2018 IEEE 11th International Conference on Cloud Computing

Exploring Serverless Computing for Neural Network Training

Lang Feng*, Prabhakar Kudva†, Dilma Da Silva*, Jiang Hu*
*Texas A&M University, College Station
†IBM Research, Yorktown Heights
flwave@tamu.edu, kudva@us.ibm.com, dilma@cse.tamu.edu, jianghu@tamu.edu

***Abstract*—Serverless or functions as a service runtimes have shown significant benefits to efficiency and cost for event-driven cloud applications. Although serverless runtimes are limited to applications requiring lightweight computation and memory, such as machine learning prediction and inference, they have shown improvements on these applications beyond other cloud runtimes. Training deep learning can be both compute and memory intensive. We investigate the use of serverless runtimes while leveraging data parallelism for large models, show the challenges and limitations due to the tightly coupled nature of such models, and propose modifications to the underlying runtime implementations that would mitigate them. For hyper-parameter optimization of smaller deep learning models, we show that serverless runtimes can provide significant benefit.**

I. Introduction

As cloud computing increasingly becomes the platform of choice for commercial and scientific computing, serverless computing (also known as Functions as a Service or FaaS) [1], has emerged in recent years. With the increased use of containers and micorservices [2], serverless computing has shown particular promise for event-driven applications. Examples of commercial and open source serverless providers include: AWS Lambda [3], IBM OpenWhisk [4], Google Cloud Functions [5] and Microsoft Azure [6]. Serverless applications can be either a set functions as code, triggered by some external event [7], or a larger application composed of multiple functions. An example of composition of larger functions are those composed via AWS Step Functions [8]. A key reason for their success has been the cost efficiency that such runtimes provide in event-driven environments, where sporadic events may trigger computations, and the users only pay for the compute time they consume [3], rather than have long running servers implemented on virtual machines for these event processing, which will cause idle waiting for those sporadic events and result in unwanted monetary cost.

Key properties of serverless computing have been event-driven behavior, stateless, short run times, agile (it can be scaled up and down instantly and automatically [9]) and cost-efficiency. These properties benefit certain classes of applications better than others. Applications that are well suited for serverless are usually stateless, event-driven and short running. However with the composition of stateless services with persistence provided by database stores and loads between stateless serverless functions, the range of applications is gradually increasing. If a long running application is needed, it is possible to sequence multiple serverless computing instances in time with intermediate external storage [9]. Thus, the range of applications where serverless computing can show benefits can be extended beyond the event-driven domain.

Serverless computing has been applied to the area of machine learning [2, 5] with mixed results. The technology has been shown to be particularly useful for inference and prediction in cloud environments. For training models, especially deep learning models, which are compute and memory intensive and tightly coupled, serverless has not yet shown promise. The use of distributed computing for deep learning with accelerators is a well understood area [10, 11, 12, 13]. While solutions such as MxNet [14] and Distributed TensorFlow have increased the performance of distributed computing with GPU acceleration to speed up deep learning training, there are limited studies on the investigation of parallelism in serverless runtimes.

In order to facilitate the development of new serverless runtimes, and add features to the implementation backend (like compute and memory affinities based on cold and warm start) and others, it is important to understand strengths and limitations of deploying deep learning models in existing technologies, which our work mainly focuses on.

Our main contributions in this paper are:

- Proposing serverless computing structures for training large deep neural networks by leveraging data parallelism.
- Development of the parallel structure optimization for reducing training latency.
- Techniques for optimizing monetary cost and performance-cost ratio for training neural networks with serverless.
- Demonstrating the benefits of serverless for hyperparameter optimization of smaller models.
- Outlining novel serverless runtimes for further investigation to overcome the limitations for larger models.

The rest of this paper is organized as follows. Previous related works are briefly reviewed in Section II. In Section III, a multi-layer structure is introduced for neural network training on serverless instances and memory optimization

This work is partially supported by NSF (CCF-1525749).

techniques are further described. Section IV shows how to leverage serverless computing for hyperparameter tuning of neural network construction and training. Experiment results are summarized in Section V. New opportunities in serverless runtime design are discussed in Section VI and finally conclusions are provided in Section VII.

II. Related Work

Previous works on serverless runtimes mainly fall into two areas. The first is the area with more benefits on cost or performance over other runtimes like virtual machine or distributed computing. Examples of applications in this area are shown in [1], where tasks are mostly event-driven, stateless and have short run time.

The second is the area which aims at broadening the use of serverless runtimes, whereas most of the works in this area do not focus on machine learning, and there is no work towards deploying neural network training on serverless runtimes as our work. In this area, the works about scientific computing have been demonstrated successfully on serverless platforms. These efforts demonstrate the feasibility and promise of deploying scientific workload on serverless infrastructures. With Pywren [15] for example, one can deploy python-based workloads on multiple AWS Lambda services. The work of [16] proposed a performance evaluation framework with the use of a scientific workflow system: HyperFlow. The results show different behaviors between different serverless providers, such as AWS, Google, IBM and Microsoft. Another work, [17], mainly focuses on the deployment of scientific workflows on serverless environment, and proposes many feasible models for implementation. Those investigations are pioneer efforts on supercomputing with serverless architectures.

There are also works in the second area towards machine learning on serverless architectures, but the machine learning in these works has mostly been used for inference [5]. For example, in [18], the latency impact of the use of serverless for deep neural networks is investigated. The experiment in [18] shows the difference of the inference latency between the warm and cold execution, and the latency difference between different memory sizes. However, this work does not focus on deploying neural network for training, while our work proposes an optimized way for taking advantage of parallelism for training deep neural networks.

III. Training Large Models with Serverless

We define large models as those whose training cannot be completed within one serverless instance either due to constraints in runtime or memory requirements, such as the model used in TensorFlow Tutorials [19]. In this section we review cases that require workflows of several serverless instances with data transfer among them.

A. Data Transfer and Parallelism with Serverless

A key difference in the application of parallelism is the different nature between serverless instances and normal distributed computing: serverless instances are inherently time-limited and stateless, unlike parallel threads previously studied with deep neural networks. At present, there is no way to transfer data between two serverless instances directly, or to assign serverless instances affinity to compute resources close to the shared data. Therefore, intermediate storage such as databases are used for holding states that are to be shared between subsequent serverless instances. The data transfer between instances is shown in Figure 1. Since two serverless instances cannot communicate directly, the parallelized data or models have additional costs, including:

- data transfer latency from source instance to database,
- data transfer latency from database to destination instance,
- warm up latency for loading data.

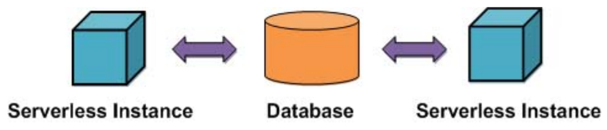

Figure 1. Data transfer gateway between two serverless instances

Distributed deep learning platforms have been well studied over the years [13], with more recent platforms such as distributed TensorFlow [20] and MxNet [14], as well as variants of them [21] showing dramatic improvements on performance and scalability. Significant improvements are noted when deep learning models take advantage of parallelism. For this work, we adapt such well known approaches to training neural networks in a distributed fashion and investigate their suitability to serverless. To coordinate the component instances in a training workflow, we use the graph-based notation of step functions [8]. In the rest of this paper, the words *graph* and *structure* will refer to the interconnected structure of serverless instances interleaved by writes and reads to storage.

B. Data Parallelism for Neural Network Training

Given a dataset, the training of a neural network is to iteratively modify network parameters, such as edge weights and biases, such that the network inference results match the dataset. Many common training algorithms, such as stochastic gradient [22], compute gradients according to the training data and then the gradients are applied to update the parameters. In data parallelism, a given dataset is partitioned into multiple subsets, each of which is applied to train a complete network model on a machine, called *worker*. All workers share the same network model. When a subset of training data is applied to a worker, corresponding gradients are computed there. Then, gradients from all workers need to be collected by a machine, called *parameter server*, where the network parameters are updated. The data parallelism by serverless computing is illustrated in Figure 2, where each gray rectangle indicates one serverless instance. In serverless environments, all the machines are serverless instances. The dataset is partitioned into n workers, which compute gradients and send the gradients to the parameter server.

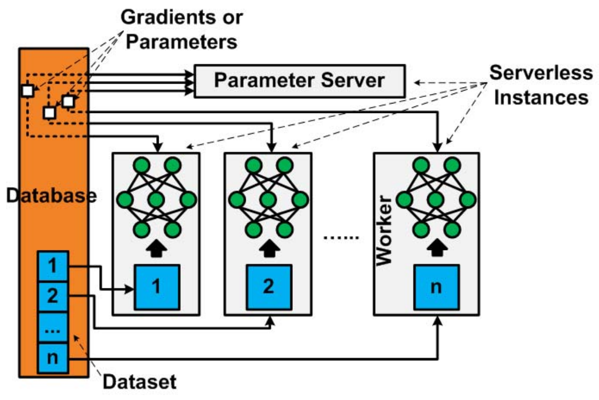

Figure 2. Data parallelism by serverless computing

There are two approaches for updating parameters in the data parallelism: Synchronous and asynchronous update.

In a synchronous update, the parameter server waits till gradients from all workers are received and then updates the parameters. In an asynchronous update, the parameter server updates the parameters each time it receives one set of gradients from one worker. In this work, we focus on the synchronous update.

C. *Optimizing Parallelism Structure for Serverless Training*

For data parallelism, data transfers occur between workers and the parameter server. There are two kinds of transfers:

- The parameter server transfers parameters to all workers.
- Workers transfer gradients to the parameter server.

For the transfer from the parameter server to workers, the latency is a constant as the number of parameters is fixed for a given neural network model and all workers can read the same parameters in parallel.

For the other transfer type, if there are n workers, the parameter server needs to receive n sets of gradients, each of which corresponds to one set of parameters on one specific worker. If it takes time t_m for each set of gradients to be transferred to the parameter server, the latency of transferring all gradients in one iteration is $n \cdot t_m$. The linear dependence on the number of gradient sets is confirmed by the measurement results shown in Figure 3, where the bars indicate plus/minus standard deviation.

Since the data transfer is the main performance bottleneck for serverless training of neural networks, we propose a multi-layer parameter server structure to reduce the transfer

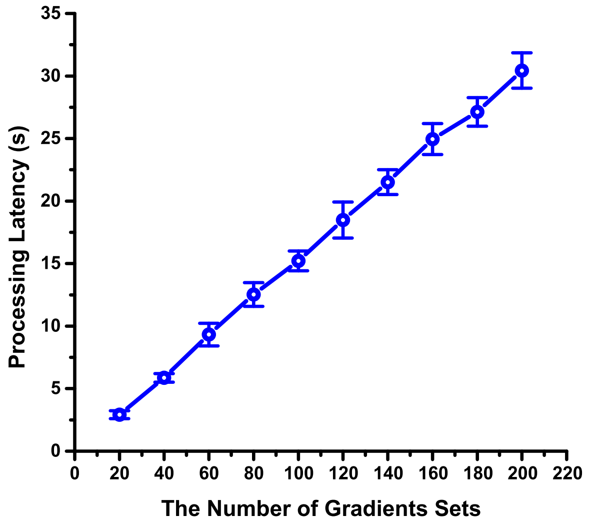

Figure 3. The relationship between data transfer latency and the number of gradients sets received by one parameter server. (experiment environment: the transferred data contains 42601 gradients, and parameter server is a 512MB serverless instance)

latency. Please note that the focus here is to reduce the latency of transferring gradients, as the latency of transferring parameters is constant. In Figure 4, we use blue nodes to represent workers and yellow nodes to indicate parameter servers. Usually, multiple workers send their gradients to the parameter server as in Figure 4(a). The parameter server is also called merging node in the graphs in Figure 4.

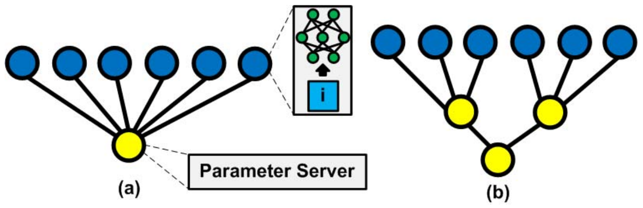

Figure 4. Different structures for merging gradients by parameter servers.

We propose a multi-layer merging structure as in Figure 4(b), which contains 6 workers and 3 parameter servers distributed in 2 layers. In such structure, a parameter server or merging node can receive gradient data from other merging nodes. In the upper merging layer of Figure 4(b), each parameter server merges gradients from 3 workers and the merging takes $3t_m$. Since the two merging nodes work in parallel, the merging latency of this layer is also $3t_m$. In the lower merging layer, there is one parameter server, which merges gradients from the two parameter servers of upper layer and the merging latency is $2t_m$. Therefore, the total gradient data transfer latency in Figure 4(b) is $5t_m$. By contrast, the naïve merging structure in Figure 4(a) costs $6t_m$ transfer latency.

The example of Figure 4 indicates that the proposed multi-layer merging structure can reduce merging (gradient data transfer) latency. A general problem is how to decide the number of merging layers and number of parameter severs in each merging layer such that the gradient data transfer latency is minimized. To solve this problem, we first introduce the latency model for transferring gradients by workers to the parameter servers.

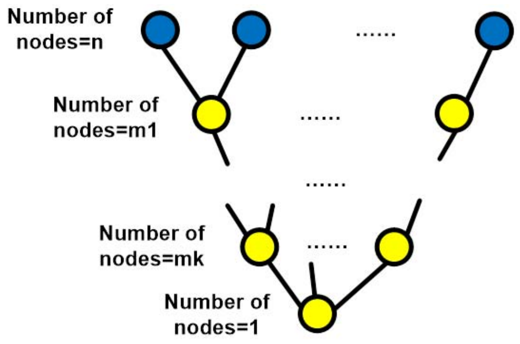

Figure 5. General structure of parameter servers

Consider a general data parallel structure as Figure 5. Assume there are n workers and k intermediate merging layers, and for the i_{th} merging layer, there are m_i parameter servers. For the bottom merging layer, there is only 1 parameter server to do the final data processing. Then, the total latency can be described as

$$t = t_m\frac{n}{m_1} + t_m\frac{m_1}{m_2} + ...t_m\frac{m_{k-1}}{m_k} + t_m m_k \tag{1}$$

To minimize t, we first take partial derivatives with respect to each m_i as below.

$$\begin{cases} \frac{\partial t}{\partial m_1} = -\frac{n}{m_1^2}t_m + \frac{1}{m_2}t_m \\ \frac{\partial t}{\partial m_i} = -\frac{m_{i-1}}{m_i^2}t_m + \frac{1}{m_{i+1}}t_m \quad 1 < i < k \\ \frac{\partial t}{\partial m_k} = -\frac{m_{k-1}}{m_k^2}t_m + t_m \end{cases} \tag{2}$$

One can tell that the second order derivatives are all positive, then the function t versus m_i is convex. By letting all first order derivatives be 0, the values of all m_i minimizing t are given by

$$\begin{cases} m_k^{k+1} = n \\ m_i = m_k^{k-i+1} \quad 1 \le i \le k-1 \end{cases} \tag{3}$$

Thus, the minimum t is found to be

$$t_{min} = t_m(k+1)n^{\frac{1}{k+1}} \tag{4}$$

We find the k that minimizes t_{min} by letting $\frac{dt_{min}}{dk} = 0$, which gives $k = ln(n) - 1$. However, a large k means many hops of data transfer, which increase the chance of packet loss and the costly data retransmission. Therefore, we bound the value of k to be no greater than 2 in practice.

Since creating new serverless instances is associated with latency overhead, the actual data transfer latency can be further reduced by reusing worker serverless instances as parameter servers as shown by Figure 6. After completing their neural network training work, workers 3 and 6 continue to collect gradient data from 1, 2, 3, and 4, 5, 6, respectively. In other words, worker 3 (6) plays the role of parameter server 7 (8) now. After merging data from workers 4, 5, and 6, node 6 further collects the gradient data from parameter server 3, and is actually doing the merging formerly done by node 9. Please note such reuse is possible only for the synchronous data update.

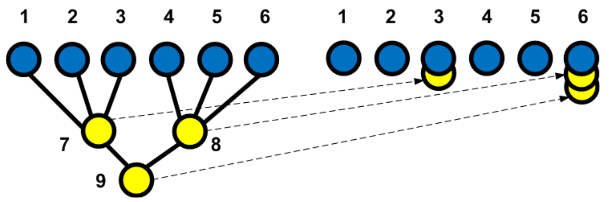

Figure 6. Reuse worker serverless instances as parameter servers.

D. Cost and Performance-Cost Ratio Optimization

A key motivation for serverless computing is its economic advantage over the conventional cloud services. Hence, we study how to minimize monetary cost and maximize performance-cost ratio in using serverless computing. In contrast to the offline structure optimization, the cost and performance-cost ratio optimizations are online techniques.

Table I
AWS LAMBDA PRICE VS. MEMORY USE.

Memory (MB)	Price per 100ms ($)
128	0.000000208
192	0.000000313
256	0.000000417
...	...
1408	0.000002292
1472	0.000002396
1536	0.000002501

We derive a model of monetary cost with respect to the memory allocated to a serverless instance. Please note memory size z of a serverless instance should be no less than the minimum memory required to run the application there. In addition, a large memory size z implies shorter latency [23]. Hence, latency is a function of memory size as $t(z)$. For AWS Lambda [3], the monetary price per unit runtime for different memory sizes [24] is shown in Table I. By such pricing, the monetary cost has linear dependence on memory size of the Lambda instance and latency, and can be defined as

$$C(z) = p \cdot n \cdot z \cdot t(z), \tag{5}$$

where $p = 1.63 \times 10^{-8}\$/(MB \cdot s)$ and n is the number of Lambda instances.

We propose an online gradient descent method for finding memory size z for each serverless instance such that the monetary cost $C(z)$ is minimized. The online optimization starts with a random memory size z_1, which is sufficiently large for the neural network training and satisfies serverless instance specification. The training with z_1 memory is continued with q iterations and the average cost $\bar{C}(z_1)$ over the q iterations is estimated according to Equation (5). Then, memory size is changed to another random and feasible value z_2 for another q iterations of training to obtain an average estimation $\bar{C}(z_2)$. After the sampling of two random sizes, we find an optimized memory size as

$$z_3^* = z_1 - \alpha \frac{\bar{C}(z_1) - \bar{C}(z_2)}{z_1 - z_2} \tag{6}$$

where α is the step size for the gradient decent. Since there are lower bound z_{min} and upper bound z_{max} for the actual memory size due to serverless instance restrictions and application requirement, the actual memory size to be used next is

$$z_3 = \max(z_{min}, \min(z_{max}, z_3^*)) \tag{7}$$

and this procedure can be repeated such that the memory size is continuously optimized.

We propose another online gradient decent method for maximizing performance-cost ratio. Given a computing task that requires f floating point operations, the performance can be characterized by FLOPS (floating point operations per second), which can be estimated by $\frac{f}{t(z)}$. Then, the performance-cost ratio is defined by

$$R(z) = \frac{f}{p \cdot n \cdot z \cdot t^2(z)} \tag{8}$$

which is a function depending on memory size z. Like minimizing the cost, one can sample a size for q iterations. If the two consecutive sample sizes are z_{j-1} and z_j, then the optimized memory size can be obtained as

$$z_{j+1} = \max(z_{min}, \min(z_{max}, z_j + \alpha \frac{\bar{R}(z_j) - \bar{R}(z_{j-1})}{z_j - z_{j-1}})) \tag{9}$$

where $\bar{R}$ indicates the average ratio over q iterations.

IV. Parallel Hyperparameter Tuning of Neural Network Models with Serverless

The effectiveness of a neural network model and its training efficiency highly depend on hyperparameters, such as the number of hidden layers, activation function and training rate. The hyperparameters can be decided either manually or through automated search such as random search, grid search and Bayesian optimization [25].

Since the evaluations of different hyperparameters can be independently carried out, serverless computing is a particularly appealing choice for the tuning. Suppose $H = \{h_1, h_2, ...\}$ is a set of hyperparameters for a specific neural network model. All sets hyperparameters to be explored are $\mathcal{H} = \{H_1, H_2, ...\}$. One can request n_i serverless instances for training the model specified by $H_i \in \mathcal{H}$. Since the total number of serverless instances one can request is bounded by N. We need to make sure that

$$\sum_{i=1}^{|\mathcal{H}|} n_i \leq N. \tag{10}$$

Due to this restriction, severless hyperparameter tuning is mostly for small network models.

V. Experiment Results

A. Experiment Setup

These experiments are conducted on a randomly generated dataset, CIFAR-10 dataset [26] and MNIST dataset [27]. The random dataset contains 1 million samples, each of which is composed by 20 binary features and 1 binary label. The random dataset is applied with a fully connected neural network with 5 hidden layers, 500 hidden nodes and 42601 parameters. The CIFAR-10 dataset is to be trained by a convolution neural network, which has 2 convolution layers, 2 pooling layers, 2 normalization layers, 2 fully connected layers and 1 softmax output layer. This structure is the same as the structure used in the code of TensorFlow Tutorials [19]. The model for MNIST is a fully connected neural network, whose structure is investigated through the hyperparameter tuning. The characteristics of the 3 testcases are summarized in Table II. The training of using the datasets on the models is by TensorFlow [28]. The serverless computing experiments are conducted through AWS Lambda [3], where latency, memory use and monetary cost are measured. The training experiment is also performed on a desktop computer with a Intel 3.4GHz CPU with 16GB memory.

Table II
Testcases

	Case A	Case B	Case C
Dataset	Random dataset	CIFAR-10 [26]	MNIST [27]
Network type	Fully connected neural network	Convolution neural network	Fully connected neural network
Network structure	5 hidden layers, 500 hidden nodes and 42601 parameters.	Same as in the code of TensorFlow Tutorial [19].	Structure investigated through the hyperparameter tuning.

B. Latency Variation

When evaluating serverless computing latency, one faces the challenge of its variations. Serverless runtimes are instantiated on infrastructure via resource scheduling by the service provider in a manner invisible to the end user. Similarly, the location and the latency response of database for reads and writes may vary depending on the resource allocation on the cloud provider side. There are no guarantees on latency and performance of such serverless instantiations beyond the requested parameters such as memory size (which are priced). Likewise, the read and write latency between serverless instance and database may vary depending on a variety of factors, like the actual location of the database relative to the instance, traffic on networks, multi-tenancy, to name a few. The end-user does not have control on these latencies and performance metric, and expectations are that they vary within a certain known range (based on the provider) from a statistical perspective. *Therefore, all latency and performance measurements reported in the paper are representative, and a few percent variation or improvement is considered normal statistical variation.*

C. Structure Optimization

This part of experiment is to evaluate the effectiveness of the proposed multi-layer merging structure and its optimization, which are introduced in Section III-C. It is performed on Case A and Case B. For Case A, the number of training iterations is 50. The training is done by 100 workers, each of which has 512MB memory. For Case B, the number of training iterations is 20. The training is done by 100 workers, each of which has 1536MB memory. The results are summarized in Table III. Each structure is indicated by a vector, where each element specifies the number of nodes in a layer and the elements are in bottom-up order of the tree structure depicted in Figure 5. For example, $[1, 5, 100]$ means the gradients from 100 workers are transferred to 5 parameter servers, and finally merged at a single parameter server. The result in the first row, which is labeled with '*', is the optimal solution according to our optimization. For Case A, One can see this is the second to the minimal latency result according to the measurement. Its actual latency 569.55 is close to the minimal latency 534.28. The discrepancy between our optimal solution and the actual minimal is due to the latency variation, which is discussed in Section V-B. For Case B, our optimal solution has the least latency and therefore the effectiveness of our optimization is confirmed. One should also note that according to our discussion in Section III-C, the optimal solution does not depend on the neural network model used but only depends on the number of workers.

Table III
LATENCY OF DIFFERENT STRUCTURES

Structure	Latency for Case A (s)	Latency for Case B (s)
*[1,5,22,100]	569.55	789.20
[1,100]	1216.97	1848.05
[1,2,100]	878.78	1195.97
[1,5,100]	650.66	866.29
[1,25,100]	616.67	920.22
[1,2,10,100]	570.66	823.25
[1,5,50,100]	604.90	890.91
[1,10,50,100]	534.28	838.89
[1,2,10,50,100]	585.25	883.96
[1,5,20,50,100]	578.22	868.78

D. Training Accuracy and Convergence Rate

We evaluate the training accuracy and convergence rate of the proposed serverless computing and sequential computing on desktop PC on Case A and Case B. The serverless structures used here are the same as in Section V-C. The accuracy of a neural network is estimated by comparing its inference results on training dataset labels. The accuracy versus training time results for Case A are shown in Figure 7. The serverless computing converges slower than desktop PC, but reaches a better accuracy. The results for Case B are plotted in Figure 8, where the serverless computing leads to worse accuracy and convergence rate than the desktop PC.

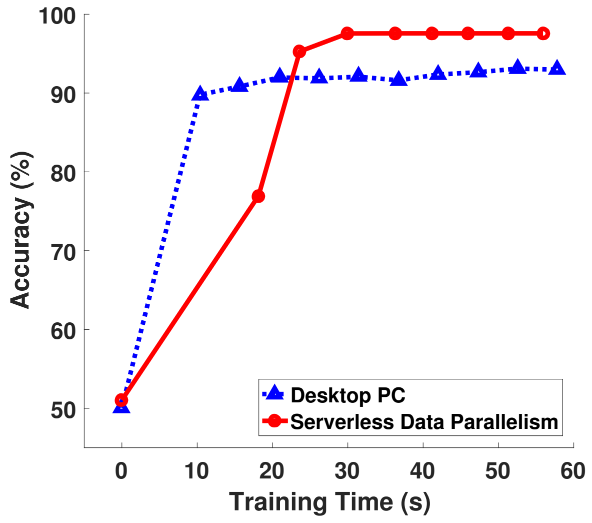

Figure 7. Training accuracy vs. training time for Case A.

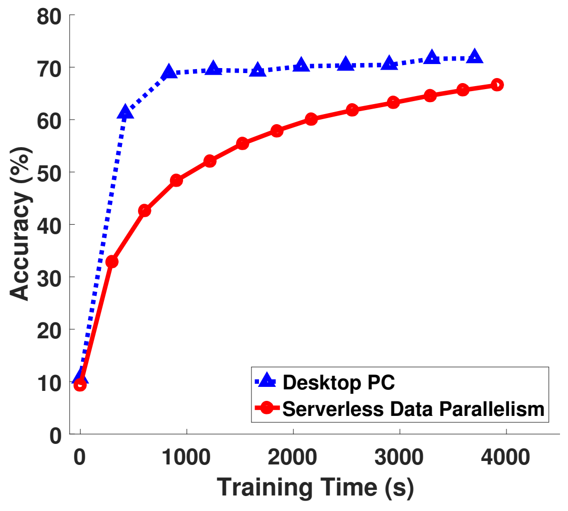

Figure 8. Training accuracy vs. training time for Case B.

The accuracy difference between the desktop PC and serverless results arises from the parameter update difference between sequential and parallel training. In serverless computing, the parameter update is based on the average of gradients obtained from multiple workers. In the sequential training on desktop PC, by contrast, each parameter update is according to a single set of gradients from a single process.

E. Result of Cost and Performance-Cost Ratio Optimization

The proposed online cost minimization method is evaluated on Case A. In this experiment, $q = 10$, which means we modify the memory size every 10 iterations. In addition, the gradient decent is performed at most five times. The lower and upper bounds of memory size are set as $z_{min} = 256MB$ and $z_{max} = 1536MB$, respectively. The results are shown in Figure 9, where the red circles indicate our optimization results. The experiment is repeated 10 times. Due to the latency $t(z)$ variations, two different results (red circles) are obtained. For 9 times, the optimization result is $256MB$ and $512MB$ is obtained once. The blue triangles and bars are the measurement results of monetary cost at different memory sizes without optimization. For each memory size, the experiment is repeated 100 times. Each blue triangle represents the average cost and the bars indicate $\pm\sigma$, which is the standard deviation. One can see that the cost variation can be very large due to the latency uncertainty. Moreover, the cost vs. z change is not monotone. The average cost of $640MB$ is less than that for $512MB$ memory. Most importantly, our optimization indeed reaches the minimum or near minimum cost memory size.

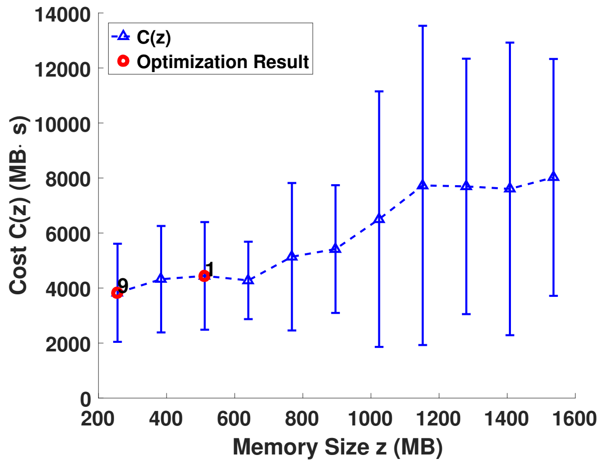

Figure 9. Cost per iteration under different Lambda instance memory sizes and optimization results.

The performance-cost ratio optimization results are plotted in Figure 10. Here, we attempt to maximize the ratio. Indeed, the red circle results from our optimization are generally at memory sizes where the ratio is at least near the maximum.

F. Results on Hyperparameter Tuning

The experiment on hyperparameter tuning is performed on Case C. In Figure 11, the computing latency results versus the number of searched hyperparameter sets $|\mathcal{H}|$

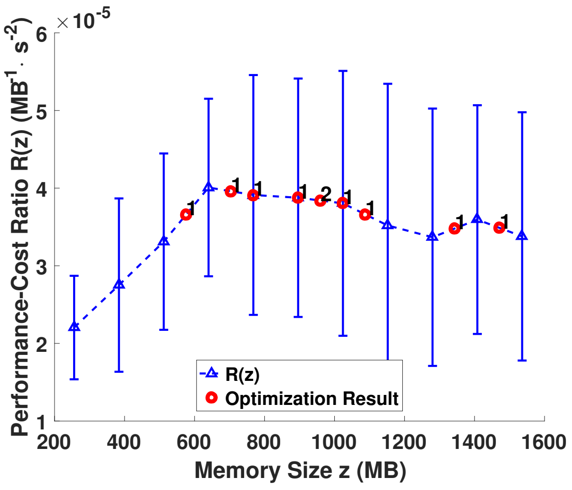

Figure 10. Performance-cost ratio per iteration under different Lambda instance memory sizes and optimization results.

for desktop PC and AWS Lambda are plotted. Each dot in the figure is the average of 10 different experiments with the same number of searched hyperparameter sets. One can see that the latency of desktop PC grows linearly when more hyperparameters are evaluated because of its sequential computing nature. The hyperparameter tuning on AWS Lambda is carried out in parallel. Thus, its latency does not change when the hyperparameter search is expanded. This result clearly demonstrates the advantage of serverless computing for hyperparameter tuning in neural network model construction and training.

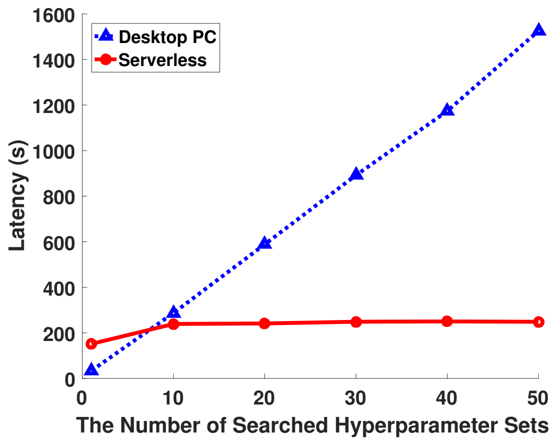

Figure 11. Computing latency versus the number of searched hyperparameter sets $|\mathcal{H}|$.

VI. Opportunities in Serverless Runtime Design

Serverless runtimes have been used for inference with good results. Our exploration of using serverless for training large deep learning models has identified some disadvantages compared to other distributed computing runtimes where data transfer between compute instances are not as frequent (such as with GPUs). In order to improve serverless performance for the task of training deep learning models, it is necessary to minimize the frequency and quantity of data transfer between subsequent serverless instances. We illustrate opportunities for improved data transfer latencies, while at the same time maintaining the benefits of serverless such as the ability to pay for a compute instance used only when and just long enough for needed computation.

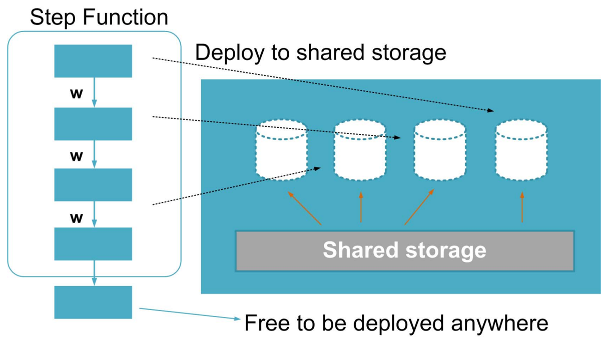

Figure 12. Serverless affinity in runtimes

Consider a step function where edges between serverless instances can be assigned higher affinities indicating sharing of data between them, then the instances can be mapped by the infrastructure manager in a manner where the latencies for data transfer between the instances is minimized. In Figure 12, a generic approach to such a solution is given, where a portion of the step function has some serverless instances with weight w on the edges indicating shared data, while the last instance has no weight assigned, indicating no such affinity. Such a specification can be mapped in several ways as described below:

- Given affinities between serverless instances in a step function, the infrastructure maps these instances to a common host where storage is persistent across serverless instance invocations. In a runtime where each serverless instance is implemented as a Linux container, the storage on the host is mounted onto the container during boot up, thus enabling sharing of data.
- Implementations based on processor in memory (PIM) may also be considered for this purpose. A processor associated with a memory device such as a Linux on ARM associated with SSD or Memory, can also be used to support the affinity for especially large models.

VII. Conclusion

Training deep learning models with serverless runtimes is challenging and provides several opportunities. We have investigated both large and small models. For large models, various structures for composition of serverless instances to provide the best performance and cost to train deep learning models, while taking advantage of data parallelism were explored. The challenges posed by the ephemeral, stateless and warm up latency of serverless runtimes were studied. Potential innovations in runtime design for future serverless runtimes with containers were proposed to mitigate the challenges and strengthen the opportunities. For smaller models, it was shown that serverless runtimes showed benefit for hyperparameter tuning that could be performed in a truly distributed manner.

References

[1] Awesome serverless Git. https://github.com/anaibol/awesome-serverless.
[2] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *Computing Research Repository*, 2017.
[3] AWS Lambda. https://aws.amazon.com/lambda/.
[4] S. Fink, "OpenWhisk." https://developer.ibm.com/open/wp-content/uploads/sites/50/2016/06/OpenWhisk-Charts.pdf.
[5] Google Cloud, "Building a Serverless ML Model." https://cloud.google.com/solutions/building-a-serverless-ml-model.
[6] Microsoft, "Microsoft Azure." https://azure.microsoft.com/en-us/.
[7] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops*, pp. 405–410, June 2017.
[8] AWS, "AWS Step Functions." https://aws.amazon.com/step-functions/.
[9] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service(faas) in industry and research," *Computing Research Repository*, 2017.
[10] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized stochastic gradient descent," in *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, pp. 2595–2603, 2010.
[11] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24*, pp. 693–701, 2011.
[12] J. Keuper and F.-J. Pfreundt, "Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, pp. 1:1–1:11, 2015.
[13] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, pp. 1223–1231, 2012.
[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2016.
[15] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," *Computing Research Repository*, 2017.
[16] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *Euro-Par 2017: Parallel Processing Workshops* (D. B. Heras and L. Bougé, eds.), pp. 415–426, 2018.
[17] M. Malawski, "Towards serverless execution of scientific workflows - hyperflow case study," in *WORKS@SC*, November 2016.
[18] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," *Computing Research Repository*, 2017.
[19] TensorFlow Tutorials: Convolutional Neural Networks. https://www.tensorflow.org/tutorials/deep_cnn.
[20] Distributed TensorFlow. https://www.tensorflow.org/deploy/distributed.
[21] J. Yang, Y. Chen, S. Wang, L. Li, C. Meng, M. Qiu, and W. Chu, "Practical lessons of distributed deep learning," 2017.
[22] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, pp. 400–407, 1951.
[23] Configuring Lambda Functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html.
[24] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.
[25] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, pp. 2951–2959, 2012.
[26] CIFAR-10 Dataset. https://www.cs.toronto.edu/~kriz/cifar.html.
[27] The MNIST Database. http://yann.lecun.com/exdb/mnist/.
[28] TensorFlow. https://www.tensorflow.org/.