Edge-Assisted Federated Learning: An Empirical Study from Software Decomposition Perspective

Yimin Shi[1,2], Haihan Duan[1,2], Yuanfang Chi[3], Keke Gai[4], and Wei Cai[1,2](✉)

[1] The Chinese University of Hong Kong, Shenzhen, Shenzhen, China
{yiminshi,haihanduan}@link.cuhk.edu.cn
caiwei@cuhk.edu.cn

[2] Shenzhen Institute of Artificial Intelligence and Robotics for Society, Shenzhen, China

[3] The University of British Columbia, Vancouver, Canada
yuanchi@ece.ubc.ca

[4] Beijing Insitute of Technology, Beijing, China
gaikeke@bit.edu.cn

Abstract. Federated learning is considered to be a privacy-preserving collaborative machine learning training method. However, due to the general limitation of the computing ability of the terminal device, the training efficiency becomes an issue when training some complex deep neural network models. On the other hand, edges, the nearby stationary devices with higher computational capacity, might serve as a help. This paper presents the design of a component-based federated learning framework, which facilitates the offloading of training layers to nearby edge devices while preserving the users' privacy. We conduct an empirical study on a classic convolutional neural network to validate our framework. Experiments show that this method can effectively shorten the time cost for mobile terminals to perform local training in the federated learning process.

Keywords: Federated learning · Deep learning · Mobile edge computing · Program decomposition · Distributed computing

1 Introduction

In a traditional cloud-centric approach, if the cloud wants to utilize the data from mobile terminal devices to train a Machine Learning (ML) model, local data of the terminal has to be directly uploaded to the cloud, which can hardly preserve the privacy of clients and will pose a burden on the backbone networks [7].

This work was supported by Project 61902333 supported by National Natural Science Foundation of China, by the Key Area R&D Program of Guangdong Province with grant No. 2018B030338001, by the Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS).

Currently, Federated Learning (FL) has been introduced as a privacy-preserving distributed ML approach [10]. In a gradient-descent based FL, mobile terminals will locally train the given model with its privacy-sensitive dataset. Meanwhile, based on the current parameters, all terminals will return different sets of parameter gradients to the cloud [17]. This approach avoids directly sending the dataset to the untrusted cloud, which makes it possible to preserve user privacy and reduce network overhead.

Compare to the traditional cloud-centric approach, FL migrates part of the task from the cloud to the terminal. However, the computation resource of the mobile terminal device varies and is generally constrained. When some more complicated ML models like Deep Neural Network (DNN) [5] are trained with multimedia datasets, it may be too time-consuming for the terminal to complete the task. However, the computing power of the mobile edge computing (MEC) device is much stronger than that of the terminal. The distance and the communication cost between edge and terminal are relatively low, and it will not create a significant burden on the backbone network. Besides, the edge has higher security than the cloud [9]. Therefore, the edge may help solve this problem.

In terms of model training efficiency optimization, current works focus on the following two aspects: the optimization of the traditional FL procedure by improving the existing protocol and the optimization of the DNN training process by applying distributed methods. However, these works do not consider how to apply the distributed methods to accelerate the local update process of FL with the help of MEC.

In our work, we propose a component-based FL framework that can offload some of the training layers of the DNN model to the nearby edge while preserving the privacy of the terminal user. The local parameter updating process is further decomposed into inter-independent components by coarse-grained program decomposition, and each component can be separately deployed either on the edge or the terminal. Each layer component can independently complete the forward and backward propagation. We conduct an empirical study on a classic convolutional neural network (CNN). Experiments show that this framework can efficiently shorten the time cost of local training. The optimal strategy of component deployment and the effectiveness of user privacy preservation in different edge trusting situations are separately discussed.

The remaining sections of the paper are organized as follows. We review some related works in Sect. 2, and discuss the general method applied for layers decomposition of DNN in Sect. 3. Section 4 presents the architecture and the program design of the proposed component-based distributed training system. In Sect. 5, we discuss different trusting situations of the edge device and provide the optimal strategies which can efficiently preserve user privacy. Moreover, in Sect. 6, we present the results of a set of experiments that explore the efficiency of the system in different conditions. Finally, Sect. 7 concludes the paper and discusses the aspects that can be further studied on the basis of this work.

2 Related Work

In the field of federated learning, current researches on the optimization of FL focus on at least two aspects as follow: (1) User privacy preservation and (2) FL efficiency improvement. For users' privacy preservation, some existing protocols [1] have been proven to be effective. But some existing cloud storage protection method [13] is challenging to use directly in the FL process. For FL efficiency improvement, recent works discuss how to make user selection [11] for clients during the learning process, as well as how to determine the frequency of local up-dates and global aggregation [16]. Our work focuses on improving the efficiency of the local updates through the distributed method with the help of MEC, which is different from the above works and does not contradict each other.

In the field of software program decomposition, currently, there are two primary schemes: Fine-Grained Decomposition and Coarse-Grained Decomposition. Coarse-grained decomposition partitions the program into a set of functional-independent and stateless components. A web-based coarse-grained program decomposition platform has been developed in the current work [2] [3], which provides a good execution engine and API for software decomposition. In this work, we apply the coarse-grained decomposition to decompose the layers in a CNN into different functional components. The number of components is ensured to be sufficient to provide us with more possible component deployment strategies.

In the field of the distributed deep neural network, current works focus on the following aspects: (1) Improving the training efficiency of DNN through distributed parallel training method [4]. (2) Using distributed DNN to do the training process or forward propagation to protect user privacy [8,12,15]. However, the existing works do not carry out program design or deployment strategy under the framework of FL, do not guarantee user privacy, or propose a suitable decomposition schema. In this paper, we decompose the training process of DNN into more fine-grained components under the framework of FL and MEC.

3 Layers Decomposition

Different DNN models have a lot in common in their architectures: they are usually composed of multiple relatively independent layers with different functionalities. Furthermore, there exists a sequential relationship between the different layers, which provides the possibility to design a general DNN decomposing method: In the forward propagation process of DNN models, the L^{th} layer always receives the input from the $(\mathrm{L}-1)^{th}$ layer and gives the output to the $(\mathrm{L}+1)^{th}$ layer. In the process of backward propagation, the L^{th} layer receives the gradients from the $(\mathrm{L}+1)^{th}$ layer, calculates gradients of the weight, and returns the gradients of its input to the $(\mathrm{L}-1)^{th}$ layer.

The dependency relationship between adjacent layers must be considered when decomposing the training process of the DNN model. Each layer in the

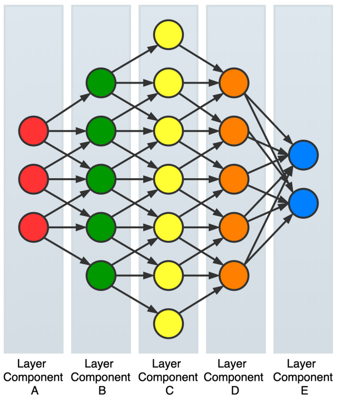


Fig. 1. Layer decomposition of DNN.

original DNN model can be decomposed as an independent component where the specific calculating procedure of the layer is implemented and encapsulated. Meantime, in the forward and backward propagation, the original layer-to-layer association can be achieved through mutual communication between devices. Besides, the intermediate results generated during the forward propagation process must be reasonably cached since these results will be used when performing the backward propagation.

In our work, in the local update process, we decompose the entire training procedure into different inter-independent coarse-grained layer components, as shown in Fig. 1. Each component is stateless and can be selected by instructions to execute a specific layer's forward or backward propagation procedure. Meanwhile, each component can be independently deployed on different devices and communicate with each other through the network protocol. The design of the layer component will be further discussed in detail in the next section.

4 System Design

4.1 Component-Based Program Design

The system is divided into two main components, which separately execute on the cloud and the client-side. The component executes on the client-side is responsible for the local updating and can be further decomposed into multiple more fine-grained components, which can be deployed either on the terminal or edge. As shown in Fig. 2, the components on the client generally interact with the terminalController component to cache intermediate data and exchange messages. Since the layer components have the possibility of migration at run time, there is no direct interaction between the layer components.

The entire procedure of FL can be decomposed into three sets of components: CloudController, TerminalController, and Layer Components.

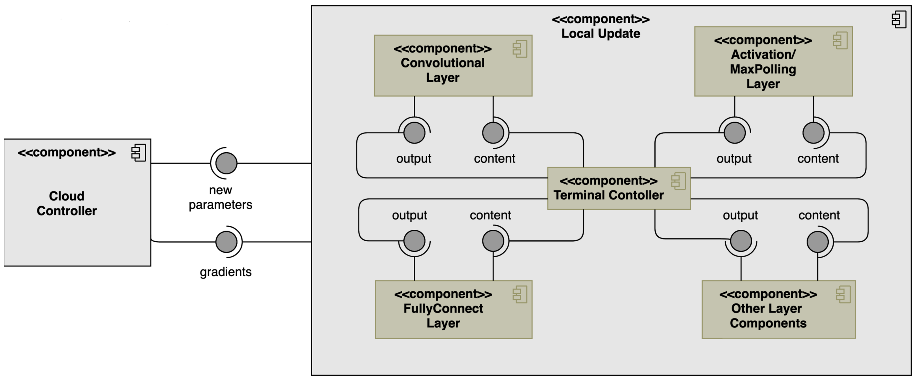


Fig. 2. Component diagram of the system.

CloudController Component. The cloudController component fixedly executes on the cloud and is responsible for the original cloud task, which includes initialization, parameter distribution, and gradient aggregation. When the initialization instruction is raised, the cloudController will perform cloud initialization and then send the initialization instruction to the terminalController component. When terminalController returns the gradient of weights, the cloudController will perform aggregation and distribute new parameters.

TerminalController Component. The terminalController fixedly executes on terminal devices. It is responsible for managing local privacy-sensitive datasets, caching intermediate results generated in the forward propagation of the DNN model, and determining which layer components to call next after the execution of the last layer component and gives the corresponding content. When the initialization instruction is raised, the terminalController will perform local initialization and inform the cloudController after its completion. In the forward propagation, when the intermediate output of the last layer component is sent back, the terminalController will cache the calculation results and redirect the control to the next layer component. In the backward propagation, it will store the sent back gradients from the last layer component and redirect the control.

Layer Components. Layer Components (Fig. 3) are a collective term for various layer components. Layer components in the general CNN model include and are not limited to convolutional layer components, fully connected layer components, activation function components, and pooling layer components. Since the communication cost and calculation cost of each layer is generally different, layer components should be established for each layer in a particular model. For example, if a CNN model is mainly composed of two convolutional layers and a fully connected layer, then it should consist of two independent convolutional

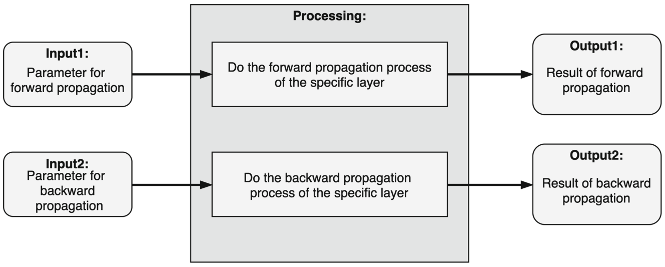

Processing:
Input1:
Parameter for forward propagation
Do the forward propagation process of the specific layer
Output1:
Result of forward propagation
Input2:
Parameter for backward propagation
Do the backward propagation process of the specific layer
Output2:
Result of backward propagation

Fig. 3. The IPO diagram of Layer Components.

layer components with different hyper-parameters and only one fully connected layer type component. These components are independent of each other and are uniformly called by the terminalController.

4.2 Data Flow

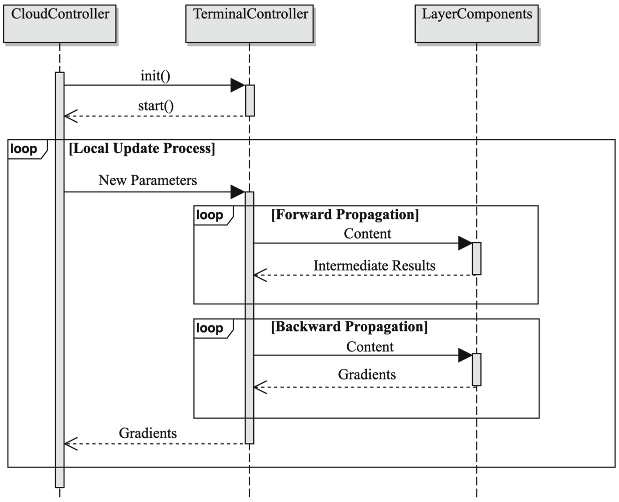


Fig. 4. The sequence diagram of the system.

Before the training process, the cloudController will issue initialization instructions to the terminalController. After the entire initialization is completed, in each round of training, the cloudController will send the current weight to the terminalController of each terminal. The local update process of each terminal can be divided into the forward propagation and the backward propagation.

In forward propagation, the terminalController sends content to each layer component accordingly. The state of the training process is passed through content, which also contains all the parameters needed for the forward propagation of this specific layer. After the forward propagation of this layer is completed, the layer component will return the intermediate results. The process of backward propagation is the same as that of the forward propagation. The difference is that the order of accessing the layer components is reversed, and the layer components return gradients of the weights to the terminalController. After the backward propagation is all over, the terminalController will return gradients of weights in this round of training to the cloudController. After cloudController aggregates the gradients returned by all terminals, it will update the existing weights and distribute them back to the terminalController to start the next iteration of training.

5 Privacy Preservation Analysis

Privacy preservation is one of the main features of FL. When using distributed methods for FL training, it is necessary to formulate specific component deployment strategies according to different situations to preserve user privacy.

First, the same as the general assumption of FL, it is assumed that the cloud is not trusted. The property of the cloud is that it is the publisher of training tasks and aggregator of gradients. The cloud knows the specific structure of the model, and after each mobile terminal device completes the local update process, the gradients generated will also be uploaded to the cloud. Recent works have proven that in the FL instances without any protection mechanism, algorithms like DLG [18] can be deployed on the cloud to restore user multimedia inputs at the pixel level. However, similar privacy risks can be avoided by gradient pruning [18] or formulating effective privacy protocols [1]. In our work, if the components execute on the cloud are separated from the other components, then the communication content existing between the former and the latter only includes the distribution of parameters and uploads of gradients, which is same as the communication content between the mobile terminal devices and the cloud in the traditional FL method. Therefore, the existing privacy preservation methods can still help eliminate the risk of privacy leakage on the cloud, so this is not the focus of our discussion.

Second, three different assumptions about the security situation of the edge device are made. The corresponding privacy preservation deployment strategies are discussed base on each assumption. Assumptions: (a)The edge is fully trusted (b)The edge is untrusted, but there is no direct communication or cooperation between the edge and the cloud (c)The edge is untrusted while there is direct communication or cooperative relationship between the edge and the cloud.

Case A. The edge device may be fully trusted in some cases; for example, the edge device is privately owned by an organization instead of being public. In this case, there is no additional privacy concern when doing the local update. The optimal strategy is which can maximize the overall efficiency.

Case B. The edge is untrusted, but it does not have direct communication or cooperation with the cloud. It means that there may exist potential attackers on the edge device, but the edge has different knowledge from the cloud: the edge does not have the specific knowledge about the entire structure of the DNN model. However, edge knows the input and the corresponding intermediate results and gradients of the layer components executing on it. In this way, it should be avoided to deploy the input layer component on edge, which will lead to direct leakage of user input. However, since the components are independent of each other and are stateless, layer components executing on edge will not be informed about the structure of the entire DNN model or the hyper-parameters, algorithms like DLG will not be able to obtain user privacy. Therefore, in this case, the optimal deployment strategy should be a strategy that maximizes the operating efficiency after fixing the input layer component on the mobile terminal.

Case C. The edge is not trusted, and there may exist direct communication or cooperation between the edge and the cloud, the potential attackers on edge and the cloud may share knowledge about the DNN model. In this case, directly giving the input of a specific layer component to the edge could be dangerous. Furthermore, we need to find new privacy preservation methods, such as designing new protocols, to ensure client privacy.

6 Experiments and Results

6.1 Experimental Settings

In our work, we decompose the FL process with classic CNN model LeNet [6] as the training model into two fixed components and seven free components (Fig. 5) as follow: Conv1 and Conv2 are layer components correspond to the convolutional layers in the order of the forward propagation, ReLu1 and ReLu2 are components for activation layers, Pool1 and Pool2 for pooling layers, FC for the remaining fully connected layers. The system is implemented by using the existing distributed software platform [2,3] bases on program decomposition. Base on the different deployment strategies in Table 1, we conducted a series of experiments and discussions on the efficiency of local training.

We deploy our system and conduct all experiments by constructing virtual machines on the existing OpenStack-based cloud platform. If there is no further instruction, the default experiment configuration is as follows: the operating system of the edge is Ubuntu 18.04, whose operating environment is Node.js 12.18.0, with 8 CPU cores and 32 GB RAM. The operating system of the terminal is Ubuntu 18.04, whose operating environment is Firefox 77.0.1, with 4 CPU cores and 16 GB RAM. The screenshots of the log output on the edge and terminal during the local training process are shown in Fig. 6.

Table 1. Different components deployment strategies.

Strategy	Components on terminal	Conponents on edge	Strategy	Components on terminal	Conponents on edge
A	No Component	All Components	B	Conv1	Conv2, ReLu1, ReLu2, Pool1, Pool2, FC
C	ReLu1, ReLu2	Conv1, Conv2, Pool1, Pool2, FC	D	Pool1, Pool2	Conv1, Conv2, ReLu1, ReLu2, FC
E	FC	Conv1, Conv2, ReLu1, ReLu2, Pool1, Pool2	F	Conv1, ReLu1, ReLu2	Conv2, Pool1, Pool2, FC
G	Conv1, Pool1, Pool2	Conv2, ReLu1, ReLu2, FC	H	Conv1, FC	Conv2, ReLu1, ReLu2, Pool1, Pool2
I	ReLu1, ReLu2, Pool1, Pool2,	Conv1, Conv2, FC	J	Conv1, ReLu1, ReLu2, Pool1, Pool2	Conv2, FC
K	ReLu1, ReLu2, Pool1, Pool2, FC	Conv1, Conv2	L	Conv1, ReLu1, ReLu2, Pool1, Pool2, FC	Conv2
M	All Components	No Component			

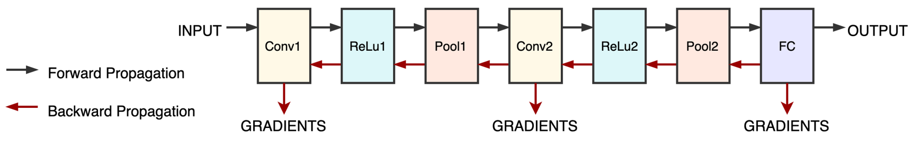
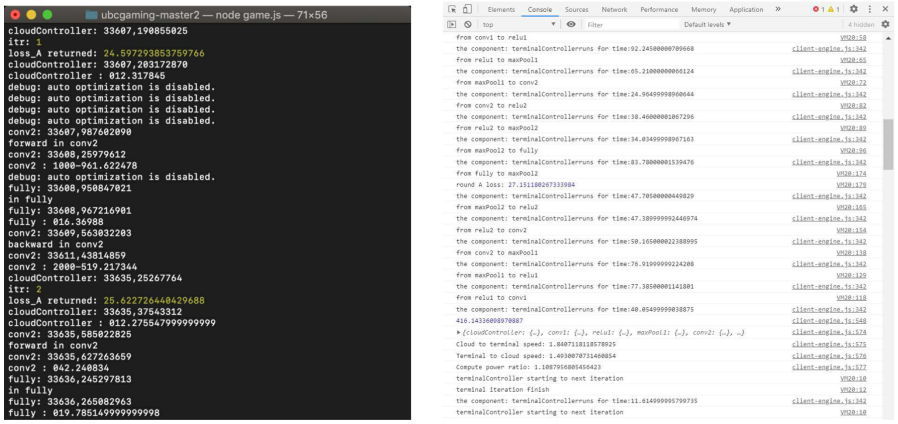


Fig. 5. LeNet architecture from a component perspective.

6.2 Run-Time Efficiency with Different Deployment Strategies

We observe the time cost of each iteration for the mobile terminal and the edge to complete the local update process through the distributed FL method under different deployment strategies in Table 1. It is defined that the time cost of each iteration starts when the new parameter distributed by the cloudController component, ends when the local update is completed and terminalController uploads the gradients to the cloudController. The batch size is set to 10 for each strategy. Figure 7 plots the execution curves of several representative deployment strategies in 150 iterations. The specific component deployment situation corresponding to each strategy is shown in Table 1. In Strategy M, all components are deployed on the terminal, which is similar to the traditional FL method. In Strategy A, all components are deployed on edge. The Strategy I and J are the optimal deployment strategies can be found under two different trusting assumptions (Case A and B) of edge devices, which can maximize the training efficiency.

As can be found from the figure, with same operating environment and hardware setting, The run-time efficiency of Strategy J and Strategy M is more unstable than Strategy A, and Strategy I. Especially in Strategy M, which is



```
ubcgaming-master2 — node game.js — 71×56
cloudController: 33607,190855025
itr: 1
loss_A returned: 24.597293853759766
cloudController: 33607,203172870
cloudController : 012.317845
debug: auto optimization is disabled.
debug: auto optimization is disabled.
debug: auto optimization is disabled.
debug: auto optimization is disabled.
conv2: 33607,987602090
forward in conv2
conv2: 33608,25979612
conv2 : 1000-961.622478
debug: auto optimization is disabled.
fully: 33608,950847021
in fully
fully: 33608,967216901
fully : 016.36988
conv2: 33609,563032203
backward in conv2
conv2: 33611,43814859
conv2 : 2000-519.217344
cloudController: 33635,25267764
itr: 2
loss_A returned: 25.622726440429688
cloudController: 33635,37543312
cloudController : 012.275547999999999
conv2: 33635,585022825
forward in conv2
conv2: 33635,627263659
conv2 : 042.240834
fully: 33636,245297813
in fully
fully: 33636,265082963
fully : 019.785149999999998
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  »   1  1
top  Filter  Default levels ▾   4 hidden
from conv1 to relu1                                                    VM20:58
the component: terminalControllerruns for time:92.24500000709668       client-engine.js:342
from relu1 to maxPool1                                                 VM20:65
the component: terminalControllerruns for time:65.21000000066124       client-engine.js:342
from maxPool1 to conv2                                                 VM20:72
the component: terminalControllerruns for time:24.96499998960644       client-engine.js:342
from conv2 to relu2                                                    VM20:82
the component: terminalControllerruns for time:38.46000001067296       client-engine.js:342
from relu2 to maxPool2                                                 VM20:89
the component: terminalControllerruns for time:34.03499998967163       client-engine.js:342
from maxPool2 to fully                                                 VM20:96
the component: terminalControllerruns for time:83.78000001539476       client-engine.js:342
from fully to maxPool2                                                 VM20:174
round A loss: 27.151180267333984                                       VM20:179
the component: terminalControllerruns for time:47.70500000449829       client-engine.js:342
from maxPool2 to relu2                                                 VM20:165
the component: terminalControllerruns for time:47.389999992446974      client-engine.js:342
from relu2 to conv2                                                    VM20:154
the component: terminalControllerruns for time:50.165000022388995      client-engine.js:342
from conv2 to maxPool1                                                 VM20:138
the component: terminalControllerruns for time:76.91999999224208       client-engine.js:342
from maxPool1 to relu1                                                 VM20:129
the component: terminalControllerruns for time:77.38500001141801       client-engine.js:342
from relu1 to conv1                                                    VM20:118
the component: terminalControllerruns for time:40.05499999038875       client-engine.js:342
416.14336098970887                                                     client-engine.js:548
▸{cloudController: {…}, conv1: {…}, relu1: {…}, maxPool1: {…}, conv2: {…}, …}   client-engine.js:574
Cloud to terminal speed: 1.8407118118578925                            client-engine.js:575
Terminal to cloud speed: 1.4930070731460854                            client-engine.js:576
Compute power ratio: 1.1087956805456423                                client-engine.js:577
terminalController starting to next iteration                          VM20:10
terminal iteration finish                                              VM20:12
the component: terminalControllerruns for time:11.614999995799735      client-engine.js:342
terminalController starting to next iteration                          VM20:10
```

Fig. 6. Screenshots of the log generate on both edge and terminal during the training process. The left figure is the console output of the Node.js server on edge. The right figure is the console output of the Chrome browser on the terminal.

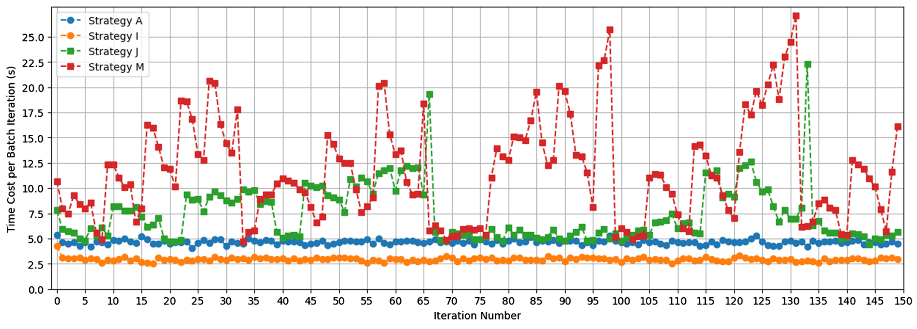


Fig. 7. Time cost of each iteration with different deployment strategies.

equivalent to the traditional FL deployment, the run-time efficiency of local training fluctuates greatly and periodically. This may be caused by the more stable execution in the Node.js environment at the edge compare to the browser engine at the terminal. Besides, it can also be seen that there exist significant differences in the average execution efficiency of different deployment strategies: compare with Strategy M, the average training efficiency of Strategy I and J increases by 74.8% and 36.5% respectively, which will be discussed in detail in the next section.

6.3 Average Efficiency with Different Deployment Strategies

Multiple sets of different component deployment strategies are tested with different batch sizes. In Fig. 8, the histogram shows the average time costs of each deployment strategy for an iteration with the batch size of 5, 10, and 20, which

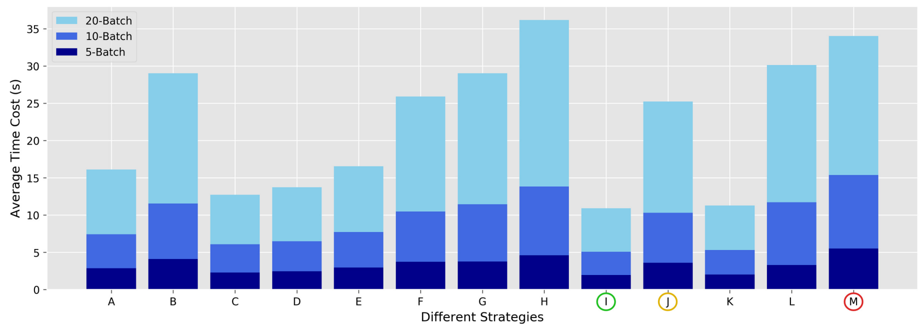


Fig. 8. Average time cost of different deployment strategies with different batch sizes. (Color figure online)

represented by dark blue, blue and light blue respectively. The specific deployment location of layer components in each strategy is shown in Table 1, we can see that there exists an apparent difference between different deployment strategies. Moreover, the difference widens with the increment of the batch size.

By observing the deployment properties of layer components in different strategies, the conclusion can be drawn. In the assumption where the edge device is completely trusted (Case A), the training efficiency is our only concern, so the optimal deployment strategy for all tested batch sizes is Strategy I (marked with green circle), which deploys the activation layer and polling layer components on the terminal and other layer components on edge. It is worth mentioning that some efficient compression techniques [14] can be further utilized to compress and decompress the original image at the terminal and edge, respectively.

When the edge is not trusted, and has no direct communication or cooperation with the cloud (Case B), the input layer locally must be kept on the terminal, so of all the strategies deploying Conv1 component on the terminal, Strategy J (marked with yellow circle) is the most efficient for all tested batch sizes. Strategy J deploys the input layer, activation layer, and pooling layer components on the terminal and other layer components on edge. The above two deployment strategies I and J are more efficient than the traditional federated learning deployment strategy M (marked with red circle). It can be inferred from the result that, since the communication costs of the layer components are relatively similar, when deploying some layer components with lightweight computation task (such as the activation layer component and the pooling layer component) on the terminal and deploy the computation-intensive layer components (such as convolutional layer component) on edge, the overall efficiency is higher.

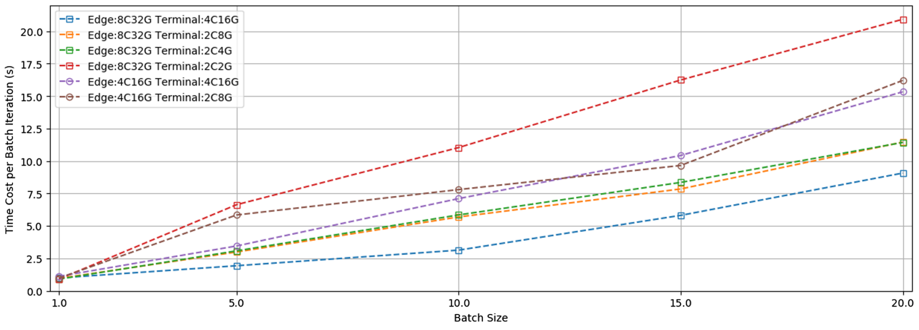


Fig. 9. Average time cost of a single iteration for different batch sizes when the hardware configuration of the edge and the terminal changes.

6.4 Effects of Different Hardware Resources on Average Training Efficiency

The effect of different hardware configurations on local training efficiency is also observed. In the following experiment, the layer components are deployed according to Strategy J, and the computing power resources of the edge and the terminal are adjusted. The average time cost of the batch iteration with different batch sizes is calculated, and results are shown in Fig. 9. When the batch size is equal to 1, with different computing power resources, the average training efficiency is basically the same. However, the difference in efficiency comes to be more significant with the gradual increment of the batch size. Among all the hardware configurations tested with the batch size of 20, the training efficiency comes to be the highest when the edge has 8 CPU cores and 32 GB RAM, and the terminal has 4 CPU cores and 16 GB RAM. On the contrary, the training efficiency comes to be the lowest when the edge has 4 cores with 16 GB RAM, and the terminal has 2 cores with 2 GB RAM. The training time cost of the latter is about 2.31 times the former.

It could be concluded that when the memory of the terminal device is below a certain threshold, which is about 2 GB in our experiment, the local training efficiency will be greatly reduced. Besides, changes in computing resources of edge devices will also significantly affect the local training efficiency. However, when the computing resources of the edge device remain unchanged, and the memory size of the terminal device is relatively sufficient, the computing resource of the terminal device has a limited effect on efficiency.

6.5 Effects of Different Terminal Operating Environments on Training Efficiency and Run-Time Stability

Different operating environments on the terminal device are tested to find their effects on the training efficiency further. We separately test the efficiency of the mobile terminal device using Chrome 83.0 browser and Firefox 77.0.1 browser

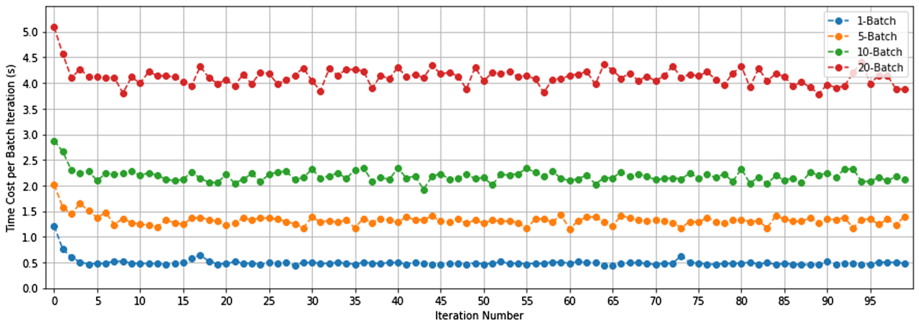


Fig. 10. Compare the time cost per iteration with different batch sizes when using the Chrome 83.0 browser in the Ubuntu 18.04 operating system as the terminal executing environment.

as the operating engine when the batch size is equal to 1, 5, 10 and 20 under the default hardware configuration with deployment strategy J. The results are shown in Fig. 10 and Fig. 11. It can be found that different terminal operating environments will make a noticeable difference in training efficiency and stability. In Fig. 10, when using Chrome as the browser, for all batch sizes, the time cost of each batch iteration is relatively stable. As the batch size increases, the fluctuation of the curve slightly increases but still not apparent. In Fig. 11, when using Firefox as the browser, it can be observed that for batch sizes like 1 and 5, the curve does not fluctuate significantly, but for larger batch sizes like 10 and 20, the curve fluctuates periodically and significantly, and there exist some unreasonably high peaks.

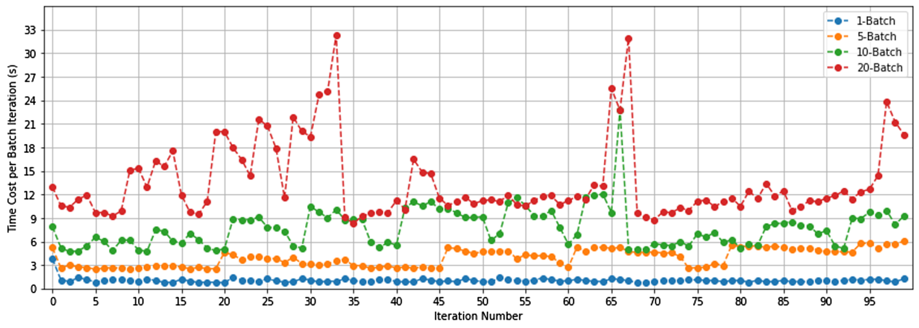


Fig. 11. Compare the time cost per iteration with different batch sizes when using the Firefox 77.0.1 browser in the Ubuntu 18.04 operating system as the terminal executing environment.

In fact, with the batch size increasing from 1 to 20, the standard deviation of the time cost of each iteration in the Chrome environment increases from 0.083

to 0.170, while for the Firefox, it increases from 0.318 to 4.884. Moreover, the latter's average time cost in different batch sizes is about 2.85 times the former. Since in some application scenarios of FL, the aggregation server may need to communicate with a large number of terminal devices at the same time in a single iteration of training, the global time cost of the specific iteration may depend more on terminals cost maximum time to complete the local training procedure instead of the average cost of all terminals. Therefore, the operating environment for the terminal should be reasonably chosen so that the local training of the terminal in each iteration is stable and has high average efficiency.

7 Conclusions

In this paper, we have proposed a component-based framework for FL, which can effectively improve the local training efficiency. Unlike the previous related works, we apply the coarse-grained program decomposition on the DNN model to allow the terminal to offload the training layers to the edge while preserving user privacy. An empirical study on a classic CNN is conducted to show the system's effectiveness and explore the optimal deployment strategies under different edge-trusting assumptions.

Several aspects that can be the future work: (1) apply program decomposition for more complex DNN models and find the corresponding optimal deployment strategies; (2) improve the system and provide API; (3) consider the impact of different network environments on training efficiency.

References

1. Bonawitz, K., et al.: Practical secure aggregation for privacy-preserving machine learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1175–1191, October 2017
2. Cai, W., Zhu, C., Chi, Y.C., Leung, V.C.: Balancing cloud and mobile terminal: an empirical study on decomposed cloud gaming. In: 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017), Hong Kong, China, 11 December–14 December 14 (2017)
3. Cai, W., Chi, Y., Zhou, C., Zhu, C., Leung, V.C.: UBCGaming: ubiquitous cloud gaming system. IEEE Syst. J. **12**(3), 2483–2494 (2018)
4. Harlap, A., et al.: Pipedream: fast and efficient pipeline parallel DNN training. arXiv preprint arXiv:1806.03377 (2018)
5. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
6. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
7. Lim, W.Y.B., et al.: Federated learning in mobile edge networks: a comprehensive survey. In: IEEE Communications Surveys & Tutorials (2020)
8. Mao, Y., Yi, S., Li, Q., Feng, J., Xu, F., Zhong, S.: A privacy-preserving deep learning approach for face recognition with edge computing. In: Proceedings USENIX Workshop Hot Topics Edge Computer (HotEdge), pp. 1–6 (2018)

9. Mao, Y., You, C., Zhang, J., Huang, K., Letaief, K.B.: A survey on mobile edge computing: the communication perspective. IEEE Commun. Surv. Tutorials **19**(4), 2322–2358 (2017)
10. McMahan, H.B., Moore, E., Ramage, D., Arcas, B.A.: Federated learning of deep networks using model averaging (2016)
11. Nishio, T., Yonetani, R.: Client selection for federated learning with heterogeneous resources in mobile edge. In: ICC 2019–2019 IEEE International Conference on Communications (ICC), pp. 1–7. IEEE, May 2019
12. Osia, S.A., et al.: A hybrid deep learning architecture for privacy-preserving mobile analytics. IEEE Internet of Things J. **7**(5), 4505–4518 (2020)
13. Qiu, H., Noura, H., Qiu, M., Ming, Z., Memmi, G.: A user-centric data protection method for cloud storage based on invertible DWT. IEEE Trans. Cloud Comput. **2020**, 1–12 (2019)
14. Qiu, H., Zheng, Q., Memmi, G., Lu, J., Qiu, M., Thuraisingham, B.: Deep residual learning based enhanced JPEG compression in the internet of things. IEEE Trans. Ind. Inform. **2020**, 1–10 (2020)
15. Wang, J., Zhang, J., Bao, W., Zhu, X., Cao, B., Yu, P.S.: Not just privacy: improving performance of private deep learning in mobile cloud. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2407–2416, July 2018
16. Wang, S., et al.: Adaptive federated learning in resource constrained edge computing systems. IEEE J. Selected Areas Commun. **37**(6), 1205–1221 (2019)
17. Yang, Q., Liu, Y., Chen, T., Tong, Y.: Federated machine learning: concept and applications. ACM Trans. Intell. Syst. Technol. (TIST) **10**(2), 1–19 (2019)
18. Zhu, L., Liu, Z., Han, S.: Deep leakage from gradients. In: Advances in Neural Information Processing Systems, pp. 14747–14756 (2019)