy does not change significantly as ϵ varies, although the accuracy in the 1st iteration of the main loop is higher when ϵ is smaller.

The additional evaluation on the CIFAR-10 dataset showed that the accuracy was lower than that of TensorFlow, but the results were still comparable.. The difference is minimal, especially when the network size is small. Since LesserDNN transforms learning into a weight combination problem, having fewer neurons makes the search easier and allows for more efficient use of computational resources.

The process of training LesserDNN iterates between inference and evaluation; thus, training is possible even in systems with limited resources, such as embedded systems and IoT devices. Additionally, training can be distributed among GPUs in the same system or among different systems over IP networks. Therefore, various systems can be constructed. For example, in one system, only inferences may be performed on edge devices, while training can be performed on GPUs in a cloud service.

9 Future work

We conducted performance comparisons with TensorFlow, for algorithm validation, and as a result, we did not precisely measure the computation times. However, there was a substantial difference, even in relatively simple problems like MNIST. This issue is expected to become more significant when dealing with more complex problems in the future. Therefore, it will be necessary to explore methods for efficient batch switching during training, similar to SGD.

In future work, we intend to address how to apply convolutional neural networks (CNNs) to LesserDNN as well as attempt to apply LesserDNN to more difficult problems, such as CIFAR-100 and Tiny ImageNet. In

this paper, we assure that the small fully connected networks in LesserDNN are sufficient for small problems such as MNIST. However, CNNs are important for applying LesserDNN to practical problems.

Another issue we intend to address is accelerating the hardware with the algorithm. Multiplications of the weight and input values of the neurons can be replaced by SHIFT operations since quantized weights are representable by a factor of 2. This is advantageous for creating efficient logics in FPGA and converting networks to ASICs.

One important concept of LesserDNN is that the training process solves a combinatorial problem; thus, LesserDNN should have a high affinity with emerging quantum computers. Quantum computers are known to be great for solving combinatorial optimization problems. Therefore, we will attempt to apply our algorithm in quantum computers in future work.

Acknowledgement

To Tadasuke Furuya Ph.D. of the Faculty of Marine Technology at Tokyo University of Marine Science and Technology, and Takehiko Kashiwagi from Parallel Networks LLC, we would like to express our sincere gratitude for the support.

References

[1] Vincent Vanhoucke, Andrew Senior and Mark Z. Mao. Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011.

[2] Jingyong Cai, Masashi Takemoto, and Hironori Nakajo. A Deep Look into Logarithmic Quantization of Model Parameters in Neural Networks. In Proceedings of the 10th International Conference on Advances in Information Technology (IAIT '18). Association for Computing Machinery, Article 6, 1–8, 2018. https://doi.org/10.1145/3291280.3291800

[3] Seyyed Mohammad Mousavi, Elham S. Mostafavi, Pengcheng Jiao. Next generation prediction model for daily solar radiation on horizontal surface using a hybrid neural network and simulated annealing method, Energy Conversion and Management, Volume 153, Pages 671-682, 2017. https://doi.org/10.1016/j.enconman.2017.09.040

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: training deep neural networks with binary weights during propagations. In Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15), Vol. 2, 3123–3131, 2015.

[5] B. Liu, F. Li, X. Wang, B. Zhang and J. Yan. Ternary Weight Networks. ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1-5, 2023 https://doi.org/10.1109/ICASSP49357.2023.10094626

[6] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, arXiv, 2017, https://doi.org/10.48550/arXiv.1412.6980

[7] Open MPI https://www.open-mpi.org/

[8] NVIDIA Corporation https://www.nvidia.com/

[9] OpenCL https://www.khronos.org/opencl/

[10] CUDA Toolkit https://developer.nvidia.com/cuda-toolkit/