Computers and Electrical Engineering 82 (2020) 106558

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Adaptive cache pre-forwarding policy for distributed deep learning ☆

Sheng-Tzong Cheng [a], Chih-Wei Hsu [a], Gwo-Jiun Horng [b,*], Che-Hsuan Lin [a]

[a] Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan
[b] Department of Computer Science and Information Engineering, Southern Taiwan University of Science and Technology, Tainan, Taiwan

ARTICLE INFO

Article history:
Received 3 June 2019
Revised 18 January 2020
Accepted 20 January 2020
Available online 1 February 2020

Keywords:
Deep learning
Distributed computing
Cache, Reinforcement learning

ABSTRACT

With the rapid growth of deep learning algorithms, several high-accuracy models have been developed and applied to many real-world domains. Deep learning is parallel and suitable for distributed computing, which can significantly improve the system throughput. However, there is a bottleneck for cross-machine training, that is, network latency. Nodes frequently need to wait for synchronization, and the content of each synchronization may range from several megabytes to hundred megabytes. Thus, network communication takes considerable time in the training process, which reduces system performance. Therefore, many computing architectures have been proposed. This paper proposes a type of distributed computing system for deep learning. Our design aims to reduce synchronization times and network blocking times by using a new cache mechanism, called cache pre-forwarding. The design concept of cache pre-forwarding aims to exploit reinforcement learning to train a pre-forwarding policy to increase the cache hit rate. Because of the features of reinforcement learning, our policy is adaptive and applicable to different computing environments. Finally, we experimentally demonstrate that our system is feasible.

1. Introduction

Following cloud computing, deep learning, which is a branch of neural networking, has become the most popular research field in recent years [11,22,24,25]. Similar to a human brain, a neural network model contains numerous neurons that connect to others through synapses. Hundreds of neurons make up a layer, and layers compose a network. When a model includes dozens or hundreds of layers, it becomes deep, and the network is called a “deep neural network.” In the 1950s, scientists designed a neural network, but it did not prevail due to limited computing resources. Training a simple model takes days even when there is only one hidden layer. In 2006, Hinton and LeCun proposed a new neural network, called deep belief net [4], by initializing parameters based on unsupervised learning rather than random assignment, with which the neural network returned to the mainstream. In addition to the study by Hinton and his team, Moore's law promotes the growth of neural networks, and researchers have developed deeper and more complicated networks with modern

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Associate Editor Dr. Luiz Bittencourt..
* Corresponding author.
E-mail addresses: awei.hsu@seed.net.tw (C.-W. Hsu), grojium@gmail.com (G.-J. Horng).

https://doi.org/10.1016/j.compeleceng.2020.106558
0045-7906/

CPUs [8] as well as GPUs [18]. Some applications have also used FPGAs to accelerate the forwarding and backpropagation. However, compared with CPU and GPU, FPGAs exhibit low flexibility and face difficulties in quickly developing a project.

Using GPU acceleration, data scientists in machine learning can reduce the experiment times from days to hours. In addition to GPU acceleration, scaling up machine learning using a parameter server [10,12,13,15,16,19,21] can significantly improve data throughput, which is one of the most critical factors for evaluating system performance in the big data era. Because the entire training process is parallel, the programmer can easily parallelize the program by multithreading or multiprocessing. However, when the program is written to be parallelized or distributed, it incurs a synchronization problem. For instance, a distributed deep learning system works like the MapReduce [5] program, in which the system maps tasks to multiple workers to perform feed-forwarding or backpropagation and reduces the results from workers to obtain the final answer. Here, the map step is used for parallelization, whereas the reduce step is used for synchronization. A training flow comprises several map and reduce steps with considerable lock acquisitions and releases to prevent the race condition. This is the reason that Hadoop [7], a well-known distributed big data platform implemented with MapReduce, is not suitable for machine learning scaling because it always saves the result to the local disk or the remote database, and these data I/Os are time consuming. Because distributed deep learning requires several synchronizations to ensure that each worker has the same cognition of the model, all syncs are handled by a central server, called a parameter server. After each training iteration, every worker needs to send gradients or weights back to the server. In a traditional parameter server design, the server averages the gradients or weights to update the model in a local database and resends the latest model to the worker. All these executions are protected by a global lock. Workers need to compete for the global lock if they attempt to upload data to the server, which may cause several lock contentions. The communications between workers and the server are commonly connected through the ethernet, which causes high network latency with limited bandwidth, in contrast to local machine IPCs. Because the content of each sync is not small, the cumulative synchronization time will constrain the convergence speed; in other words, researchers spend more time on training a model distributedly compared with training on a local machine.

This paper proposes a distributed deep learning system called “model-cache pre-forwarding parameter server system” (MPPS). Each worker in the system has its own GPU for matrix computation acceleration. The MPPS properly handles the synchronization issue and reduces the communication time by using the cache mechanism with an accurate prediction policy for cache pre-forwarding. It also reduces the critical section size between lock acquisition and lock release. We experimentally confirm that our approach exhibits better throughput performance than Downpour SGD [10].

The remainder of this paper is organized as follows. The background and related work are described in Section II. The proposed system design, including the system architecture, model training flow, and cache pre-forwarding policy, is presented in Section III. Section IV presents the experimental results, which demonstrate the advantages of the system. Section V reviews the proposed system, MPPS, and proposes the direction of future work for further research.

2. Background and related work

2.1. Distributed deep learning parallelism

Gradient descent is by far one of the most popular algorithms for performing optimization and the most common way to optimize neural networks. It can be expressed as follows:

$$W_{i+1} = W_i - \frac{\alpha}{m}\sum_{j=1}^{m}\frac{\partial L^j}{\partial W_i} \tag{1}$$

where W is the model parameter, L is the loss function with respect to the parameter and training data j, m is the minibatch size, α is the learning rate, and i is the iteration number. According to different algorithms, we assign different m values to the formula. Parallelism is a commonly used approach to increase the computational efficiency of gradient descent. The two popular distributed parallel algorithms are data and model parallelism. The main difference between them is that data parallelism divides the data for parallelizing, whereas model parallelism divides the model. Literally, data parallelism means that each worker in the cluster has the same model but performs computations using different data, whereas model parallelism means that each worker has a part of the model and holds the same data. In a neural network application, a worker in data parallelism feeds different minibatches to a general neural network model to obtain the results. By contrast, a worker in model parallelism is fed with the same training data but holds only some of the layers in the model; for example, worker A preserves the parameters in the first three layers, whereas worker B does so in the remaining layers. Each method has advantages and disadvantages, which vary from architecture to architecture. However, data parallelism is used in most cases, and model parallelism is only suitable for extremely large-scale models, which are unable to fit into GPU memory.

It is simple to parallelize the gradient descent in data parallelism. As shown in (1), the model is updated only when m gradients have been computed. We map m compute-gradient tasks to n workers. Therefore, each worker computes x gradients in an iteration, where x is equal to m divided by n. Then, one worker reduces the gradients to a gradient result and applies it to the model.

Fig. 1 illustrates data parallelism by using a gradient descent to optimize a neural network. A cluster comprises three computing workers and one parameter server. The server is responsible for computing a new model based on the gradients received from the workers. The workers are responsible for computing the gradient based on batch data. A study

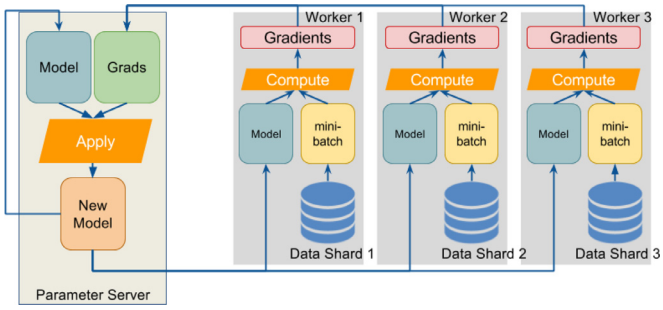

Fig. 1. Data parallelism training scheme.

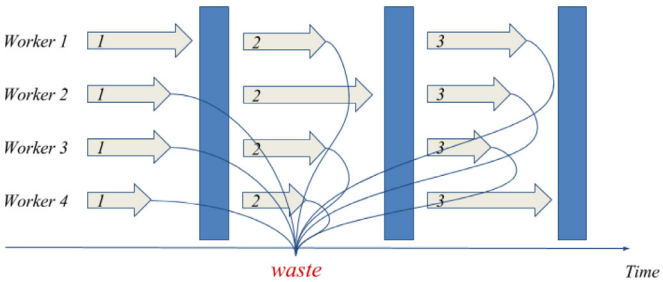

Fig. 2. Bulk synchronous parallel.

[10] proposed a distributed deep learning [26–30] implementation called Downpour SGD, which is different from the aforementioned task assignment approach. Workers in Downpour SGD compute all m gradients and upload them to the server.

There are two synchronization approaches for data parallelism: bulk synchronous parallel (BSP) and asynchronous parallel (Async).

2.1.1. BSP

BSP is a bridging model for designing parallel algorithms developed by Leslie Valiant of Harvard University during the 1980s [1] and is shown in Fig. 2. A BSP computing model consists of many iterations, and each iteration comprises the following four components:

- Concurrent computation: each participating worker performs local computations, namely, computing gradients depending on the local training data and the local model.
- Communication: the workers transmit the gradients or model to the parameter server and obtain the latest model from the server.
- Barrier synchronization: when a worker finishes one iteration, it waits until all other workers have finished the iteration.
- Server-side computation: the parameter server computes a new model based on the data received from the workers.
- Therefore, the cost of BSP is denoted as follows:

$$\sum_{t=1}^{T}\left(max_{i=1}^{n}\left(c_i+m_i+L_i+S\right)\right)_t, L>0 \tag{2}$$

where n is the number of workers, c_i is the cost of local computation for worker i, m_i is the transmission time for sending the gradients and receiving the model by worker i, L_i is the cost of a barrier synchronization of worker i, S is the cost of the server-side computation, and T is the predefined iteration times.

2.1.2. Async

Async is a BSP variant whose concept is shown in Fig. 3. Downpour SGD is a branch of Async. Workers in BSP have to perform the barrier synchronization step in an iteration, whereas those in Async are not required to wait for the other workers. All model replicas run independently of each other. Due to the algorithm design, Async works more effectively than BSP. We denote the cost of Async as follows:

$$max_{i=1}^{n}\left(\sum_{t=1}^{T}\left(c_i+m_i+S\right)_t\right) \tag{3}$$

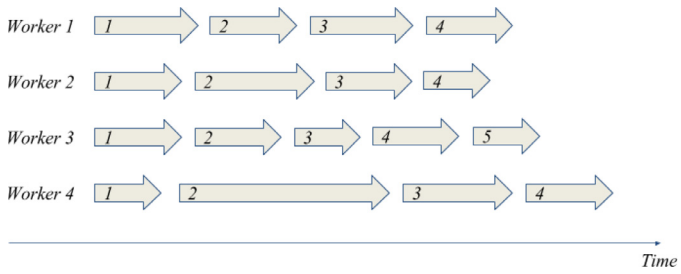

Fig. 3. Asynchronous parallel.

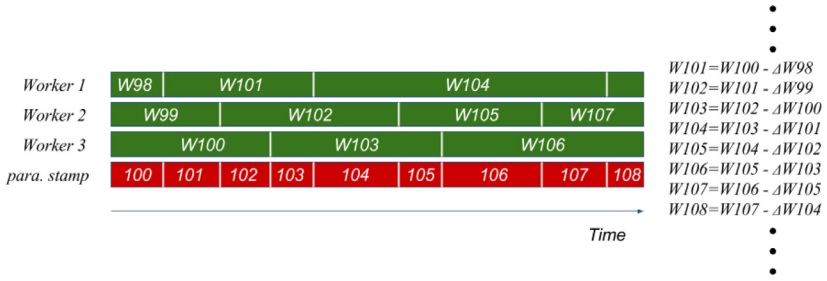

Fig. 4. The staleness problem in Async.

As shown in (3), Async does not have barrier synchronization, and its total cost will be determined by the last worker to finish the iterations. Async overcomes the drawback of barrier synchronization. However, using this asynchronous behavior adds the "staleness problem" to the system. Some of the workers compute the gradients using a model parameter that may be several gradient steps behind the latest model in the parameter server. In other words, the staleness problem refers to applying gradients learned from the stale model to the latest model and is shown in Fig. 4. In a computing cluster, workers may have different computing resources, which cause some workers to execute the task rapidly, while some execute it slowly. A faster worker will update the model in the server frequently and make a slower worker hold a relatively stale model before synchronization. When a slower worker uploads gradients, the server will apply these gradients to the model even if these gradients are not learned from the latest model. A study [23] indicated that staleness can potentially slow the convergence.

To solve the staleness problem, Ho proposed a new server system for distributed deep learning [12] called stale synchronous parallel (SSP). SSP reduces the time the workers spend on performing network I/O by reducing the sync times while still guaranteeing convergence. SSP also controls the staleness problem within a certain range by delaying the fastest worker to wait for the slower worker. Additionally, Zhang [16] proposed a variant of the Async algorithm in which the learning rate is tuned according to staleness and provided a theoretical guarantee of convergence.

2.2. Reinforcement learning

Reinforcement learning is a type of machine learning as well as a branch of artificial intelligence [2,3]. It allows machines and software agents to automatically learn a concept in a specific environment to maximize the performance. An agent receives feedbacks (i.e., rewards), which are required to learn optimized actions within a context. The learning process is a sequence of discrete time steps, where each step has different environmental states and actions.

Fig. 5 illustrates the interaction between an agent and the environment in one step. There are three key elements in the figure: the agent, the environment, and the interpreter. The action maker and the learner are called agents, and the agent interacts with the environment. The third element is an interpreter, which is responsible for parsing environmental signals. At each step t, the agent receives the state s_t and scalar reward r_t, and it executes an action a_t. The environment receives the action a_t and emits the environmental information to the interpreter. The interpreter parses the information and emits state s_{t+1} and then emits the scalar reward $r_{t+1}$to the agent. The process iterates the steps until the end condition is reached.

In reinforcement learning, the agent learns a policy using learning algorithms. For instance, Q-learning is a famous reinforcement learning algorithm that can be used to find an optimal action-selection policy for any given Markov decision process. It is a table-based algorithm that uses a table to store critical data, and it is only suitable for solving discrete problems and perform discrete actions. However, in real-world applications, almost all states and actions are continuous. Although we can discretize the state and action, it raises a question about how to set up the fineness. Therefore, we focus on another mainstream algorithm of reinforcement learning, the policy network.

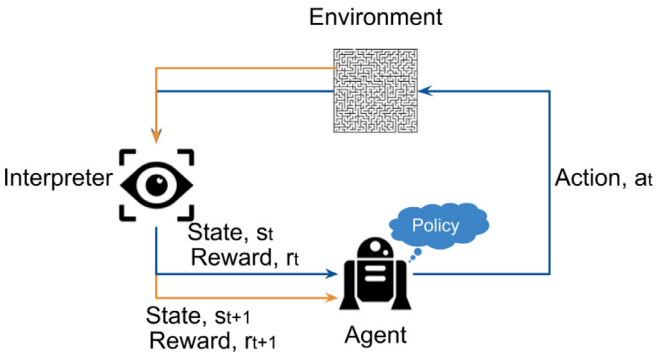

Fig. 5. Agent–environment interaction.

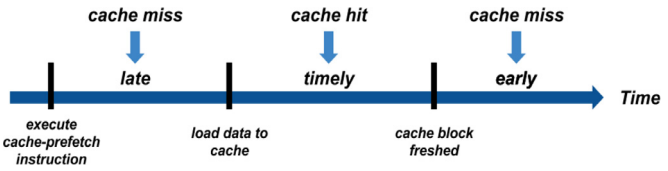

Fig. 6. Cache prefetching timeline.

A policy network is a neural network that uses numerous parameters to store a concept or idea as a policy. It is suitable for solving a problem with continuous states and actions. We encode the state as a multidimensional matrix and express the action as a one-dimensional matrix. The training data are encoded states, and the training labels are the formalized action and reward. The policy network is trained as a normal neural network and iteratively learns how to optimize the agent's performance during the agent–environment interaction process.

In AlphaGo [22], the policy network is successfully used to learn how to play GO chess. The network in AlphaGo is a convolutional neural network. Depending on different applications and goals, researchers can select an appropriate neural network as the core network.

2.3. Cache and cache prefetching

The cache is a hardware or software component used to store data and is designed to serve other system components quickly. For instance, in the computer memory hierarchy, the CPU cache has a lower latency for CPU access than the main memory. Therefore, the CPU requests data from the cache first, and if the cache does not have the data, which is called a cache miss, it then requests the data from the main memory; otherwise, if the data can be found in cache, the CPU accesses them directly. Because the CPU can access data from a quicker device, the overall efficiency is increased. In addition to the CPU cache, which is a hardware component, an HTTP cache is a type of software cache. In the internet world, we use a web browser to request a web page from the web server. The web server sends the html, image, or XML back to the web browser, through which we can view the content. However, fetching something over the network is both slow and expensive. The web client has to wait for the server response due to network latency and pay the internet cost. Thus, the ability to cache and reuse previously fetched resources is a critical aspect of optimizing the performance. The web browser can save the web content locally. When a user tries to access the same content the next time, the web browser directly reads data from the disk and presents it to the user.

Cache prefetching is a technique used by the computer CPU to boost execution performance by fetching execution instructions or data from the original storage in slower memory to a faster local memory, the CPU cache, before the instructions or data are actually needed. Hence, the action is called prefetching. Nonetheless, cache prefetching does not always improve system performance; in contrast, inefficient cache prefetching degrades the performance due to the wastage of CPU resources.

Fig. 6 shows the cache prefetching timeline. There are three specific time points: the time that the processor executes the cache prefetching instruction, the time that the data are loaded into the cache, and the time that the cache block is refreshed. If data are needed between the first and second time points, a cache miss is incurred because the data have not yet been loaded into the cache. The second condition is that data are needed when they have already been loaded into the cache; hence, the processor can access data from the cache directly. The third condition is that the data have been removed from the cache before the processor requests them; this also results in a cache miss. The first and third conditions do not

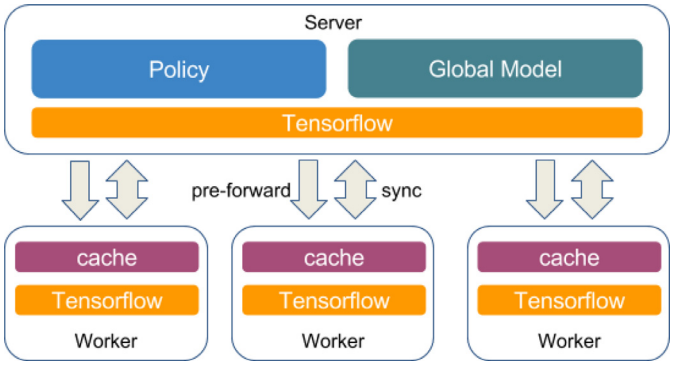

Fig. 7. System architecture overview.

speed up data access; instead, they waste CPU resources to execute cache prefetching. Therefore, it must accurately load the correct data into the cache at the right time. Lee et al. analyzed the conditions when prefetching works, when it does not, and the associated reasons [9]. In addition, a cache prefetching policy has been proposed to enhance the prefetching efficiency [6,14].

3. System design

Two main problems need to be overcome in a data parallelism system, which we aim to solve to improve the performance of a distributed computing system. The first problem is how to build an efficient parameter server. A parameter server controls model training in the Downpour SGD architecture. However, without optimization, the workload in the critical section (i.e., handling updates from workers) is too heavy, which causes several lock contentions, slowing the process. When a worker wants to upload gradients, it usually requires a global lock kept by another worker and keeps waiting until the lock is freed. In this study, we analyze this situation and identify what tasks can be removed from the critical section to reduce the workload in the server. The analysis and solution will be discussed in this section. The second problem, how to reduce the considerable network communication overhead, can be solved in many ways, such as increasing the network bandwidth between the nodes, RDMA, reducing the data size, and reducing the transmission times. Some are hardware solutions, and some are software design improvements. In this study, we focus on reducing the transmission times. In a general computer system, a cache is used to reduce the average cost (time or energy) of accessing data from a slow storage. We want to apply this design to our system to reduce the data movements between the worker and the parameter server. To achieve significant improvement, we need to carefully design the caching mechanism, including what data can be cached, the cache size, and the replacement policy. The detailed caching approach is discussed below.

3.1. MPPS system

The MPPS is based on data parallelism and asynchronous parallel and is a variant of Downpour SGD. Our system mainly aims to improve the throughput to obtain better performance in training a deep neural network model. Before describing the system design in more detail, we first present a schematic of the system architecture in Fig. 7.

In Fig. 7, the server holds two models, and the workers hold a cache, which is not a CPU cache but rather a memory space, to reserve the model from the server. Both the server and workers use Tensorflow [20] to train the neural networks; nevertheless, these networks are different. The server trains a policy network as a cache pre-forwarding policy used only for the training process, and the workers train a general neural network model that the researcher wants to train for specific applications. The networking communications between the server and workers include not only model synchronizations but also the cache pre-forwarding conducted by the server.

Fig. 8 illustrates the training of a model with one server and three workers. The rectangles represent a limited memory data structure to store data, the parallelograms represent the Tensorflow operations, and the cylinders indicate the data stored in the disk. Each box with a dotted line represents a service. The parameter server provides two services, a specialized database service and a predictor service, and the database is used to store a global model referenced by the predictor. The predictor is used to train a policy network and pre-forward the global model to the workers depending on the prediction result. Each worker in the cluster is responsible for training the model based on different data shards and feeding the updated model into the server database. After providing an overview of the training scheme, we discuss the training flow between the server and a worker.

Fig. 9 shows the model training flow. The arrows indicate the directions in which data will pass. The rectangles represent the limited memory data structure for storing data, the parallelograms represent Tensorflow operations, and the cylinders indicate the data stored in the disk. The steps are described as follows:

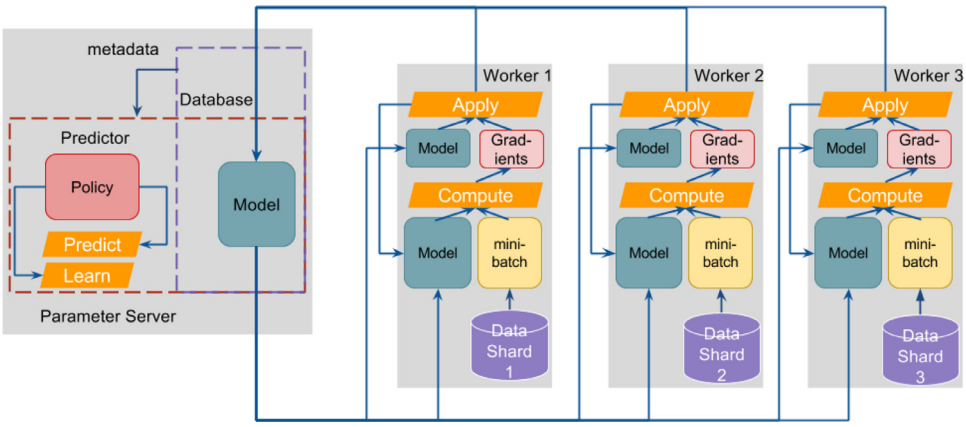

Fig. 8. MPPS training scheme.

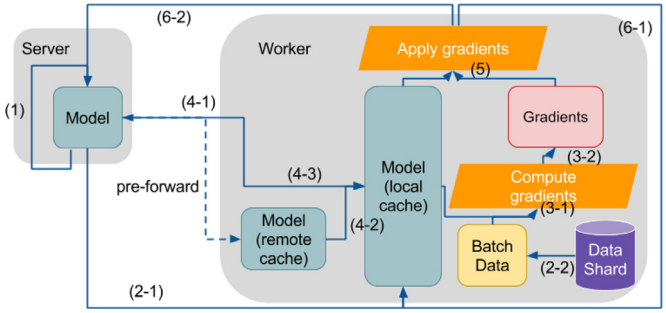

Fig. 9. Model training flow between the server and a worker.

- Init step (1): The parameter server initializes several variables in a multilayer neural network, whose shape is defined by the user before training. Our system supports training all models as long as they can be implemented with Tensorflow.
- Load step (2): The worker does not fetch the model and assign it to the local memory (aka local cache, described later) until the server finishes the Init step (2–1). In addition, the worker loads the training data from the disk to the main memory (2–2).
- Compute step (3): The worker feeds the model and minibatch data into the GPU memory to compute the gradients by feed-forwarding and backpropagation, which are accelerated by the GPU (3–1). The worker then loads the gradients from the GPU memory to the main memory (3–2). Incidentally, the gradients have already been multiplied by the learning rate; therefore, the worker need not multiply the gradients by the learning rate in the Apply step.
- Fetch step (4): To apply the gradients to the model, the worker must first download the latest model from the server. In an unoptimized way, the worker needs to send a request to the server to download the most recent model in the server; then, the server sends it to the worker. However, these worker−server interactions are slow and inefficient compared to the interprocess communication. Thus, we use two small amounts of memory as a cache to cache the model. One is a local cache, where Tensorflow will read and load data to the GPU, and the other is a remote cache, which stores data received from the predictor. The worker skips the Fetch step and goes to the next step if the cache hits. The caching flow is shown in Fig. 10.
  - Check the local cache: If the model in the local cache is the same as the server's, the worker proceeds to the next step; otherwise, it checks the remote cache.
  - Check the remote cache: If the model in the remote cache is the same as the server's, the worker copies the model from the remote cache to the local cache (4–2) and proceeds to the next step; otherwise, it fetches the model from the server.
  - Fetch data from the server (4–3): The worker asks the server to send the model to the worker.
- Apply step (5): When the worker has the latest model and the gradients, it applies the gradients to the model. This step is also executed in the GPU. In addition to subtracting only the gradients, we use the approach proposed by Zhang [16] to control the staleness problem in Async.

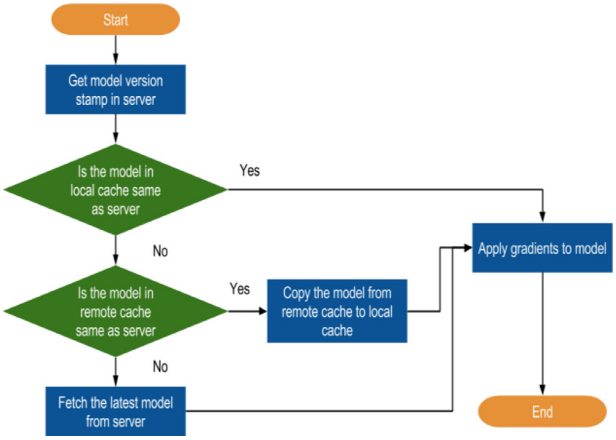

Fig. 10. Cache flowchart.

- Update step (6): After applying the gradients to the model, the worker has a latest model in the GPU. The worker then saves it to the local cache (6–1) and also uploads it to the server (6–2). The Update step is the last step in a training iteration; therefore, once the worker finishes this step, it goes back to the Compute step and continues until the loss threshold is reached.

3.2. Parameter server design

The main idea of designing our parameter server is to minimize the CPU overhead in the critical section. In the traditional parallelism described in Section II, the parameter server has to deal with the upload requests sent by the workers and apply the gradients to the global model, as shown in the following lines of pseudocode.

```
1. Start_Server:
2.   initialize a global lock
3.   initialize the weights in the model
4.   create a network listener and define event handlers
5.   start listening
6.   def event upload_handler(gradients){
7.     acquire the global lock
8.     preprocess and decompress the gradients
9.     for each layer in the model
10.        apply the gradients to the weights in the layer
11.        free the global lock
12.  }
13.  def event download_handler(){
14.    acquire the global lock
15.    preprocess the model
16.    free the global lock
17.    return the preprocessed model
18.  }
19.  signal to stop listening
20. End
```

In the parameter server design, the server will create a thread to handle the request from a worker. Lines 6–12 show how a server thread handles the upload request. Initially, the thread tries to acquire the global lock; if the global lock is not held by another server thread, the thread will get the lock successfully; otherwise, it is blocked until the lock is free. Next, the server thread deserializes the data from the socket buffer and decompresses them if they are compressed. The thread then applies the gradients to the model within a loop. The final step is to release the global lock. Lines 8–10 indicate a critical section according to the accepted definition. The overhead in this section involves preprocessing the data and some GPU computing, which cause additional latency and increase the probability of lock contention. In addition, lines 13–18 show how the download handler deals with the request in four instructions. The handler also needs the lock to prevent the race condition. Because the global lock is indispensable, we focus on how to minimize the size of the critical section as much as possible to reduce the influence of lock contention. We remove the applied job from the worker; consequently, the server only serves as a database. The new design is shown as follows:

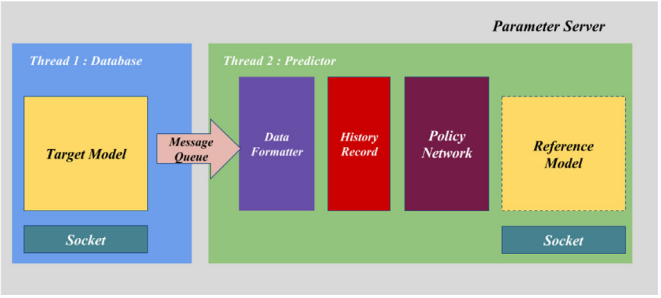

Fig. 11. Parameter server design.

```
1. Start_Server:
2.   initialize a global lock
3.   initialize the weights in the model and preprocess the model
4.   create a network listener and define an event handler
5.   create a predictor and start it
6.   start listening
7.   def event upload_handler(data){
8.      acquire the global lock
9.      assign data only to the model memory space
10.   free the global lock
11.   send a message to the predictor
12.   }
13.   def event download_handler(){
14.      acquire the global lock
15.      copy the model
16.      free the global lock
17.      return the copy
18.   }
19.   signal to stop listening
20. End
```

The critical section in the upload handler presented above comprises only line 8, which assigns the data to a memory space, namely, the local cache. This is simpler than the original design; the only difference is that the server thread pushes a message to the message queue to the predictor in line 11, but it is not in the critical section and hence does not matter. The optimized download handler also differs from the original. Because of the difference in the data structure of the model, our download handler need not preprocess the model before returning it to the worker.

Fig. 11 illustrates the internal design of the parameter server. As described previously, the parameter server has two services, the database and the predictor, which are standalone threads but are connected via a message queue. Here, we focus on the predictor design. The predictor has the following three primary components: the data formatter, history record, and policy network. The data formatter plays an important role that is similar to that of an interpreter in reinforcement learning and interpreting the environment to a string of states or a specific data structure. The second component, the history record, is a limited memory space storing all history states and predictions from the beginning of the training process to the end. The policy network is a neural network used to train a pre-forwarding policy to predict which worker will be the next to ask for the global model (in the Fetch step). Finally, the predictor will forward the global model to a worker according to the prediction result.

3.3. Computing worker design

Fig. 12 shows the design of the computing worker. A worker comprises two threads: a trainer and a model receiver. Each thread has its own memory space to save a model. However, the two models may be different. The memory space in the trainer is called the local cache and is created by Tensorflow. All Tensorflow APIs copy the model in this memory space to the GPU memory for computations, such as convolution and partial differential. The memory space in the receiver is called the remote cache, which is used to store the model pre-forwarded from the predictor. The main purpose of using two memory spaces is to differentiate the cache hierarchy, and the idea originates from the CPU cache mechanism. In a modern computer system, the CPU has an L1 cache and an L2 cache with different sizes and read-write speeds. The local cache is similar to the L1 cache, whereas the remote cache is similar to the L2 cache. However, the worker has to assign the model in the remote cache to the local cache before the GPU computation due to the Tensorflow API regulations.

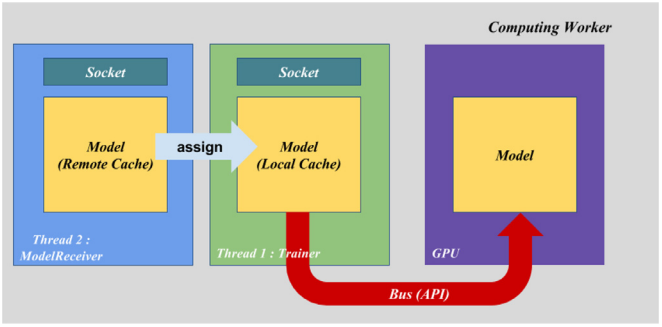

Fig. 12. Computing worker design.

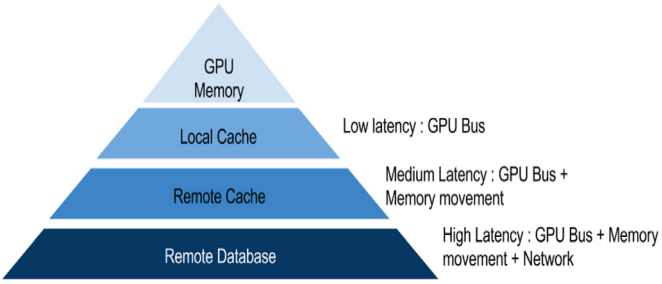

Fig. 13. Memory hierarchy.

3.4. Cache mechanism

In the original deep learning system, a worker holds a data cache, which is used to temporarily keep a small amount of training data and a few training labels to accelerate the CPU accessing. Some of the optimized systems use prefetching to preload data from the memory to the cache by using SSE prefetching instructions. The training data prefetching is static and plain. Workers can easily predict the prefetching distance and make a large improvement. Unfortunately, it only performs well in a single machine-training system. In a distributed deep learning system, the bottleneck is not the speed of loading the training data but rather the network latency. As described previously, the worker needs to synchronize the model trained by other workers in the Fetch step. This is time-consuming in network communication, especially when the model is extremely large. Therefore, we want to create a network cache to improve the performance in this situation. In our system, we apply the model cache for each computing worker. The model cache stores the partially trained model, which is the main content for the synchronization. In each iteration, the worker checks to determine if the cache hits. The network cache contains two small partitions, the local cache and the remote cache. If the local or remote cache hits, the trainer thread in the server fetches the model in the cache and feeds it into the GPU memory. Therefore, the process can be skipped over a network transmission.

Fig. 13 shows a schematic of the memory hierarchy. The top of the pyramid is the GPU memory, which is used by the GPU cores directly. The second layer is the local cache, which is a limited main memory assigned by Tensorflow. The third layer is the remote cache, which is also a limited main memory, and the bottom is a remote database, which is located in the parameter server. The latency is as shown in the figure. If a worker wants to use data in the remote cache for GPU computing, it has a higher latency than the local cache because the system needs to move data from the remote cache to the local cache and then to the GPU memory. However, using the model in the remote cache is still more efficient than accessing the model in the remote database.

The pseudocode algorithms of traditional data parallelism training BSP worker is shown as following code block.

```
1. Start_worker:
2. get initial model from server
3. for total_iterations {
4.   compute gradients
5. push gradient to server
6. wait to receive latest model from server
7. get latest model
8. }
9. End
```

And the pseudocode algorithms of our MPPS worker is following.

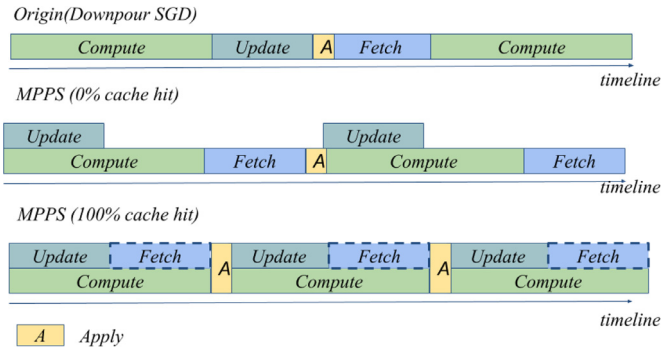

Fig. 14. Cache miss analysis.

```
1. Start_worker:
2. get initial model from server
3. for total_iterations {
4. compute gradients
5. get latest model (from cache or parameter server)
6. apply gradients to latest model
7.    create a thread to push new model to parameter server
8. }
9. End
```

Comparing to traditional worker running line 5–7, which is always a blocking call for sure, our MPPS worker finishes line 5 quickly if cache hits. And MPPS worker runs line 7 in a thread, hence, the main thread can continue to run next loop.

The improvement in system performance is directly proportional to the number of cache hits. The more the cache hits, the less time the worker spends in networking. Because we set two caches in our system, there are two types of caches: cache hits and cache misses. We define two freshness values to identify whether each cache is valid.

- Freshness L: the freshness value of the local cache.

$$Freshness\ L = VS\ in\ PS - VS\ in\ local\ cache \tag{4}$$

where *VS* is the version stamp, which is set as the version number of a model. The version stamp accumulates after one Apply step. Freshness L is the difference between the version stamp in the parameter server and that in the local cache. In Fig. 10, the worker calculates Freshness L to determine if the model in the local cache is same as that in the parameter server. A Freshness L equal to zero means that the models are the same; in other words, the local cache hits.

- Freshness R: the freshness value of the remote cache.

$$Freshness\ R = VS\ in\ PS - VS\ in\ remote\ cache \tag{5}$$

If Freshness L is not equal to zero, the worker calculates Freshness R and verifies whether the model in the remote cache is same as that in the server. If Freshness R is zero, the remote cache hits; else, it misses. Next, we express the cache hit rate as follows:

$$cache\ hit\ rate = \frac{local\ cache\ hit + remote\ cache\ hit}{sync\ times} \tag{6}$$

where sync times is the count of synchronizations, which is equivalent to the number of iterations. The total cache hit equals the number of local cache hits plus the number of remote cache hits. Intuitively, the cache hit rate has a positive correlation with the performance gain. We demonstrate two extreme examples of the training process with different cache hit rates in Fig. 14.

The first subfigure in Fig. 14 shows that the Downpour SGD process is always step-by-step. All steps are blocking executions, such as GPU computing and network I/O. Therefore, the entire training process is slow. In contrast, MPPS performs better in model training, and the timelines are presented in the second and third subfigures. Even if the cache hit rate is 0%, the worker can perform the Update step in the background by changing the system architecture and using a nonblocking socket; thus, the main thread can continue to perform the Compute step. Another example is 100% of the cache hit rate; the worker always fetches the model in either the local cache or the remote cache in the Fetch step. In this case, the network latency is hidden in the background or skipped. On average, the cache hit rate in our system will fall between 0% and 100%. Nevertheless, the performance is always better than that of Downpour SGD.

3.5. Cache pre-forwarding

In this subsection, we introduce the cache pre-forwarding policy, which is used to actively increase the cache hit probability. The cache pre-forwarding concept is associated with cache prefetching. Both approaches emphasize preloading data

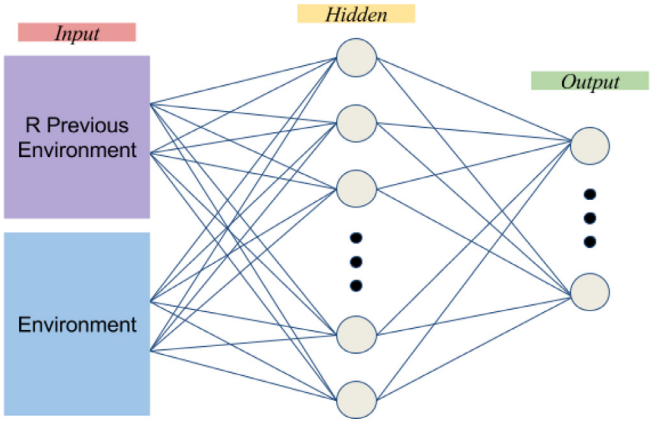

Fig. 15. Policy network.

to the cache before the data are needed; the difference lies in who takes the initiative. Cache prefetching means that the worker initiatively preloads the data to the cache. In contrast, cache pre-forwarding means that the server pre-forwards data to the cache in the worker. As described previously, workers have two blocks of cache, local and remote. The local cache is updated after the worker finishes the Apply step, where the model in the GPU will be written to the local cache. The remote cache differs from the local cache. The model receiver is always listening for the parameter server to receive the model pre-forwarded from the server irregularly. Therefore, the remote cache only updates when the receiver obtains a new model from the server. If the server never sends a model to the worker, the remote cache never hits. To increase the number of remote cache hits, the parameter server needs to pre-forward the latest model in a timely manner. Considering the network bandwidth, the server is unable to forward the latest model to all workers; in contrast, the server needs a policy or schedule to forward the model intelligently. We use reinforcement learning to train the pre-forwarding policy, which can learn from the environment automatically, to meet our expectation. We formulate the problem as follows:

- Environment and state: The environment is a set of worker statuses, including the computation time in one iteration for each worker and the timestamp of when the worker sends a new model in the Update step. The interpreter encodes the environment to a state as follows:

$$CompT_i = (T_1, T_2, ..., T_n)$$

$$TSTP_i = normalize(tstp_1, tstp_2, ..., tstp_n)$$

$$Rec_i = (CompT_i, TSTP_i)$$

$$State_i = (Rec_i, TSTP_{i-1}, ..., TSTP_{i-R}), R \in N$$

where $tstp_n$ is the n^{th} worker's timestamp, T_n is the n^{th} worker's computation time, $State_i$ is the i^{th} state, and R is the reference window size, which is used to indicate the number of history *TSTP*s added to the current state. The origin policy network training, which works like the Markov chain, makes the decision based only on the current environment without any historical record. However, for a more realistic system configuration, we add R previous *TSTP*s to the environment interpretation. In a cloud cluster, the computing resource of each machine may be different. One worker may run faster than the others; for example, a worker may perform the Update step twice when the others have only performed it once. Therefore, by setting a nonzero R value, our policy can learn the worker execution pattern.

- Agent: The agent in the parameter server is the predictor. One of the jobs of the predictor is to pre-forward the latest model to one worker in each round. This worker is predicted as the one who will perform the Fetch step immediately. Another job is to train the policy according to the reward from the environment. Therefore, the policy will improve and gradually have less loss.
- Policy: We use a three-layer artificial neural network as the policy network. The neurons in each layer vary according to the number of workers and the R value. The neurons in the input layer are $R^*n + n + n$, and the number of neurons in the output layer is equal the number of workers. The policy network is shown in Fig. 15.
- Action: The predictor pre-forwards the model to the worker according to the prediction result.
- Reward: The reward equals 1 if the prediction is right; else, it equals -1.
- Training frequency: We train five history states per update request received.

Fig. 16 shows a schematic of the policy training and prediction in the predictor. The arrows represent the directions of data flow. Additionally, we explain the meaning of each step as follows:

- The predictor receives a message from the message queue and interprets it in the data formatter.

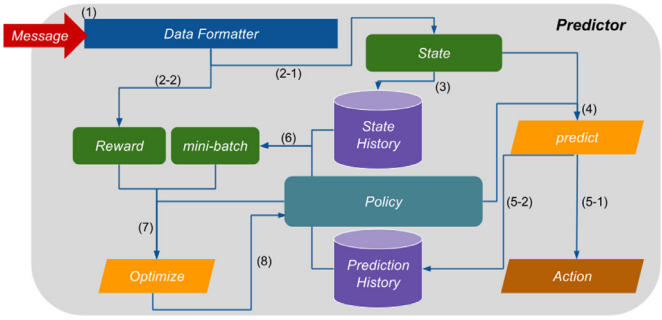

Fig. 16. Policy training and prediction flow.

- (2–1)(2–2) According to the R factor, the state and reward are generated.
- The predictor pushes the current state into a state history database.
- The predictor uses the policy network and current state to make a prediction.
- (5–1) The predictor pre-forwards the model based on the prediction results.
- (5–2) In this step, the prediction result, generated in step (5–1), is pushed into a prediction history database.
- (6) To perform self-learning, the predictor fetches a minibatch dataset from the state database and prediction database.
- (7) We use a minibatch dataset, a reward as input, and train the policy network to decrease the loss.
- (8) The policy network is updated.

These steps constitute the process of how the predictor manages a message, including making a prediction, model pre-forwarding, and self-learning. In summary, our predictor works as a smart assistant in the distributed deep learning system. Because the parameter server supervises all events from the workers, it has a global view to perform the cache pre-forwarding so as to optimize the training flow. Furthermore, we use a dynamic policy rather than static scheduling, where the policy network can self-learn during the process and is suitable for any environmental setting with high flexibility. The experiments presented in Section IV confirm that this policy network can significantly improve the system throughput.

4. Experimental results

4.1. System implementation

In the experiments, we build a system with Python and Tensorflow. The Tensorflow API is only used for model training and policy training, such as computing the gradients and applying the gradients. The other components in the system are implemented in Python, such as the networking and cache. Next, we briefly explain the implementation of each component.

Because the general mathematical operation and learning computation are already provided by Tensorflow, we only need to implement the cross-machine communication to build a distributed deep learning system. We choose RPC as our inter-process communication approach. To obtain the best performance, we compare two RPC libraries in Fig. 17.

As shown in Fig. 17, XML-RPC spends approximately ten times the cost in network communication. Therefore, we choose Thrift as our networking framework.

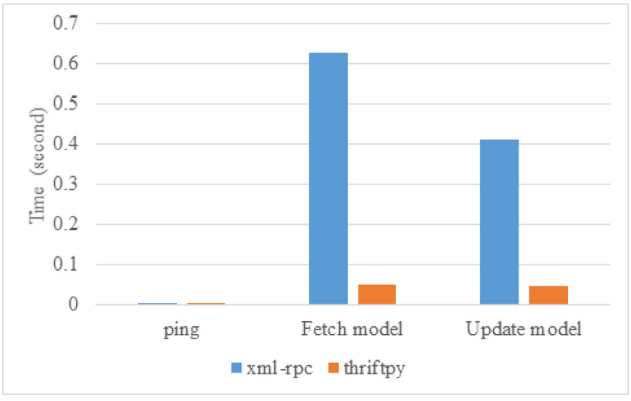

Fig. 17. Performance comparison of XML-RPC and Thrift.

Table 1
Model compression ratio.

Target	Model size (byte)	Compression ratio
Origin	4,273,192	X
DEFLATE	3,953,924	1.08
Float16	2,136,596	2
Float16+DEFLATE	1,959,108	2.18

Table 2
Software specifications.

Item	Content
OS	Ubuntu 14.04 Desktop 64 bit
Tensorflow	0.12.1
Python	2.7.6
Thrift	thriftpy 0.39

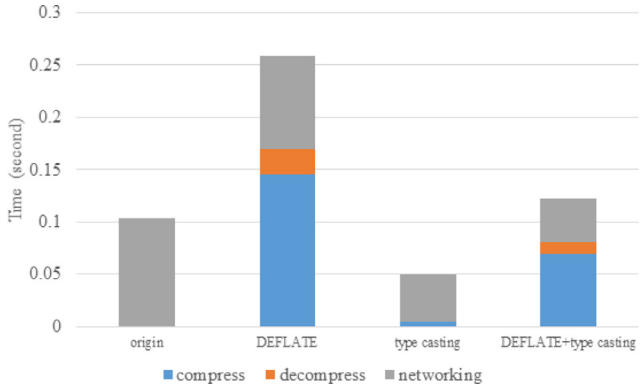

Fig. 18. Compression and networking performance.

Next, we analyze the compression algorithm and the effectiveness of compressing the model before networking. In a distributed computing system, two nodes usually have to exchange information. In a limited-network-bandwidth environment, researchers apply data compression to reduce the data size to reduce network latency. Compression can be either lossy or lossless. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Lossy compression reduces bits by removing unnecessary or less important information. We try to apply two algorithms to compress the model size. First, we choose zlib as the lossless compression library, which is a software library and a de facto standard. Zlib supports only one algorithm, called DEFLATE, and a lossless data compression algorithm and associated file format that uses a combination of the LZ77 algorithm and Huffman coding. Additionally, for lossy compression, we use type casting to decrease the bit usage in presenting a numerical value. The original data type used in deep learning is a 32-bit floating point, but we use a half-precision floating point, which is only 16 bits. This compression approach makes the model learning process work like half-precision learning. Tim [17] also proposed 8-bit approximations for distributed deep learning. Table I shows the compression ratios of the original system without compression and those using DEFLATE, Float16 and their combination. In addition, Fig. 18 illustrates the results in a bar chart.

The experimental environment is in a limited-bandwidth network and uses thriftpy as a cross-machine communication interface. In this empirical method, the client compresses a model before sending it to the server, the server echoes the message back to the client immediately, and the client decompresses the model as the final step. We measure the time durations of the four approaches. The results show that the DEFLATE algorithm takes 0.14 s to compress an ~4.2 MB model, and the compression ratio is 1.08. By contrast, the type casting approach takes 0.004 s to compress the model, the compression ratio increases to 2, and the mixed version has a ratio of 2.18. These statistics show that only type casting spends less time than the original method. The other two approaches spend too much time on data compression and decompression. Even though they reduce the model size and networking latency, the overall benefit is lower due to the slow compression operation. Therefore, in the following experiments, we use only type casting as a compression approach to compare with the original technique in our distributed deep learning system (Table 1).

4.2. Experimental environment and settings

The software specifications are detailed in Table 2. To emulate a real-world cloud cluster environment in which not all workers run on different machines, we use two cases in the experimental environments. The first case is a hybrid mode

Table 3
Model training configuration.

Dataset	Dataset size	Model	Floating point	Model size
CIFAR-10	50,000 images	Convolutional Neural Network	32 bits	4.2 MB
			16 bits	2.1 MB

containing two machines: one with one server and two workers and the other with one worker. The second case is a virtual mode with only one machine run with one server and three workers. We call the worker that owns the entire machine resource the real worker and that sharing the resource with others the virtual worker.

The experiments use the CIFAR-10 dataset for image classification, and the model training configuration is listed in Table 3. This dataset contains 50,000 images labeled in 10 classes. Each image is composed of $32 \times 32 \times 3$ pixels; the first two figures are the width and height, respectively, and the last is the image depth, which comprises the RGB layer. For a realistic image input, we distort the images and randomly crop them to smaller sizes. Therefore, the number of neurons in the input layer is $28 \times 28 \times 3$, and that in the output layer is 10. The remaining layers in the model include two convolutional layers and three fully connected layers. The model has approximately one million connections for each image, and the model sizes are 4.2 and 2.1 MB (compressed by type casting), respectively. As mentioned in the previous section, type casting is an efficient compression approach for reducing the model size and network latency by the float16 data type. Hence, in the following experiments, we use the same model but different data types and compare their performances. The other hyperparameters of the model include the learning, which is set to 0.003, and the minibatch size, which is set to 200.

4.3. Experimental results

To evaluate the proposed computing system, several objectives are defined and described before presenting the experimental results. We explain the following five evaluation objectives:

- Throughput: This is the main objective used to evaluate the proposed system. We previously indicated that the data throughput is a critical performance indicator in the big data era. The processing system must be as fast as possible to manage millions or even billions of pieces of data in a few seconds. Therefore, we set the throughput as our indicator. The goal is to improve the amount of data trained in one second over the Downpour SGD architecture.
- Critical section reduction: In Section III, we mentioned that we minimize the functionality of the server by removing some data operations from the worker. This design reduces the size of the critical section. However, its effectiveness must be validated.
- Policy network hyperparameter: We use the policy network as our predictor policy and test the model pre-forwarding on different sets of hyperparameters to find the best set with the lowest training loss and rapid convergence.
- Cache hit rate: The cache hit rate has a positive correlation with the throughput; narrowly speaking, it has a positive correlation with the time efficiency in the Fetch step. If the cache hit rate is higher, the duration of all Fetch steps is lower. On the other hand, if the cache hit rate is lower, it takes more time in the network I/O to exchange the information.
- Cache hit distribution: The cache hit is the sum of the local cache hit count and the remote cache hit count. One way to evaluate the efficiency of our predictor is to analyze the cache hit distribution. A more remote cache hit means that our cache pre-forwarding approach is significantly effective in practice.

These five objectives are the key performance indicators of our distributed deep learning system. In the next subsection, we present the experimental results to confirm our work.

4.3.1. Throughput

The first experiment focuses on the system data throughput. The control group is Downpour SGD, and the experimental groups are MPPS32 and MPPS16. MPPS32 means that the data type of the model is 32-bit floating point, whereas MPPS16 means that the data type is 16-bit floating point. To prevent the extreme case, we run each group 10 times and average the results. Fig. 19 shows the training throughput. C1RW means the real worker in case 1, whereas C1VW means the virtual worker in case 2. In addition, C1 and C2 represent the overall system throughputs of cases 1 and 2, respectively.

As shown in Fig. 19, MPPS16 has the highest throughput, and it achieves $1.65\times$ and $1.51\times$ speedups in C1 and C2, respectively, compared with Downpour SGD. MPPS32 also as greater throughputs than Downpour SGD, and the speedups are $1.16\times$ and $1.09\times$, respectively. As expected, all experiment groups perform better than Downpour SGD because of the redistribution of tasks and model caching in the worker. Moreover, MPPS16 is significantly better than MPPS32 due to the data reduction in the cross-machine communication. In summary, MPPS exhibits better data throughput than Downpour SGD. Additionally, given the same computational load, MPPS can provide a greater performance gain if the synchronization content is smaller.

4.3.2. Critical section reduction

Lock contention occurs in the parameter server when the workers attempt to update the model to the server simultaneously, and it can significantly decrease the performance. We reduce the critical section in MPPS; theoretically, MPPS has

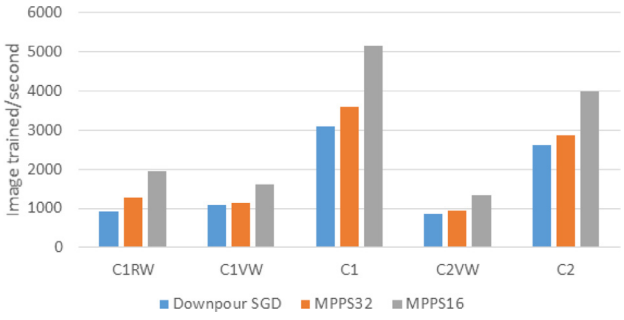

Fig. 19. Data throughputs.

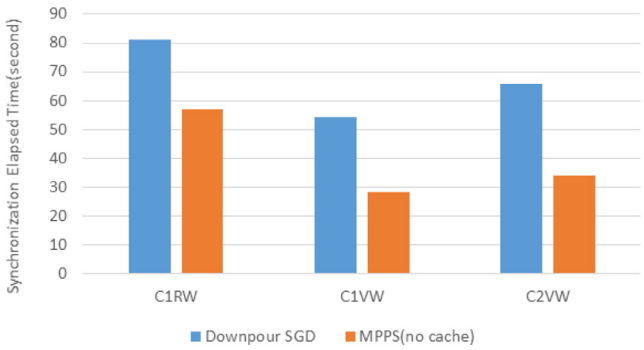

Fig. 20. Critical section reduction.

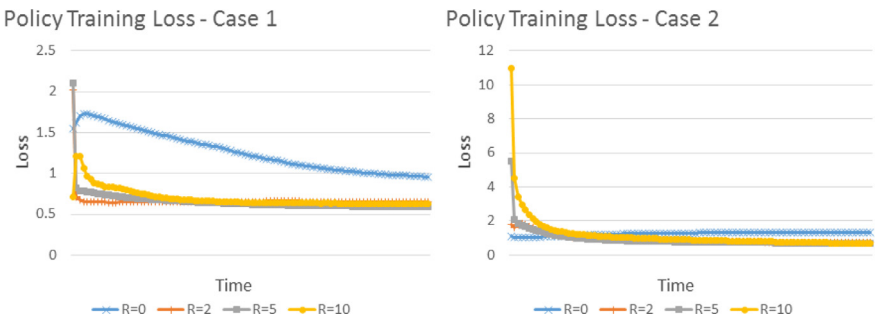

Fig. 21. Policy training losses on MPPS32.

higher performance than Downpour SGD. To measure the effectiveness of the critical section reduction, we perform two experiments using Downpour SGD and MPPS (without a cache). We capture the durations of the Fetch step and the Update step. This is because the workers communicate with the server for model synchronization in these two steps. Every algorithm trains 30,000 images at a time and is performed ten times.

Fig. 20 shows the experimental results of the critical section reduction effectiveness. MPPS without the cache requires approximately 70% of the time of Downpour SGD in C1RW and 52% of the time in C1VW and C2VW.

4.3.3. Policy network hyperparameter

Because we use the policy network as the predictor self-learning algorithm, we need to find the best set of hyperparameters for the network to obtain the highest cache pre-forwarding accuracy. In this experiment, we set four different R values: 0, 2, 5, and 10. Consequently, the numbers of neurons in the input layer are 6, 12, 21, and 36, respectively, where n is 3.

Fig. 21 show the policy training losses on MPPS32 in case 1 and case 2. The training loss curves are slightly different because MPPS is executed in an open environment. The workers and server are executed on the operation system and connected with a LAN. A context switch or network failure, such as a packet loss, leads to uncertainty and irregularity for the policy training. Additionally, our policy network uses small minibatch training, which is easily affected by noise. Nevertheless, there is still a tendency to converge gradually. Thus, we conclude that the predictor can obtain higher accuracy

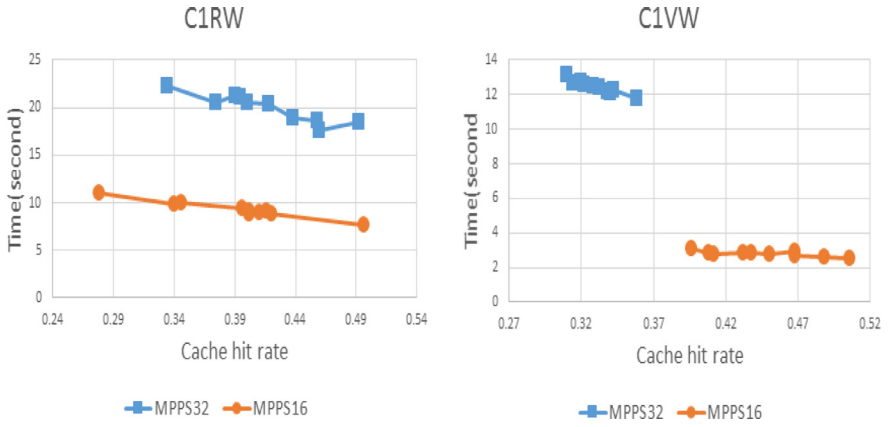

Fig. 22. Cache hit rate and elapsed time of the Fetch step in case 1.

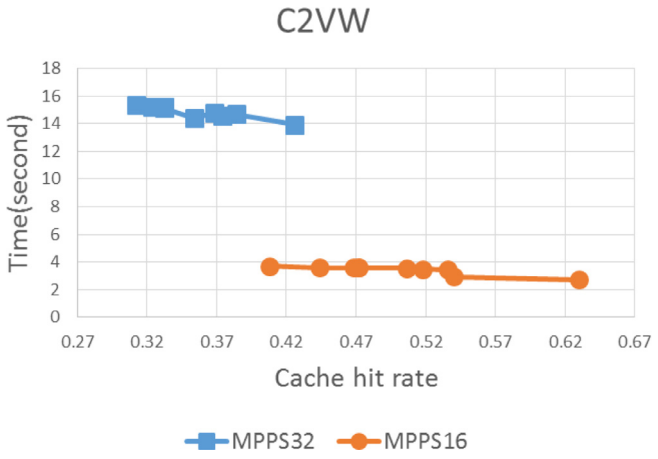

Fig. 23. Cache hit rate and elapsed time of the Fetch step in case 2.

when R is 2 or 5. When R is equal to 0 or 10, the loss is higher. Thus, we infer that the R value should be set to be neither too large nor too small.

4.3.4. Cache hit rate

MPPS uses the cache to reduce the model synchronization times. If the cache hits, the worker skips downloading the model from the server in the Fetch step. Therefore, the cache hit rate is theoretically inversely proportional to the elapsed time of the Fetch step. To validate this theory, we conduct experiments on MPPS32 and MPPS16 to test the cache hit rate effectiveness. Each algorithm trains 30,000 images at once and is performed ten times.

Figs. 22 and 23 show the relationships between the cache hit rate and the elapsed time of the Fetch step in cases 1 and 2, respectively. To easily understand these relationships, we sort the experimental results by the cache hit rate. The X-axis indicates the cache hit rate, and the Y-axis indicates the elapsed time of the Fetch step. The results show that the total elapsed time of the Fetch step decreases with an increase in the cache hit rate; all workers have this tendency. These results confirm that the relationship is consistent with our theory and that the model cache design can improve the system performance.

4.3.5. Cache hit distribution

The cache pre-forwarding affects only the remote cache hit. To evaluate the approach's effectiveness, we conduct several additional experiments on MPPS32 and MPPS16, and each algorithm trains 30,000 images. We show the cache hit distribution in Fig. 24.

The percentage of the remote cache in C1RW is approximately 3.7% on the MPPS32 system and 13.9% on the MPPS16 system. The remote cache hit in C1VW is approximately 31% on MPPS32 and 56% on MPPS16. The remote cache hit proportion varies in these two workers because the network latency between C1RW and the parameter server is higher than that between C1VW and the parameter server. Even though the pre-forwarding policy is always accurate, the parameter server pre-forwards the model late, which reduces the remote cache hits. In contrast, pre-forwarding works better in a virtual

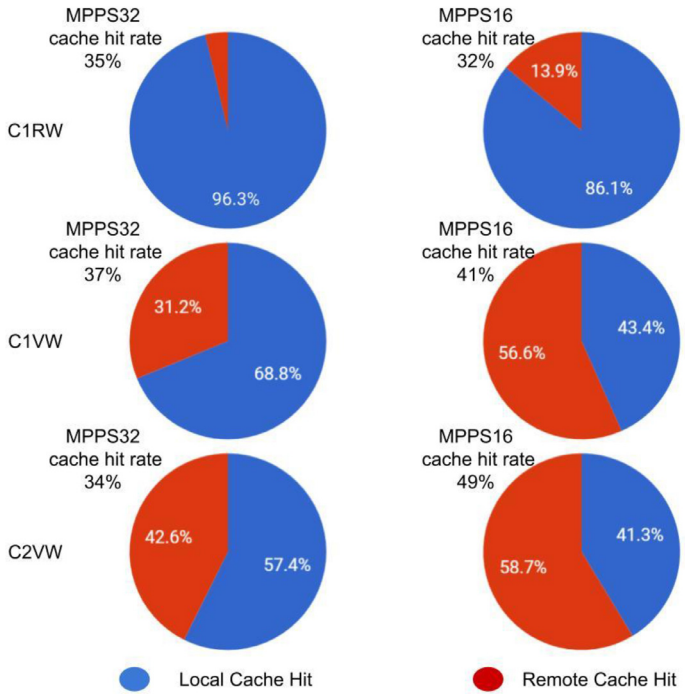

Fig. 24. Cache hit distribution.

worker. The experimental results show that the pre-forwarding policy can improve the cache hit rate by approximately 11% on MPPS32 and by 23% on MPPS16 in C1VW.

Even if the pre-forwarding policy is always accurate, the parameter server pre-forwards the model late, which makes the remote cache hit times fewer. In case 2, data transfers are via the local network since all virtual workers and parameter server are on the same machine, model pre-forwarding can be performed quickly. With the growth of remote cache hit, the overall cache hit rate is improved. Consequently, pre-forwarding works better in a virtual worker by contrast.

The proportions of the remote cache hit are quite different on MPPS32 and MPPS16. We infer this to be due to the reduction in network latency. The remote cache hit has a relatively high percentage on MPPS16 because the predictor can pre-forward the model to the remote cache quickly when the model is smaller, meaning that the latency is lower. In contrast, the predictor in MPPS32 spends more time on network transmission even if the prediction is correct and may miss the time that the worker executes the Fetch step.

5. Conclusion and future work

We reduce the parameter server workload and adopt a model cache to reduce cross-machine communications. In addition, based on the concept of cache prefetching, we propose cache pre-forwarding to increase the cache hit rate. This policy is an adaptive algorithm in that it automatically adjusts during the model training process, irrespective of the system configuration, by using reinforcement learning.

To confirm the effectiveness of the proposed system, we conduct multiple experiments on the system throughput, critical section effectiveness, and cache hit effectiveness. The experimental results show that the system can improve the system performance compared with the Downpour SGD system. Although the average cache hit rate is only approximately 40%, we believe that the hit rate can still be increased by optimizing the policy network.

The second one is to upgrade the hardware resources, such as using multiple ethernet cards to parallelize the network transmission or using RDMA. With RDMA support, the parameter server and workers can directly access the model on remote machine without the involvement of the network software stack. Moreover, these data transfers would not consume any CPU time in the remote server. However, hardware upgrading still needs a software algorithm to reach the most beneficial performance.

In the future, we will further optimize the system throughput. The bottleneck of distributed deep learning is network latency, and we can only solve part of it by reducing the networking times. There are still several approaches to improve the performance. The first is model compression, which reduces the synchronization content to reduce network latency; this requires real-time compression rather than a high compression ratio. Otherwise, the worker requires too much CPU time for model compression, and the network latency is decreased; there is a trade-off between them. The second is to upgrade the hardware resources, such as using multiple ethernet cards to parallelize the network transmission or using RDMA to support

zero-copy networking. However, hardware upgrading still requires a software algorithm to attain the highest performance. Therefore, we want to develop new algorithms for these scenarios. In addition to the network issue, we want to improve the cache pre-forwarding policy. The policy network can be tuned with different hyperparameters to obtain the best prediction accuracy. Finally, we believe that reinforcement learning can be applied on different occasions. For example, given a distributed deep learning cluster for which each machine has different computing resources, the master automatically assigns the task to the workers according to a special policy. The master can learn an assignment policy during the training process and average the workload according to the computational ability. This approach can theoretically improve the system performance, and we are interested in exploring this topic.

Declaration of Competing Interest

The author has declared that no competing interests exist.

Acknowledgment

This work was supported by the "Allied Advanced Intelligent Biomedical Research Center, STUST" under Higher Education Sprout Project, Ministry of Education, Tainan, Taiwan.

References

[1] Valiant LG. A bridging model for parallel computation. Commun ACM Aug. 1990;33(8):103–11.
[2] Williams RJ. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn 1992;8(3–4):229–56.
[3] Sutton RS, Barto AG. Introduction to reinforcement learning, 135. Cambridge: MIT Press; 1998.
[4] Hinton GE, Osindero S, Yee-Whye T. A fast learning algorithm for deep belief nets. Neural Comput July 2006;18(7):1527–54.
[5] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM 2008;51(1):107–13.
[6] Liao S-W, Hung T-H, Nguyen D, Chou C, Tu C, Zhou H. Machine learning-based prefetch optimization for data center applications. In: Proceedings of the conference on high performance computing networking, storage and analysis. IEEE; 2009.
[7] Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. 2010 IEEE 26th symposium on Mass storage systems and technologies (MSST); 2010.
[8] Vanhoucke V, Senior A, Mark ZM. Improving the speed of neural networks on CPUs. In: Proc. deep learning and unsupervised feature learning NIPS workshop, 1; 2011.
[9] Lee J, Kim H, Vuduc R. When prefetching works, when it doesn't, and why. ACM Trans Arch Code Optim 2012;9(1).
[10] Dean J, et al. Large scale distributed deep networks. In: NIPS'12 proceedings of the 25th international conference on neural information processing systems, 1; 2012. p. 1223–31.
[11] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In: NIPS'12 proceedings of the 25th international conference on neural information processing systems, 1; 2012. p. 1097–105.
[12] Ho Q, Cipar J, Cui H, Kim JK, Lee S, Phillip B G, Gibson GA, Ganger GR, Xing EP. More effective distributed ml via a stale synchronous parallel parameter server. In: NIPS'13 proceedings of the 26th international conference on neural information processing systems, 1; 2013. p. 1223–31.
[13] Chilimbi T, Suzue Y, Apacible J, Kalyanaraman K. Project Adam: building an efficient and scalable deep learning training system. 11th USENIX symposium on operating systems design and implementation, 14; 2014.
[14] Maldikar P. Adaptive cache prefetching using machine learning and monitoring hardware performance counters. Diss. University of Minnesota; 2014.
[15] Li Mu, Andersen DG, Park JW, Smola AJ, Ahmed A, Josifovski V, Long J, Shekita EJ, Su B-Y. Scaling distributed machine learning with the parameter server. In: OSDI'14 Proceedings of the 11th USENIX conference on operating systems design and implementation, 1; 2014. p. 583–98.
[16] Zhang, W., S. Gupta, X. Lian, Ji Liu, "Staleness-aware async-sgd for distributed deep learning", arXiv:1511.05950, 2015.
[17] Dettmers, T., "8-bit approximations for parallelism in deep learning", arXiv:1511.04561, 2015.
[18] Chen C-FuR, Lee GGC, Xia Y, Sabrina Lin W, Suzumura T, Ching-Yung L. Efficient multi-training framework of image deep learning on GPU cluster. 2015 IEEE international symposium multimedia (ISM); 2015.
[19] Hegde V, Usmani S. Parallel and distributed deep learning. Stanford University; June 2016. Tech. reporthttps://stanford.edu/~rezab/dao/projects_reports/hedge_usmani. pdf, 2016.
[20] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yuan Yu, Zheng X. TensorFlow: a system for large-scale machine learning, 16. OSDI; 2016.
[21] Cui H, Zhang H, Ganger GR, Gibbons PB, Xing EP. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In: Proceedings of the eleventh european conference on computer systems. ACM; 2016.
[22] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillicrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D. Mastering the game of Go with deep neural networks and tree search. Nature 2016;529:484–9.
[23] Gupta S, Zhang W, Wang F. Model accuracy and runtime tradeoff in distributed deep learning: a systematic study. Data mining (ICDM), 2016 IEEE 16th international conference on. IEEE; 2016.
[24] Cobb J, Aarag HEl. Web proxy cache replacement scheme based on back-propagation neural network. J. Syst. Softw. 2008:1539–58. URL https://www.sciencedirect.com/science/article/pii/S016412120700249X] .
[25] Tsai KC, Wang Li, Han Z. Mobile social media networks caching with convolutional neural network. IEEE wireless communications and networking conference workshops; 2018. URL https://ieeexplore.ieee.org/abstract/document/8368988] .
[26] Hardy C, Merrer EL, Sericola B. Distributed deep learning on edge-devices: Feasibility via adaptive compression. IEEE 16th International Symposium on Network Computing and Applications (NCA); 2017. doi:10.1109/NCA.2017.8171350.
[27] Wu Z, Lu Z, Patrick C, Hung K, Huang S-C, Tong Yu, Wang Z. QaMeC: a QoS-driven IoVs application optimizing deployment scheme in multimedia edge clouds. Future Gener. Comput. Syst. 2019;92:17–28.
[28] Chen X, Tang S, Lu Z, Wu J, Duan Y, Huang S-C, Tang Q. iDiSC: a new approach to iot-data-intensive service components deployment in edge-cloud-hybrid system. IEEE Access 2019;7:59172–84.
[29] Qiao W, Li Y, Wu Z H. DLTAP: a network-efficient scheduling method for distributed deep learning workload in containerized cluster environment. In: ITM Web of Conferences, 12; 2017. p. 03030.
[30] Ooi BC, Tan KL, Wang S, et al. SINGA: A Distributed Deep Learning Platform. MM '15: Proceedings of the 23rd ACM international conference on Multimedia; 2015. p. 685–8.

Sheng-Tzong Cheng received a B.S. (1985) and an M.S. (1987) in electrical engineering from National Taiwan University, Taipei, Taiwan. He received an MS (1993) and a PhD (1995) in computer science from the University of Maryland, College Park, MD, USA. He was an assistant professor of Computer Science and Information Engineering at National Dong Hwa University, Hualien, Taiwan, in 1995. He joined the Department of Computer Science and Information Engineering, National Cheng-Kung University (NCKU), Tainan, Taiwan in 1997 and became an associate professor and a full professor in 1999 and 2004 respectively. He was the recipient of the Lee, Kuo-Din Research Award in 2002, highlighting his research on multimedia topics and wireless-communication topics. He also received several awards from the Ministry of Education and the Institute of Information Industry. He has published more than 130 journal and conference papers. Currently, he is directing the Wireless Communication and Mobile Network Laboratory at NCKU. His research interests include design-and-performance analysis of mobile computing, wireless communications, multimedia, quantum computing, and real-time systems.

Chih-Wei Hsu is the semiconductor automated process software development center and chairman of the board of United Smart Electronics Corporation in the Taiwan (www.usai.com.tw), and He is received the Ph.D. (2015) in Computer Science and Information Engineering at National Cheng Kung University, Taiwan. Currently, the research at the Institute Informa-tion for Industry include Internet multimedia streaming, sensor networks, AI,IIoT,community,intelligence system (AOI system),5G SDN of security system (OpenDaylight,Security for industrial communications) and Big data for sensors network applications.

Gwo-Jiun Horng received an M.S. (2008) in electronic engineering from National Kaohsiung University of Applied Sciences, Kaohsiung, Taiwan. He is received the Ph.D. (2013) in Computer Science and Information Engineering at National Cheng Kung University, Taiwan. He is currently an associate professor in the Department of Computer Science and Information Engineering, Southern Taiwan University of Science and Technology, Tainan, Taiwan. His research interests include mobile service, internet of things, intelligent computing, cloud networks.

Che-Hsuan Lin received the M.S. in Computer Science and Information Engineering from the National Cheng-Kung University, Tainan, Taiwan. His research interests include mobile computing, intelligent systems, and cloud computing.