技能・技術報告

耐故障化 BP 学習の実装に伴うディープラーニングフレームワークの機能拡張に関する研究

Deep Learning Framework Extensions with Implementation of Fault Tolerant BP Learning

アストリウィンドウサリ，堀田 忠義，秋葉 将和
Astriwindusari, Tadayoshi Horita and Masakazu Akiba

1. Introduction

1.1. Background

An artificial neural network (hereinafter referred as “neural network”) is an electrical circuit that imitates human brain nervous systems. It consists of artificial nodes called neurons. In a multilayer neural network, the previous layer node is interconnected with next layer by direct links called connection weights.

Neural networks have been studied since 1943 by McCulloch et al. McCulloch and Pitts were the first to implement a human brain in a hardware model. In 1985, Rosenblatt proposed a simple perceptron that have the ability to learn. In 1986, Rumelhart et al. are the first who proposed the Back Propagation (hereinafter referred as “BP”) algorithm[1].

Since then, the neural network field has developed and is extending become a new field called Artificial Intelligence (AI). Deep learning is one of AI technology and was developed based on multilayer neural networks.

On the other hand, the world of computer hardware is also evolving. A Graphics Processing Units (GPU) is a processor that specialized in image processing. GPUs are composed of thousands of cores for parallel processing so it can display images in screen. Nowadays, GPUs can be used for general-purpose programming as well as graphics. This is called GPGPU (General Purpose Computing on GPUs), and with the use of GPU high-speed processing by parallel computation could be obtained. In order to make GPGPU programming easier, NVIDIA inc. has developed a platform called CUDA (Compute Unified Device Architecture).

In this research, a neural network will become fault-tolerant when each part of a neural network such as neuron, connection weight and others is realized to hardware by an analog or a digital circuit. The set of connection weight values uses the set of values obtained by the fault-tolerant learning as described later. Therefore, the fault tolerance for failure during learning phase is out of range. In Addition, a neural network for solving complex problems is complex in hardware too so high fault tolerance is highly expected.

Various studies on fault tolerance in multilayer neural networks have been conducted. References [2] to [11] are the examples. As a previous study of this research, authors proposed a fault-tolerant algorithm called “Deep LM” which make 3-layered-perceptrons multiple-weights-and-neurons fault tolerant[12]. The algorithm name hereinafter will be called “FTL-algo” so it will not be confused with the deep learning. FTL-algo is based on the BP algorithm with some modifications. With FTL-algo, it has been proved that a 3-layered-perceptrons with 100% fault-tolerant can be realized for multiple weight-and-neuron faults. If the value of the degree of fault tolerance becomes bigger, FTL-algo learning time also becomes enormous bigger than normal BP algorithm. Therefore, authors developed FTL-algo with CUDA C programming and proposed acceleration method using GPGPU[13].

In 2017, Amatya et al. studied fault tolerance for extended deep learning framework(hereinafter referred as “DLFW”) called MaTEx-Caffe that used in a parallel computing environment called Message Passing Interface (MPI)[14]. In 2019, Duddu et al. studied fault tolerant learning for deep neural network by Tikhonov regularization using Pytorch[15]. In addition, Xu et al. studied fault tolerance for deep learning with applied N-version programming using Pytorch[16]. These three studies imply that researches on fault tolerance on DLFWs have begun in recent years.

Researches on implementations of fault-tolerant learning algorithms including FTL-algo using GPGPU-based accelerations(including DLFWs) to find more efficient methods for implementing fault-tolerant in multilayer neural networks is highly desired in discovery. The previous researches [13] and [17] described acceleration of FTL-algo using GPGPU. However, this kind of research has not been found in previous studies except [13] and [17].

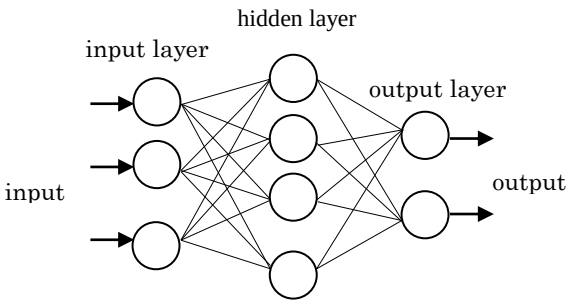

Fig.1 3-layered perceptron (MLP)

1.2. Purpose

The purpose of this research are listed below.

- The first purpose is to select the optimal environment among two DLFWs on two OSs to implement FTL-algo. The two DLFWs are Keras Tensorflow(hereinafter referred as "Keras") and Pytorch, and the two OSs are Linux and Windows. The programming development for FTL-algo is performed in each environment, and the best one is selected from the viewpoint of calculation time and difficulty of programming.
- The second purpose is to find a method that requires less learning time than previous studies of [13] and [17]. For this purpose, we compare the result in these reference with the result obtained from the "first purpose" above.
- The third purpose is to show how to extend the function of Pytorch with implementation of "BP algorithm with some modifications like FTL-algo". The parts of algorithm that is not implemented in the original Pytorch needs to be implemented by the user as its extension. However, explanations how to implement them are not described in the "official manuals" that are provided by Pytorch developers. This paper shows how to achieve this for above purpose in Pytorch. Most Pytorch users have less knowledge and programming skills of Pytorch developers. Therefore, the descriptions in this paper should be useful information for most Pytorch users who want to implement parts of algorithm that are not implemented in the original Pytorch.

2. 3-Layered perceptron

2.1. Overview

A 3-layered perceptron (hereinafter referred as "MLP") consists of three layers. The first layer is the input layer, the last layer is the output layer, and the one between them is called the hidden layer. Values are input to the input layer. Each neuron in the hidden layer and the output layer is fully connected to all the neurons in the previous layer through weights, and signals are input from the input layer and output to the output layer through the hidden layer, as shown in Fig. 1.

X_i is called "inner potential" of the i-th neuron, defined by Eq. (1) as the sum of the input signals,

$$X_i = \sum_{j=0}^{N_{pre}} \omega_{ij} \cdot u_j \tag{1}$$

where ω_{ij} is the value of synaptic weights from the j-th neuron in the preceding layer to the i-th neuron, u_j is the output of the j-th neuron, and N_{pre} is the number of the neurons in the preceding layer connected to the i-th neuron. Therefore, the relationship between the input and the output of each neuron is given by Eq. (2).

$$o_i = f(X_i) \tag{2}$$

Here, f is the activation function called "sigmoid function" shown in Eq. (3). T_{emp} is a constant called the "temperature" that determines the slope of f.

$$f(x : T_{emp}) = \frac{1}{1+\exp\ (\frac{-x}{T_{emp}})} \tag{3}$$

2.2. BP Algorithm

The BP algorithm uses the output square error E_p of the output layer as an evaluation function, modifies the weights, reducing it to achieve the smallest error value. The output layer square error E_p for the input teacher signal is defined by Eq. (4),

$$E_p = \sum_{i \in O} \frac{(t_i^p - o_i^p)^2}{2} \tag{4}$$

where P is the set of input signals, O is the set of output layer neuron, o_i^p is the output value and t_i^p is an ideal teacher value of the i -th neuron in output layer.

The modification of the weight ω is based on the steepest-descendants gradient rule so that the square error E in Eq. (5) may decrease.

$$E = \sum_{p \in P} E_p \tag{5}$$

Let ω_{opt} be the weight that minimizes E_p, and ω_{old} is the current weight value. At this time, the weight must be updated so ω_{old} approaches ω_{opt} in learning. Therefore, the change of $\Delta\omega$ is obtained from the slope of ω_{old} and a new weight is obtained from Eq. (6).

$$\omega_{new} = \omega_{old} + \Delta\omega \tag{6}$$

Weight modification is shown in Eq. (7),

$$\Delta\omega_{ij}^p = -\eta \cdot \frac{\partial E_p}{\partial w_{ij}} \tag{7}$$

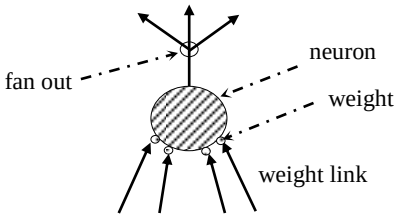

Fig.2　Elements in an MLP

where η is the learning rate.

Weight modification between hidden and output layers is shown in Eq. (8) and (9),

$$\Delta\omega_{ij}^{p} = \eta \cdot \delta_i^p \cdot h_j^p \tag{8}$$

$$\delta_i^p = \left(t_i^p - o_i^p\right) \cdot f'\left(s_i^p\right) \tag{9}$$

where h_j^p is output of the *j*-th neuron in hidden layer, and s_i^p is inner potential of *i*-th neuron in output layer. f' is derivative of sigmoid function.

Also, weight modification between input and hidden layers is shown in Eq. (10) and (11),

$$\Delta\widehat{\omega}_{jk}^{p} = \eta \cdot \sigma_j^p \cdot i_k^p \tag{10}$$

$$\sigma_j^p = \left(\sum \delta_h^p \cdot \omega_{hj}\right) \cdot f'\left(x_j^p\right) \tag{11}$$

wherei_k^p is *k*-th neuron output in input layer, ω_{hj} is weight value between *j*-th neuron in hidden layer and *h*-th neuron in output layer, and $x_j^p$is inner potential of *j*-th neuron in hidden layer.

In this research, modification value obtained from each input signal $p \in P$ is added, and when all input signal are input, the modification is executed.

Then, weight modification is repeated until condition as shown in Eq. (12).

$$\max_{p\in P, i\in O}\left(t_i^p - o_i^p\right)^2 < e_o^2 \tag{12}$$

where e_o is called output error in a learning phase. If a learning satisfies condition in Eq. (12), the learning ends and will be called “successful”. Otherwise, learning is called to be “failed”.

2.3.　Fault Model

As an MLP is composed by neurons and weights, faults can occur in three places such as neuron, weight line, and weight as shown in Fig. 2.

In this paper, there are three assumptions concerning faults in the elements of MLPs, which are shown in Fig.2, are described as follows.

Assumption 1: (the range of faults）

Faults occur only at neurons in hidden and output layers also weights. Disconnection of an interconnecting link is considered a weight or a neuron fault that stuck at 0.

Neurons in the input layer are fault-free because if an MLP is implemented in hardware, they are only terminals that transmit input signals to weights between the input and hidden layers so they are simple circuits and the fault possibility considered extremely low.

Assumption 2:(the value of weight)

The value of weight is assumed to be in the range from –1 to 1, regardless whether there is a fault or not.

Assumption 3: (the output value of a neuron fault）

The value of neuron is assumed to be in the range from 0 to 1, regardless whether there is a fault or not.

2.4.　FTL-algo

The core part of FTL-algo is the same as normal BP algorithm. However, modified or additional parts compared to the normal BP algorithm are listed as follows.

1. The sigmoid function f_o of a neuron in the output layer is defined by Eq. (13) below. The temperature T_{emp} of the sigmoid function f_o is calculated using Eq. (14), where N_{lm} is a parameter of positive integer, that is the degree of fault-tolerance,e_o is the maximum value of output error that can be regarded as “successful” fault-tolerant learning with $e_o = 0.1$

$$f_o(x) = \frac{1}{(1+e^{\frac{-x}{Temp}})} \tag{13}$$

$$T_{emp} = \frac{N_{lm}}{\ln(e_0{}^{-1}-1),} = \frac{N_{lm}}{\ln 9} \tag{14}$$

2. The sigmoid function f_h of a neuron in hidden layer is shown in Eq. (15), which is equivalent with normal sigmoid function.

$$f_h(x) = \frac{1}{(1+e^{-x})} \tag{15}$$

3. If a weight value to be modified is greater (less) than 1(–1), it is set to 1(–1) to make it in the range from 1 to 1 which is called “$W_{|1|}$-process”. With “$W_{|1|}$-process”, absolute value of weight is 1 or less.

The following is proved in Reference [12].

Table 1 DLFW environment

	Windows	Linux
OS	Windows 7 (64 bit)	Ubuntu 16.04 LTS
CPU	AMD Fx™-8320 eight core	AMD A10 5800K
GPU	NVIDIA GeForce GTX 1070	NVIDIA GeForce GTX 1070
GPU RAM	8 GB	8 GB
GPGPU Platform	CUDA 9.0 / cuDNN 7.1	CUDA 9.0 / cuDNN 7.0
Programming Language	Python 3.6	Python 3.6
DLFW	Tensorflowgpu 1.8.0/ Keras 2.1.16 Pytorch 1.0.0	Tensorflowgpu 1.8.0/ Keras 2.1.16 Pytorch1.2.0

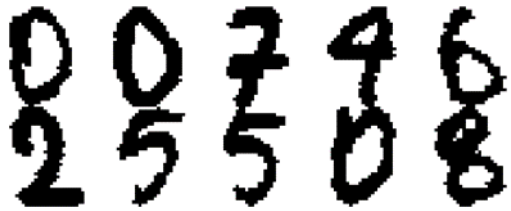

Fig.3 Samples in the Dataset

Let F be any multiple faults whose index set is $(\widehat{N}_F, \cup_{k=1}^{m} \widehat{W}_F^k)$, where $\widehat{N}_F$ is a set of indices of neuron faults in hidden layer, and $\widehat{W}_F^k$ is a set of indices of weight faults which are connected to the *k*-th neuron in the output layer.

The MLP with *m* output neurons achieved by a successful FTL-algo learning is fault-tolerant to *F* if Eq. (16) is satisfied.

$$\left|\widehat{N}_F\right| + 2\left|\widehat{W}_F^k\right| - \left|\widehat{N}_F \cap \widehat{W}_F^k\right| \leq (N_{lm} - 1) \tag{16}$$

3. Results and Programming

3.1. Research Environments

In order to run DLFWs, NVIDIA CUDA and cuDNN (cuda Deep Neural Network) deep learning libraries must be installed. The requirement condition to run cuDNN is that CUDA version is 7.0 or higher, and GPU architecture is Kepler microarchitecture with compute capability 3.0 or higher[18]. On this research, we use NVIDIA GeForce GTX 1070 with compute capability 6.1.

We create two DLFW environments in a Linux PC and in a Windows PC as shown in Table 1. For this purpose, various methods were studied so that the two DLFW environments could be constructed on a single PC. Each DLFW environment is created by using virtualenv[19], a virtual environment tool to create isolated Python environment. With virtualenv, it is possible to build an independent virtual environment for each module, package, and Python version[20].

Table 2 Simulation parameters

training epoch	1000
learning rate η	0.01
Learning stop e_o	0.1
Batch size	32

Table 3 N_{lm} and $Min.N_h$ relation

N_{lm}	3	6	9	12
$Min.N_h$	53	107	167	226

Table 4 FTL-algo learning time and Win/Lin

N_h	Keras			Pytorch		
	Linux(s)	Windows(s)	Win/Lin	Linux(s)	Windows(s)	Win/Lin
53	0.186	0.308	1.66	0.00160	0.00817	5.11
107	0.191	0.305	1.60	0.00157	0.00554	3.53
167	0.187	0.309	1.65	0.00154	0.00558	3.62
226	0.188	0.310	1.65	0.00156	0.00559	3.58

3.2. Dataset

In this research, the Optical Recognition of Handwritten Digits dataset[21] (handwritten digit dataset), that was also used in the previous researches [13] and [17], is used. The samples are shown in Fig. 3. Each pattern consists 1024(32x32) bits where black = 1 and white = 0, and there are 1934 patterns included.

The teacher data is converted to the SECDED code by H_4 matrix as shown in Eq. (17). In here, fault tolerance is up to one neuron in output layer.

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

3.3. Computation Result

3.3.1. Basic Parameters

Table 2 shows parameters used in our simulations. Furthermore, Table 3 shows the minimum numbers of neurons in hidden layer called $Min.N_h$ for the succeeded fault tolerance learnings when fault tolerance degree N_{lm} values are 3, 6, 9 and 12, respectively. These values were discovered by the previous study[13].

In our simulations detailed below, the number of neurons in hidden layer N_h is set to the value of $Min.N_h$ correspond to the case of N_{lm} as shown in Table 3.

Table 5　Learning time per epoch for normal BP algorithm

N_h	Keras		Pytorch	
	Linux(s)	Windows(s)	Linux(s)	Windows(s)
53	0.0967	0.267	0.00141	0.00522
107	0.0976	0.297	0.00145	0.00575
167	0.0967	0.299	0.00140	0.00518
226	0.104	0.301	0.00144	0.00563

Table 6　Values of ((FTL-algo time)–(normal BP time)) / (FTL-algo time)・100 (%)

N_h	Keras		Pytorch	
	Linux (%)	Windows (%)	Linux (%)	Windows (%)
53	48.0	13.3	11.9	36.1
107	48.9	2.62	7.64	3.79
167	48.2	3.24	9.09	7.17
226	44.7	2.90	7.69	0.716

Table 7　Average learning time per epoch (Linux)

N_h	CUDA[13](s)	Chainer[17](s)	Keras(s)	Pytorch(s)
53	0.00153	0.0129	0.186	0.00160
107	0.00246	0.0129	0.191	0.00157
167	0.00371	0.0132	0.187	0.00154
226	0.00493	0.0130	0.188	0.00156

3.3.2.　Comparison among two DLFWs on two OSs

Table 4 shows the average learning time per epoch of FTL-algo in each DLFW on Linux and Windows. “Win/Lin” in the table means “(learning time on Windows)/(learning time on Linux)”

Table 4 shows that learning time of Keras and Pytorch on Windows are respectively about 1.6 times and about 3 to 5 times as much as Linux.

The first possible cause is the difference in the GUI display on each OS. In other words, GUI display on Windows cannot be stopped due to its OS specification, so GPU memories are consumed by the display. On the other hand, GUI display is stopped on Linux.

Another cause concerning Keras is that NVIDIA driver on Windows has some latency time in CUDA kernel when its API is called to run, so it took some time and does not dispatch immediately[22].

3.3.3.　Comparison between normal BP algorithm and FTL-algo

Table 5 shows the average learning time per epoch for normal BP algorithm.

The values of ((FTL-algo time)–(normal BP time)) / (FTL-algo time)・100 (%) calculated from Table 4 and 5 are shown in Table 6.

Table 8　Programming environment in [17]

OS	Ubuntu 16.04 LTS
CPU	AMD A10 5800K
GPU	NVIDIA GeForce GTX 1070
GPU RAM	8 GB
GPGPU Platform	CUDA 8.0.61 / CuPy 5.1.0
Programming Language	Python 3.5.2
DLFW	Chainer 5.1.0

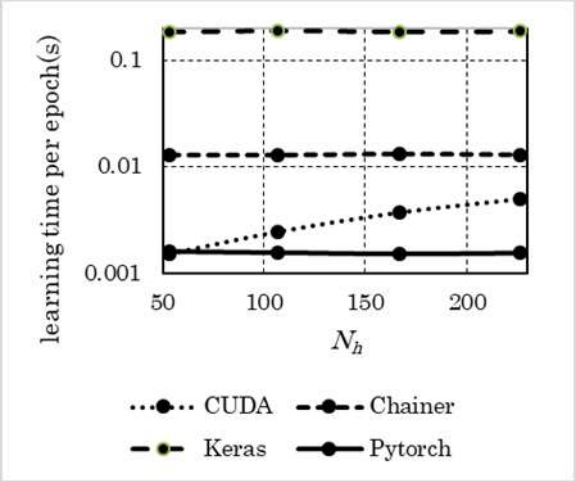

Fig.4　Relation between average learning time per epoch and N_h

Table 9　Data given by dividing each method value in Table 7 by each Pytorch value

N_h	CUDA[13]	Chainer[17]	Keras	Pytorch
53	0.684	5.75	45.5	1.00
107	1.65	8.63	62.7	1.00
167	2.58	9.16	64.9	1.00
226	3.29	8.65	63.5	1.00

Table 6 shows the following.

- The increase of the computation time of FTL-algo compared to normal BP algorithm is less than 49% per epoch.
- Especially when N_h is 107 or more, the computation time on Windows is less than 8 percent. Therefore, the difference between these two algorithms is not the main reason of computation time difference between the two OSs.

These results show that Linux environments are better than ones on Windows in term of learning time, so only execution results on Linux are shown below.

3.3.4.　Comparison Result with Previous Researches

The data of average learning time per epoch on our Linux

environments and in previous studies [13] and [17] are shown in Table 7. In [17], the methods of both previous studies are executed on the environment shown in Table 8, and their data are quoted from [17]. Fig.4 shows relation between average learning time per epoch and N_h by data in Table 7. Table 9 shows data given by dividing each method value in Table 9 by each Pytorch value.

Tables 7 and 9 and Fig.4 show the following.

1. Among these four methods, the learning time per epoch of Pytorch is the smallest when N_h is 107 or more.
2. The learning time of CUDA[13] is increasing along with the increase of N_h.
3. On the other hand, the learning time of each DLFW is almost a constant regardless of N_h.
4. When N_h is 107 or more, the learning time of CUDA[13] is longer than Pytorch.
5. The learning time of Keras is the largest, about 65 times as much as Pytorch.

The main reason for the long learning time in Keras (5-th item mentioned above) is that only Pytorch has special features which is called “CUDA SEMANTIC”[23], which is explained as follows.

- For Example, concerning torch.cuda.synchronize() feature, when data is transferred from a CPU to a GPU by using this feature, Pytorch automatically performs necessary synchronization, so data transfer and calculation for a learning can be performed simultaneously. Also, with pin_memory() feature, the speed of data transfer from a CPU to a GPU is increased.
- Furthermore, with torch.cuda.empty_cache() feature, unused cache memory is released from Pytorch to achieve higher speed.

In addition, the main reason for 4-th item mentioned above is that programming code in CUDA[13] is developed according to GPU and CUDA technologies in 2013, although version–up speed of Pytorch almost catches up with the progress speed of these recent technologies, including CUDA SEMANTIC as mentioned above. In other words, if an optimized CUDA C FTL-algo program which uses recent technologies sufficiently could be developed, it might be faster than a FTL-algo program in Pytorch.

3.4. Programming Development

3.4.1. Overview

In the following, programming development process of each part about difference between FTL-algo and BP algorithm on Pytorch environment is described. Detail of the same process on Keras environment is omitted because Keras learning time is longer than Pytorch.

Most of the contents described in this section are not explained in the official manuals of Pytorch. Therefore, this section should be useful information for most Pytorch users who want to implement parts of algorithm that are not implemented in the original Pytorch.

```
# CUSTOM ACTIVATION FUNCTION (new add)
#'''
def tempsigmoid(x):
    nd=3.0
    temp=nd/torch.log(torch.tensor(9.0))
    return torch.sigmoid(x/(temp))
#'''
```

Fig.5 Description of tempsigmoid function

```
# CREATE MODEL
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(1024, 226),
            nn.Sigmoid(),
            nn.Linear(226, 8),
            TempSigmoid()
            )

    def forward(self, x):
        x = self.layers(x)
        return x
```

Fig.6 Description of applied tempsigmoid function

```
class WeightClipper(object):

    def __init__(self, frequency=5):
        self.frequency = frequency

    def __call__(self, module):
        # filter the variables to get the ones you want
        if hasattr(module, 'weight'):
            w = module.weight.data
            w = w.clamp(-1,1)
            module.weight.data = w
```

Fig.7 Description of WeightClipper class

3.4.2. Sigmoid function with temp parameter （tempsigmoid）

In Pytorch, activation function is defined in activation.py file which is located inside nn.modules. The difference with Keras is that each optimizer is defined in each separate file.

Pytorch also has an automatic differentiation feature like Keras that is called “autograd”.

We need to create the new activation function called tempsigmoid. Here, as shown in Fig.5, Pytorch tempsigmoid activation function is inherited from sigmoid function as same as Keras.

About this part, the official manual has no explanation to modify and describe this function. Therefore, authors consulted with developers via the web community and tried various methods until success.

```
model = MLP()
model.to(device)
clipper = WeightClipper() #new add weight clipper
model.apply(clipper) # new add weight clipper
```

```
if epochs % clipper.frequency == 0:
        model.apply(clipper)
```

Fig.8　Description of applied WeightClipper

```
# STOP TRAINING AT VALUE
class StopatLossValue:
    def on_epoch_end(self, loss, **kwargs):
        # if the monitored metrics got worst
        # set a flag to stop training
        eo = 0.1
        if loss < eo.pow(2):
            return{'stop_training': True}
```

Fig.9　Description of StopatLossValue

```
outputs = model(x)
loss = loss_fn(outputs,y)
loss.backward()
optimizer.step()
StopatLossValue()
```

Fig.10　Description of applied StopatLossValue

In Pytorch, the computation process in tempsigmoid function has to be “torch tensor” type. The partial differentiation is automatically done by inheriting sigmoid function with “return” code. Fig.6 shows how to call tempsigmoid in Pytorch.

3.4.3.　Weight absolute constraint ($W_{|1|}$-process)

Unlike Keras, Pytorch have no constraint feature. Because of this, we need to create new object class for it in Pytorch. As shown in Fig.7, this object class is called “WeightClipper”.

As shown in Fig.8, the WeightClipper class is defined as the “clipper”s parameter, and is executed by “apply” code written in training model.

3.4.4.　Stop condition process

Unlike Keras, Pytorch has no function to stop learning corresponding to the condition of Eq.(12). Therefore, it is necessary to define new class for it.

About this part, there is no explanation in the official manual. Therefore, authors consulted with developers via web community, and tried various methods until success in developing this part.

Fig.9 shows the “StopatLossValue” class defined in Pytorch. Here, check is performed at the end of epoch, and if the condition is satisfied, training will be forcedly terminated. Furthermore, since StopatLossValue class is a usual object class, it is called as shown in Fig.10.

```
x_train = torch.from_numpy(x).float()
y_train = torch.from_numpy(y).float()
```

Fig.11　Description of torch.from.numpy().float()

```
dataset = torch.utils.data.TensorDataset(x_train, y_train)
loader = torch.utils.data.DataLoader(
    dataset=dataset,
    batch_size = 32,
    shuffle=False,
    num_workers=0, #data worker scheduling
    pin_memory=torch.cuda.is_available()
```

Fig.12　Description of torch.utils.data.TensorDataset and torch.utils.data.DataLoader

Table 10　Coding complexity on Keras and Pytorch

	Number of lines of code	Number of modified files	Programming language
Keras	109	4	Python
Pytorch	141	1	Python

3.4.5.　Dataset preparation

On this research, it is necessary to convert the dataset from decimal to 32x32 bit array binary. In addition, the dataset that will be used in DLFWs need to be convert to tensor data type which has multi-dimensional arrays.

In Keras, this convert is made automatically by numpy library. On the other hand, since Pytorch uses torch.tensor(), the descriptions of the procedure for preprocessing to read training data is needed. Because there is no explanation in the official manual for it, authors developed this part with trial and error, consulting with the developers via web community.

The discoveries are shown below.

- Conversion from numpy type to tensor type, “torch.from.numpy().float()” is required as shown in Fig.11.
- Furthermore, each description of “torch.utils.data.TensorDataset” and “torch.uitls.data.DataLoader” is also required as shown in Fig.12. Particularly, in order to find appropriate values for each parameter of “torch.utils.data.DataLoader”, authors through trial and error, checked the data size of output data variables, and checked data values correlation in detail. These reasons are listed below.
  - Depending on the value of each parameter given to torch.utils.data.DataLoader, the size of output data

variables in each process during training is changed. If the sizes are not set correctly, an error occurs and the program cannot be executed.

- According these parameter values, output data values in each process during training also change.
- Each process during training is like a black-box.

3.4.6. Programming code comparison

Table 10 shows data about programming codes on Pytorch and Keras.

Table 10 shows the following.

- Programming in Keras has fewer lines of codes compared to Pytorch. The reason is that Keras has fewer lines to create model training. Furthermore, in Keras the code to execute a training can be described with one line of the "model.fit()" function. On the other hand, creation process for training model in Pytorch is more complicated.
- Regarding the development of FTL-algo, there are more modified files in Keras than Pytorch. Because in Keras the files are divided for each function, so that modifying for each file is necessary.

4. Conclusions

In this research, we show how to implement and execute FTL-algo using DLFWs of Keras and Pytorch. As a result, we have discovered the following.

- Linux environments can run FTL-algo faster than Windows environments.
- Pytorch can execute FTL-algo faster than Keras.
- The number of lines of program code in Keras is fewer than Pytorch. However, the number of files which should be modified in Keras is more than Pytorch. In other words, the programming difficulty between both DLFWs is almost the same.

Based on the above, related to "the first purpose" mentioned in Section "1.2 Purpose", regarding the implementation of FTL-algo with two DLFWs (Keras and Pytorch) on two OSs (Linux and Windows), we found that the most optimal environment is Pytorch on Linux environment.

Next, comparing the results mentioned above with the methods in [13] and [17], we discovered that Pytorch environment on Linux can execute FTL-algo faster than these two previous studies. In other words, related to "the second purpose" mentioned in Section 1.2, we discovered a method that requires less learning time than both prior researches.

In addition, related to "the third purpose" mentioned in Section 1.2, how to develop and execute "BP algorithms with some modifications like FTL-algo" on Pytorch are shown. Most of these explanations are not included in the official manual provided by Pytorch developer.

As our future works, we will contact each DLFW developer to report this research result which is "new function" that can be implemented in each DLFW so that it could be considered to be implemented in the new version upgrade.

Keywords: Deep Learning Framework, Keras, Pytorch, Fault tolerant BP learning, Multi-layered Perceptron, GPGPU

Acknowledgements

We sincerely thank the developers of each DLFW for their support.

We also would like to express our sincere appreciation to K. Takada for writing and others.

References

[1] D.E. Rumelhart, G.E. Hinton, and R.J. Williams: "Learning Representations by Back-Propagating Errors", Nature, vol.323, pp.523-535, (1986).

[2] D.S. Phatak and I. Koren,: "Compete and Partial Fault Tolerance of Feedforward Neural Nets," IEEE Transactions on Neural Networks, vol.6, No. 2, pp.446-456, (1995).

[3] J. Nijhuis, B. Hoefflinger, A.v. Schaik, and L. Spaanenburg: "Limits to the Fault-Tolerance of a Feedforward Neural Network with Learning", FTCS, pp.228-235, (1990).

[4] I. Takanami and Y. Oyama: "A Novel Learning Algorithm which Makes Multilayer Neural Networks Multiple-Weight-Fault-Tolerant", IEICE Transactions on Information and Systems, vol.E86-D, no.12, pp.2536-2543, (2003).

[5] R.P. Lippmann: "An Introduction to Computing with Neural Nets", IEEE ASSP Magazine vol.4, no.2, pp.4-22, (1987).

[6] C. Torres-Huitzil and B. Girau: "Fault and Error Tolerance in Neural Networks: a Review", IEEE Access, vol.5, pp.17322-17341, (2017).

[7] S. Srinivasan and C.F. Stevens: "Robustness and Fault Tolerance Make Brains Harder to Study", BMC Biology, vol.9, pp.46-48, (2011)

[8] T. Sejnowksi and T. Delbruck: "The Language of the Brain", Scientific American, vol.307, pp. 54-59, (2012).

[9] W. Maass: "To Spike or not to Spike: That is the Question", Proceedings of the IEEE, vol.103, no.12, pp.2219-2224, (2015).

[10] P. Chandra and Y. Singh: "Fault Tolerance of Feedforward Artificial Neural Networks - a Framework of Study", Proceedings of the International Joint Conference of Neural Networks, vol.1, pp.489-494, (2003).

[11] S.E. Fahlman: "Neural Nets Learning Algorithms and Benchmarks Database", maintained by S.E. Fahlman at the Computer Science Department., Carnegie Mellon University.

[12] T. Horita, I. Takanami, and M. Mori: “Learning Algorithms which Make Multilayer Neural Networks Multiple-Weight-and-Neuron-Fault Tolerant”, IEICE Transactions on Information and Systems, vol.E91-D, no.4, pp.1168-1175, (2008).

[13] T. Horita, I. Takanami, M. Akiba, N. Terauchi, T. Kanno: “A GPGPU-based Acceleration of Fault-Tolerant MLP Learnings”, Proceedings of IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, pp.245-252, (2014).

[14] V. Amatya, A. Vishnu, C. Siegel, and J. Daily: “What does Fault Tolerant Deep Learning from MPI”, Proceedings of the 24th European MPI User’s Group Meeting Article, no.13, pp.1-11, (2017).

[15] V. Duddu, D.V. Rao, and V.E. Balas: “Adversarial Fault Tolerant Training for Deep Neural Networks”, ArXiv, abs/1907.03103, (2019).

[16] H. Xu, Z. Chen, W. Wu, Z. Jin, S. Kuo and M. Lyu: “NV-DNN: Towards Fault-Tolerant DNN Systems with N-Version Programming", Proceedingsof49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), pp. 44-47, (2019).

[17] K. Takahashi: “A Research on a Deep Learning Framework (in Japanese)”, graduation thesis in Polytechnic University of Japan, (2019).

[18] “NVIDIA® GPU Deep Learning（深層学習）開発環境構築情報” in Arcbrain Inc. home page, https://arcbrain.jp/support/NVIDIA/Deep_Learning/, (2020)

[19] I. Bicking: “virtualenv”, https://virtualenv.pypa.io/en/latest/, (2018).

[20] “仮想環境” in Python Japan home page, https://www.python.jp/install/windows/virtualenv.html, (2020).

[21] D. Dua, and C. Graff: “UCI Machine Learning Repository”, Irvine, CA: University of California, School of Information and Computer Science, http://archive.ics.uci.edu/ml, (2019).

[22] NVIDIA inc.: ”Use a Suitable Driver Model”, NVIDIA CUDA Installation Guide For Microsoft Windows, DU-05349-001_v10.2,pp.8, (2019)

[23] Torch Contributors: “CUDA Semantics”, Pytorch, https://pytorch.org/docs/stable/notes/cuda.html, (2019)

（原稿受付 2020/05/14，受理 2020/07/03）

*Astriwindusari，学士（工学）
Surakarta Vocational Training Centre, Jl. Bhayangkara No.38, Panularan, Laweyan, Kota Surakarta, Jawa Tengah 57149, Indonesia
Email: astriwindusari@hotmail.com

*堀田 忠義，博士（工学）
職業能力開発総合大学校，能力開発院，〒187-0035　東京都小平市小川西町 2-32-1
Tadayoshi Horita, Faculty of Human Resources Development, Polytechnic University of Japan, 2-32-1 Ogawa-Nishi-Machi, Kodaira, Tokyo 187-0035.
Email: horita@uitec.ac.jp

*秋葉 将和，博士（工学）
職業能力開発総合大学校，能力開発院，〒187-0035　東京都小平市小川西町 2-32-1
Masakazu Akiba, Faculty of Human Resources Development, Polytechnic University of Japan, 2-32-1 Ogawa-Nishi-Machi, Kodaira, Tokyo 187-0035.
Email: akiba@uitec.ac.jp