AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency

Jananie Jarachanthan[1], Li Chen[1], Fei Xu[2], Bo Li[3]
[1]School of Computing and Informatics, University of Louisiana at Lafayette
[2]School of Computer Science and Technology, East China Normal University
[3]Department of Computer Science and Engineering, Hong Kong University of Science and Technology
[1]{jananie.jarachanthan1, li.chen}@louisiana.edu, [2]fxu@cs.ecnu.edu.cn, [3]bli@cse.ust.hk

ABSTRACT

The salient pay-per-use nature of serverless computing has driven its continuous penetration as an alternative computing paradigm for various workloads. Yet, challenges arise and remain open when shifting machine learning workloads to the serverless environment. Specifically, the restriction on the deployment size over serverless platforms combining with the complexity of neural network models makes it difficult to deploy large models in a single serverless function. In this paper, we aim to fully exploit the advantages of the serverless computing paradigm for machine learning workloads targeting at mitigating management and overall cost while meeting the response-time Service Level Objective (SLO). We design and implement *AMPS-Inf*, an autonomous framework customized for model inferencing in serverless computing. Driven by the cost-efficiency and timely-response, our proposed *AMPS-Inf* automatically generates the optimal execution and resource provisioning plans for inference workloads. The core of *AMPS-Inf* relies on the formulation and solution of a Mixed-Integer Quadratic Programming problem for model partitioning and resource provisioning with the objective of minimizing cost without violating response time SLO. We deploy *AMPS-Inf* on the AWS Lambda platform, evaluate with the state-of-the-art pre-trained models in Keras including ResNet50, Inception-V3 and Xception, and compare with Amazon SageMaker and three baselines. Experimental results demonstrate that *AMPS-Inf* achieves up to 98% cost saving without degrading response time performance.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

serverless computing, machine learning inference, cost efficiency

ACM Reference Format:
Jananie Jarachanthan[1], Li Chen[1], Fei Xu[2], Bo Li[3]. 2021. *AMPS-Inf*: Automatic Model Partitioning for Serverless Inference with Cost Efficiency. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472501

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472501

1 INTRODUCTION

Serverless computing becomes increasingly popular due to its auto-scaling and pay-per-use natures. This provides ease-of-management and cost-efficiency, which various workloads can take advantage of. With serverless architectures deployed by cloud providers such as Amazon Lambda [14], Google Cloud Functions [19], and Microsoft Azure Functions [18], a wide variety of applications are shifting towards this computing paradigm, including real-time video encoding [33], data analytics [37, 39, 40, 45], web applications [43, 49], and *etc.*

Given the growing adoption of machine learning across different fields in today's era of Internet-of-Things and Artificial Intelligence, it is natural to consider whether and how machine learning applications can exploit serverless computing. There are a number of challenges when migrating machine learning workloads on serverless architecture mainly due to the restrictions of serverless function size and the increasing size of neural network models. For example, the deployment package size limit of a function on Amazon Lambda is 250MB [15] while the size of a model can be as large as 500MB, such as VGG16 [5] and VGG19 [6]. Although it is likely that cloud providers may increase the deployment size limit in the future, there is a faster growth of the size and complexity of advanced neural network models (such as BERT [31]), which still raises the similar challenges to be elaborated as follows.

For machine learning services, in particular the inference, to be deployed on the serverless platform, one of the main requirements is the minimization of the billing cost without violating a pre-defined Service Level Objective or SLO in term of query response time [50]. This specifies the time it takes for prediction results of the input to be returned to users. When a neural network model is too large to fit into a serverless function, it encounters a number of difficulties including how to split the complex computation graph, how to coordinate the partitions, and which function resource type to specify for each partition? Due to the complex structures of today's neural network models, it is not clear how a specific partition, among a gigantic number of possible ones, impacts the coordination efforts, the end-to-end query response time and the total billing cost. In addition, the space for resource configuration (*e.g.*, the function memory size for each partition) which also impacts the response time and monetary cost, is enormous. Given such a large decision space, domain expertise and nontrivial efforts are expected from users to satisfy their requirements on meeting response time SLOs and minimizing cost, which compromise the ease-of-management

feature and may discourage them from migrating to serverless platforms.

To cope with such complexities, we present our framework, *AMPS-Inf*, which automatically deploys machine learning inferencing workloads on serverless platform towards both *cost-efficiency* (*i.e.*, minimizing monetary cost) and *timely response* (*i.e.*, meeting response time SLO). Different from existing works on serverless machine learning ([28, 47, 50], *etc.*), we jointly consider model partitioning dimension and resource provisioning. The solution aims to derive the serverless deployment for large machine learning models, where existing solutions do not apply. Yet, *AMPS-Inf* also offers better cost-efficiency by model splitting in the case when a model fits into a single serverless function.

To achieve the objectives of timely-response and cost-efficiency, *AMPS-Inf* relies on the formulation of a constrained optimization problem to explore the complete design space with the flexibility of model split and resource allocation. Given an inference job, *AMPS-Inf* calculates the optimal execution and resource provisioning plans for serverless deployment, constrained by the response time threshold (SLO) and the limitations of the serverless platform, such as the deployment size and temporary memory. More specifically, the decision variables include the number of partitions in the model computation graph, how the graph is partitioned, and the lambda function provisioning (the number of lambdas and the function resource type) to coordinate the partitions. This can be formulated as a Mixed-Integer Quadratic Programming (MIQP) problem [27] [25]. The solution can be obtained by the MIQP solver, which will be used by *AMPS-Inf* to enforce the deployment of serverless inference towards the optimal cost-efficiency without sacrificing response time performance.

We have implemented and deployed *AMPS-Inf* on AWS Lambda platform and evaluated with four pre-trained models, ResNet50 [35], MobileNet [36], Xception [29], and Inception-V3 [46], in Keras. Upon the submission of an inference job with the model file and weights, *AMPS-Inf* automatically partitions the model, provisions lambda functions, deploys model partitions and launches functions for coordinated model serving, following the calculated solution with the optimal cost-efficiency. Experimental results have demonstrated the efficient utilization of serverless platform for machine learning inference achieved by *AMPS-Inf*. It outperformed Amazon SageMaker, the production platform for machine learning, with respect to response time, by at least 47%, 17%, and 61% for ResNet50, Inception-V3, and Xception, respectively, with at least 92% cost reductions. For small model MobileNet, *AMPS-Inf* also achieved better response time performance than SageMaker while saving cost by 98%. We further compare with two heuristic baselines, where *AMPS-Inf* achieved 4% to 49% cost reduction for ResNet50, Xception and Inception-V3. We finally compare with the optimal deployment, where *AMPS-Inf* performed almost the same for Inception-V3 and no worse than 8% for ResNet50 and Xception.

The rest of the paper is organized as follows. Sec. 2 investigates serverless inference and motivates the model partitioning and resource provisioning for large-size models in a cost-efficient manner. Sec. 3 formulates and solves the constrained optimization problem to automatically partition model and provision functions for cost-efficient serverless inference. Sec. 4 presents the architecture overview of *AMPS-Inf* and describes its components. Sec. 5 implements *AMPS-Inf* and demonstrates its advantages over the industrial platform Amazon SageMaker and three baselines for pre-trained Keras neural network models. Sec. 6 discusses the related work and Sec. 7 presents concluding remarks.

2 BACKGROUND AND MOTIVATION

In traditional cloud environment, users are responsible for launching VMs, specifying operating systems and dealing with the scaling and management issues. On a serverless platform, a user's responsibility is simplified to writing functions and specifying events to trigger executions. In other words, the user does not need to worry about the deployment, management and environment complexities which are handled by the cloud provider. In the serverless environment, the launch and termination of functions, are as fast as a few milliseconds. The cost is in the "pay-per-use" mode: the execution time of a function, as well as unit time price based on the resource type, decides its cost (*e.g.*, AWS Lambda [16]).

Machine learning inference is the execution of a query over a trained model. With the recent advancement of machine learning and deep learning techniques, models grow in size and complexity, making it computationally expensive for both training and inferencing. The salient features of serverless computing have recently drawn research attention to deploy model serving (*i.e.*, inference) to quickly adapt to the query load dynamics. However, there remain challenges to be addressed for serverless machine learning inference. In what follows, we present the status quo of serverless inference, identify the challenges and motivate our proposed solution.

2.1 Machine learning inference in AWS Lambda

Serverless inference in AWS Lambda consists of lambda function with the serving code (*e.g.*, Python), associated ML dependency libraries (*e.g.*, Keras), ML model to be inferenced, and the resource configurations (*e.g.*, the allocated memory block [15]). The handler loads the model and uses imported dependencies on execution and serves for input(s).

The deployment package for the lambda is a ZIP archive that consists of the model, dependencies, and function code with 50MB uploading limitation [20]. For the models beyond 50MB (will be elaborated later in Table 1), the function layer feature of AWS lambda can be leveraged to pull in the model and the dependencies. It keeps the deployment package small with only the function code and helps to avoid installation errors [21]. An unzipped package with the size up to 250MB can be facilitated using a maximum of five layers. Still, the dependencies (keras [2], tensorflow [4], and the pillow [3]) are typically large. To further reduce their sizes, one approach is to rebuild the libraries by filtering out only the necessary packages. Such size-reduced dependencies can be published and specified by Amazon Resource Name (ARN) [1], which can be imported in serverless inference through a function layer.

Table 1 lists the deployment sizes of popular Keras Neural Network models, including ResNet50, and Inception-V3. The second column represents the size of a model which is determined by

[1] https://github.com/antonpaquin/Tensorflow-Lambda-Layer/tree/master/arn_tables

Table 1: Model and deployment sizes of neural network models. The deployment size includes the necessary dependencies of 169MB in AWS Lambda platform.

Neural Network Models	Model Size	Deployment Size
ResNet50	98MB	267MB
InceptionV3	92MB	261MB

the amount of its parameters. For example, ResNet50 model has 25,636,712 parameters and its size is $(25,636,712 * 4)/1024/1024 \approx 98$MB. These models all rely on Keras dependencies (169 MB). The total size for model deployment is represented in the last column. As shown, they are larger than the 250 MB size limit, which needs to be addressed in deploying serverless inference. On the other hand, the execution of the inference job needs to store the model parameters and neural network layer outputs within lambda storage which is available until the lambda terminates. The temporary storage of a lambda is limited to 512MB.

As advanced deep learning models have witnessed growing sizes, we are motivated to address this critical challenge for serverless machine learning inference, given the size limit on deployment and storage. In this paper, we design and implement a novel framework to automatically deploy model inference in the serverless platform by partitioning across a number of lambda functions. More particularly, our model partitioning and function provisioning strategy is in an optimal way towards cost-efficiency, without violating the response time threshold and the aforementioned size limits.

The resource configuration impacts both the cost and the job completion time performance. A larger memory allocation is likely to improve the job completion time, with a higher price per-unit time. We use MobileNet, a relatively small model which can be implemented with a single lambda for inference, as an example to illustrate how the memory allocation impacts cost and performance. Fig. 1 presents the corresponding cost and completion time given different allocations of memory for one image serving. Here the completion time intuitively decreases with the increase of memory allocation while the cost exhibits a growing trend. Note that the x-axis starts from 256MB rather than the minimum possible memory block 128MB, because there is a timeout limitation for function execution and inferencing with 128MB memory cannot complete before the timeout. With the memory size increasing, the performance improves while the cost decreases until a point and then increases. This happens in more than one points. The reason is that the cost is determined by function execution time and function price associated with memory type. Starting from the smallest memory size, though the price per unit time is cheap, the function execution time is large, dominating the total cost. On the other hand, increasing memory size beyond a threshold does not bring much decrease in completion time, while the memory cost increases sharply and dominates the total cost. Such a relationship will be considered in our model to guide appropriate memory allocation towards desired objective.

2.2 Motivating experiments

In this subsection, we present our experimental investigation that motivates our proposed approach.

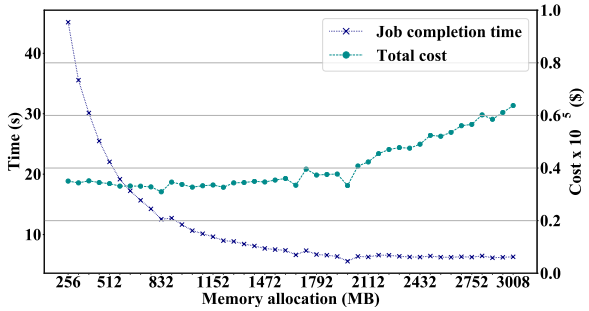

Figure 1: The completion times and costs for different memory allocations of MobileNet inference for one image. The x-ticks 1-44 represent the memory blocks from 256MB to 3008MB in 64MB increments.

We study the inference of pre-trained Keras MobileNet (<250MB) and ResNet50 (>250MB) models in the serverless environment. Particularly, we import the model (YAML file), weights (H5 file saved in the Hierarchical Data Format), and Tensorflow-Keras dependencies (169MB) in function layers in AWS Lambda. MobileNet is a small model and can be deployed in one lambda with 512MB memory allocation. ResNet50 cannot fit into one lambda and is randomly partitioned across ten lambdas for the inference job, each with 512MB memory allocation. Because of the missing feature of inter-lambda communication, there is a need for an intermediate storage, such as S3 [11], VM instance [9] or specially designed storage system ([41], *etc.*), to facilitate the transfer of intermediate data between lambdas. In our study, we leverage the simple one, S3, which could be extended to other storage to be discussed later.

We compare serverless inference with the commercial non serverless solution, Amazon SageMaker, which is a cloud platform for machine learning that integrates all the required components and automates the end-to-end workflow. The pricing of SageMaker includes the cost of on-demand instances, storage, and data processing in hosting instances [13]. SageMaker allows a cloud user to use Jupyter Notebook for its machine learning task. We consider two SageMaker settings, referred to as Sage 1 and Sage 2 throughout this paper. The user input models for both SageMaker settings are in the JSON format. Sage 1 uses a single instance of `ml.t2.medium` to store the model and weights, and deploy the model for inference. Sage 2 uses an instance of `ml.t2.medium` to handle the submission of inference jobs and invoke a hosting instance, `ml.m4.xlarge`, for model serving. HTTP endpoints are created and the model is stored in S3 in a rearranged format. The instances used in both settings are on-demand which have lower prices [13] compared to regular instances. We next compare the results for model serving in these three different settings.

2.2.1 Inference with one lambda. Fig. 2 shows the completion time and monetary cost of MobileNet serving with one image as input, in the three different settings aforementioned. The completion time is measured as the end-to-end completion time starting from model upload and ending with inference response. As clearly shown, the Lambda setting (with the function memory configured as 512MB) leads to the minimum cost when compared to SageMaker settings without degrading the completion time performance. The Sage 2 setting results in a large completion time due to the time required to run the hosting instance and deploy the model. Meanwhile, the host-

Table 2: Completion time and cost of MobileNet serving (one image request) given different memory types.

Memory (MB)	512	1024	1536	2048	3008
Time (s)	22.03	10.65	7.52	6.38	6.32
Cost ($)	0.00018	0.00017	0.00019	0.00021	0.00031

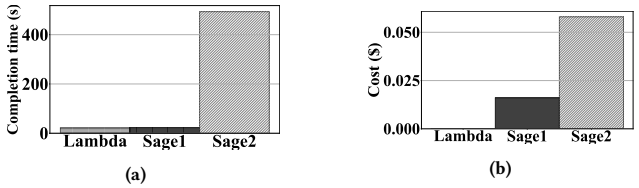

Figure 2: Completion time and cost of MobileNet serving (one image request) in three different settings: Lambda in AWS Lambda platform with 512MB allocation, Sage 1 in Amazon SageMaker's Notebook instance, and Sage 2 in Amazon SageMaker's hosting instance. Lambda cost is $0.00018.

ing instance has a higher price and incurs a larger cost, as shown in Fig. 2 (b). With respect to the Sage 1 setting, the completion time performance is similar to the Lambda setting, while the cost is higher due to the charge for the usage of Notebook instance and the re-arrangement into the necessary form of the model package (model.pb and assets). As evidenced, serverless inference exhibits great promise of cost-saving without degrading the completion time performance.

We further present the results for the Lambda setting with different memory configurations in Table 2. 1024MB is the configuration with the minimum cost. It also reduces the completion time by nearly half compared to the 512MB Lambda setting used in Fig. 2. This implies that the flexibility of resource configuration brings ample space for improving cost-efficiency.

2.2.2 Inference across lambdas. For the inference of ResNet50 model which exceeds 250MB in deployment size, we partition it across ten sequential lambda functions as discussed before. Table 3 presents the completion time and cost achieved by the ResNet50 model serving for one image, given the settings of Sage 1, Sage 2 and Lambda with different memory configurations. The incurred cost in Lambda setting includes the cost of function invocation and execution, and the cost of S3 requests for the intermediate data flowed across consecutive functions[2]. The Lambda settings with 512MB and 1024MB memory both result in smaller costs compared with Sage 1 and Sage 2, showing the cost-efficiency advantages of serverless inference. With the memory configuration of 1024MB, the completion time is shortened by at least half compared to the 512MB setting, which is the smallest among the four settings. Similar to the MobileNet serving in Fig. 2, Sage 2 takes the longest to complete, because of the time-consuming model deployment in the hosting instance. From this study, we have observed again the advantages of serverless inference, and seen ample optimization space with the flexibility of model split and resource configuration.

In summary, our background study on AWS Lambda limitations and the motivation experiments have demonstrated the promises

[2] Note that we did not use the AWS Step Function [17] to invoke lambdas, because the state transitions take nearly 15s which would cost more and lead to a larger completion time of nearly 108s.

Table 3: Completion time and cost of ResNet50 serving (one image) in different settings.

Settings	Sage 1	Sage 2	Lam. 512MB	Lam. 1024MB
Time (s)	33.346	484.509	47.078	21.799
Cost ($)	0.014	0.056	0.0017	0.0011

of neural network model partitioning across lambda functions for serverless inference, which also encourage us to investigate the best partition and resource configuration that can minimize cost without degrading performance.

3 SERVERLESS INFERENCE FOR COST-EFFICIENCY AND TIMELY RESPONSE

Based on our observations of motivation experiments in Section 2, there are great promises to exploit the serverless platform for cost-efficient machine learning inference. In this section, we formally model the response time performance and monetary cost with respect to model partition and resource allocation, based on which a constrained optimization problem is formulated and solved to achieve the maximum cost saving while maintaining timely response.

We consider a pre-trained neural network model with Y layers to be deployed on AWS Lambda. The complete set of all the possible model partitioning is denoted as $\mathcal{N}$. Given a particular partitioning, called a *cut* for ease of explanation, denoted as $g, g \in \mathcal{N}$, we specify $k, k \leq K$ lambdas to be coordinated for model serving, where K is limit on the maximum number of lambdas that can be requested. The number of layers in the partition that the i-th lambda ($i \in \{1, 2, \cdots, k\}$) will be allocated is represented by an integer variable y_i^g. Each lambda's memory allocation can be any from L memory blocks. We use the binary variable $x_{j,i}^g$ to denote whether the i-th lambda is allocated with the j-th type of memory ($j \in \{1, 2, \cdots, L\}$). Intuitively, we have

$$x_{j,i}^g \in \{0, 1\}, \quad \sum_{j=1}^{L} x_{j,i}^g = 1. \tag{1}$$

The total number of layers handled by all the lambdas is equal to Y, expressed as $\sum_{i=1}^{k} y_i^g = Y$.

For the partition in a cut g to be deployed on the i-th lambda, given the unit computation time $u_{j,i}^g$ with type j memory and the per-layer workload size d_i^g, the computation time of i-th lambda is $y_i^g d_i^g \sum_{j=1}^{L} x_{j,i}^g u_{j,i}^g$. Let p_i^g denote the intermediate output size of the partition and B denote the bandwidth between a lambda function and S3, the network transfer time of i-th lambda for cut g is $r_i^g = (p_{i-1}^g + p_i^g)/B$, for reading intermediate data from the previous partition and writing intermediate result to be used by the next partition. Thus, the completion time of the i-th lambda given a cut g is

$$T_{i,j}^g = y_i^g d_i^g \sum_{j=1}^{L} x_{j,i}^g u_{j,i}^g + r_i^g. \tag{2}$$

The monetary cost incurred by this lambda depends on the execution time $T_{i,j}^g$ and the price of its allocated memory $v_{j,i}^g$. In addition, during the execution of this lambda, all the intermediate outputs from previous partitions, with a total size of $q_i^g = \sum_{u=1}^{i-1} p_u^g$, are stored in S3 at the price of H per unit time and unit size. The lambda also incurs the cost of Get and Put request from S3, denoted

by G and U, and the lambda invocation cost I. Hence the total cost of this lambda is represented as

$$S^g_{i,j} = \sum_{j=1}^{L} v^g_{j,i} x^g_{j,i} T^g_{i,j} + q^g_i T^g_{i,j} H + (I + G + U). \tag{3}$$

Having expressed the completion time and the monetary cost incurred by the i-th lambda in cut g, we now consider the size limit constraints for the corresponding partition to be deployed on this lambda. The per neural network layer size of deployment package, containing the model description file and weights file, is represented as e^g_i. When the lambda is invoked, the deployment package will be loaded, in combination with the dependencies of size D and the handler of size F. The total size, as the deployment size, is limited by A depending on the serverless platform (*e.g.*, AWS lambda limitations/quotas [15]). In addition, there is a limit J on the size of temporary memory during execution, which is used to store the files of model and weights, the outputs of each neural network layer, and the prediction results from the previous partition. Let z^g_i denote the per neural network layer size of files related to the current partition that occupy the temporary memory (*i.e.*, all the files except for the last one mentioned in the previous sentence).

We now formulate the following optimization problem, which minimizes the cost of the i-th lambda given a cut g over the variables associated with model partition (y^g_i) and memory allocation ($x^g_{j,i}$), constrained by the platform limitations:

$$\begin{aligned}
\min_{\boldsymbol{x},\boldsymbol{y}} \quad & S^g_{i,j} \\
\text{s.t.} \quad & y^g_i e^g_i + D + F \le A && (4)\\
& y^g_i z^g_i + p^g_{i-1} \le J && (5)\\
& y^g_i \le \lceil Y/k \rceil && (6)\\
& 1 + \lceil ((\sum_{j \in L} x^g_{j,i} z^g_i) + D + F - M)/\beta) \rceil \le j && (7)\\
& x^g_{j,i} \in \{0,1\} && (8)
\end{aligned}$$

Constraints (4) and (5) regulate the limits for the deployment size and temporary storage size on the serverless platform. Constraint (6) caps the number of neural network layers per partition, for the consideration of reducing search space by removing intuitively unpromising solutions. Constraint (7) limits the number of memory blocks L in order to remove infeasible memory options. For example, assume the lambda in consideration needs a minimum of 500MB memory to store and execute function, dependencies, partition weights, *etc.* Given the minimum memory block size M as 128MB and block increment size β as 64MB on AWS Lambda, we have $j \ge 7$, which means that the feasible memory allocation is at least 576MB (7-th memory block). Hence, the memory blocks smaller than 576MB are infeasible and can thus be omitted by Constraint (7).

We next analyze the structure of the optimization problem. Substituting $T^g_{i,j}$ of Eq. (2) into Eq. (3) yields:

$$\begin{aligned}
S^g_{i,j} = & \sum_{j=1}^{L} v^g_{j,i} u^g_{j,i} x^g_{j,i} x^g_{j,i} y^g_i d^g_i + \sum_{j=1}^{L} v^g_{j,i} r^g_i x^g_{j,i} + \\
& \sum_{j=1}^{L} u^g_{j,i} q^g_i H y^g_i d^g_i x^g_{j,i} + q^g_i H r^g_i + I + G + U. \qquad (9)
\end{aligned}$$

If we use a single vector variable for both $\boldsymbol{x}$ and $\boldsymbol{y}$, the first term in the equation above is cubic in nature and the second term is quadratic, which make the problem very complex to solve. To make the problem more tractable, we apply Lagrangian multipliers as follows.

Since constraints (4), (5), and (6) of variable y^g_i do not depend on the other variable $x^g_{j,i}$, the objective function Eq. (3) and constraints (4)-(6) can be written as a function $G : F(\boldsymbol{x},\boldsymbol{y}) - \sum_{i=1}^{3} \lambda_i \rho_i(\boldsymbol{y})$, where $\rho_i(\boldsymbol{y})$ corresponds to Eq (4), (5) and (6), respectively. For example, $\rho_1(\boldsymbol{y}) = y^g_i e^g_i + D + F - A$. λ_i is the corresponding Lagrange multiplier. The feasible set of G is denoted as $\mathcal{W} := \{\boldsymbol{y} | \rho_i(\boldsymbol{y}) \le 0, i = 1, 2, 3\}$. Based on the L-subdifferential [38] of $F(\boldsymbol{x},\boldsymbol{y})$ at $\bar{\boldsymbol{y}}$ (the global minimum of G), we merge the new constraints of $\bar{\boldsymbol{y}}$ and λ_i in the objective function in a form of $F(\boldsymbol{x},\bar{\boldsymbol{y}}) - [\sum_{i=1}^{3} \lambda_i \rho_i(\boldsymbol{y}) - \sum_{i=1}^{3} \lambda_i \rho_i(\bar{\boldsymbol{y}})]$. Since one of the sufficient conditions for global minimizers is $\sum_{i=1}^{3} \lambda_i \rho_i(\bar{\boldsymbol{y}}) = 0$, we transform the optimization problem as:

$$\min_{\boldsymbol{x},\bar{\boldsymbol{y}}} \quad G = F(\boldsymbol{x},\bar{\boldsymbol{y}}) - \sum_{i=1}^{3} \lambda_i \rho_i(\bar{\boldsymbol{y}}) \tag{10}$$

$$\text{s.t.} \quad \text{Eq (7) and (8)}, \quad \bar{\boldsymbol{y}} \in \mathcal{W}. \tag{11}$$

The formulation falls into the category of linearly-constrained zero-one quadratic program on $\boldsymbol{x}$, given any $\boldsymbol{y}$. We consider the form of objective in Eq. (9) and substitute the real numbers Q_j, P_j, h_j and l_j. We re-arrange the formulation Eq. (10)-(11) given the known values of $q^g_i, H, v^g_{j,i}, d^g_i, y^g_i, r^g_i u^g_{j,i}, D$, and F:

$$\min_{\boldsymbol{x}} \quad \sum_{j=1}^{L} Q_j x_j x_j + \sum_{j=1}^{L} P_j x_j \tag{12}$$

$$\text{s.t.} \quad Eq(7) : \sum_{j=1}^{L} h_j x_j \le l_j \tag{13}$$

$$x_j \in \{0,1\}^L \tag{14}$$

To solve this problem, we build a quadratic convex reformulation using semidefinite relaxation [25]. As we do not have any equality constraint, replacing the product of $x_j x_j$ by a variable X_j yields:

$$\min \quad \sum_{j=1}^{L} Q_j X_j + \sum_{j=1}^{L} P_j x_j \tag{15}$$

$$\text{s.t.} \quad \sum_{j=1}^{L} h_j x_j \le l_j \tag{16}$$

$$X_j = x_j x_j, \quad j = 1, ..., L \tag{17}$$

$$x_j \in \{0,1\}^L \tag{18}$$

Using the semidefinite relaxation of the previous formulation, we can replace the constraints Eq. (17) and (18) with the linear matrix inequality $X = xx^t \ge 0$. From Schur's Lemma [51], the linear matrix is equivalent to $\begin{bmatrix} 1 & x^t \\ x & X \end{bmatrix} \ge 0$.

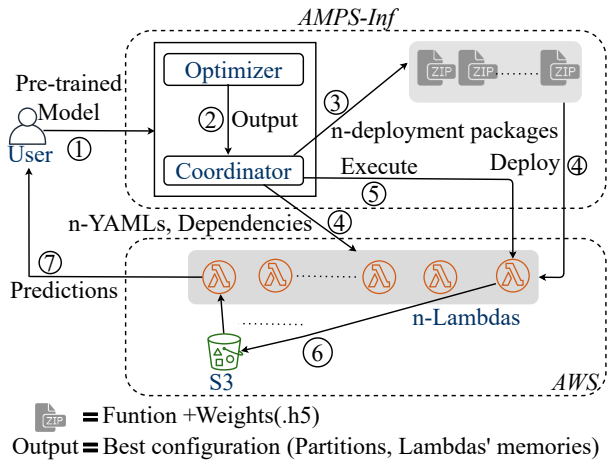

Figure 3: Architecture overview of *AMPS-Inf*.

Now the obtained form of SDP relaxation is:

$$\min \quad \sum_{j=1}^{L} Q_j X_j + \sum_{j=1}^{L} P_j x_j \tag{19}$$

$$\text{s.t.} \quad \sum_{j=1}^{L} h_j x_j \le l_j \tag{20}$$

$$\begin{bmatrix} 1 & x^t \\ x & X \end{bmatrix} \ge 0, \quad x \in \mathbb{R}^L, X \in S^L \tag{21}$$

Here the S^L defines $L \times L$ symmetric matrices. Now we use the optimal solution to this SDP in order to build a quadratic reformulation. We introduce the QCR method [25] of reformulating the formulation with adding a combination of quadratic functions that can vanish on a feasible solution set X. For any $\mu \in \mathbb{R}^L$, consider the following quadratic function:

$$F_\mu(x, \bar{y}) = \sum_{j=1}^{L} Q_j x_j x_j + \sum_{j=1}^{L} P_j x_j + \sum_{j=1}^{L} \mu_j (x_j^2 - x_j) - \sum_{i=1}^{m} \lambda_i g_i(y)$$

From (10) for any y ,

$$F_\mu(x, y) = F(x, y) + \sum_{j=1}^{L} \mu_j (x_j^2 - x_j) - \sum_{i=1}^{m} \lambda_i g_i(y) \tag{22}$$

The function $F_\mu(x, y)$ is a reformulation since for all $x \in X$, $F_\mu(x, y)$ is equal to $F(x, y)$. And we have to find the μ such that $F_\mu(x, y)$ is convex. So, from the semidefinite relaxation, $F(x, y)$ is transformed into convex. We can solve the reformulated problem (22) using mixed-integer convex quadratic programming. It has already been proved in [25] that solving the above semidefinite relaxation SDP allows us to deduce optimal values for μ. The optimal value μ_j^* of $\mu_j; j \in \{1, 2, \cdots, L\}$ will be given by the optimal values of the dual variables associated with constraints Eq. (20) and (21).

The resulting quadratic convex reformulation is:

$$RQ_{conv} : \text{Min} \quad F_{\mu^*}(x, y) \tag{23}$$

$$\text{s.t.} \quad \text{Eq. (16) and (18)} \tag{24}$$

The optimal value of SDP equals to the optimal of the continuous relaxation of RQ_{conv} and can be solved in polynomial time. The total number of neural network layers Y is always equal to the sum of the number of neural network layers of each partition (lambda) $\sum_i y_i^g$. Following the aforementioned deduction, minimizing cost of a lambda in Eq. (3) is a mixed-integer quadratic programming (MIQP). Intuitively, we can obtain the minimization of the total cost $\sum_i S_{i,j}^g$ with constraints Eq. (4)-(8) by solving MIQP problems, using any MIQP solver such as Gurobi [1], CPLEX [26], *etc.*

4 DESIGN AND IMPLEMENTATION

In this section, we present our design and implementation of *AMPS-Inf* for automatic serverless inference in AWS Lambda.

Fig. 3 illustrates the architecture overview, which relies on the Optimizer to find the best execution plan and resource allocation for the pre-trained model (in YAML/JSON format) as user input. The problem formulation and solution presented in Sec. 3 are implemented in the Optimizer component, which generates the best configuration with minimum cost, selected from the different combinations of model partitions and lambda memory allocations. The Coordinator component creates the zipped deployment packages, which consist of function and weights files of each partition, and deploy them on the AWS Lambda platform. Given the Optimizer's solution of partition points, *AMPS-Inf* divides the YAML file into partitioned ones, adds input and output layers, and uploads them (and dependencies) to lambdas. The inference starts by calling the lambda corresponds to the first partition, followed by sequential invocations of the other lambdas. The intermediate output of each is stored in AWS S3. The final prediction will be sent back to the user.

Automatic model partitioning. The state-of-the-art architecture mentioned in a report [42] handles the AlexNet inference using Pytorch Framework in AWS Lambda with AWS EC2 instance as a driver, Redis/S3 for storage, and AWS step functions to execute the workflow. When an image arrives, the driver partitions and uploads them to the shared storage, and invokes the first lambda layer. Here the lambda layer (Lambda function) represents a part of the model which needs to be written by the user as a python class with a handler including feature forwarding method. In contrast, *AMPS-Inf* does not require the user's efforts to partition the model or write the model as a handler's class. The complexities associated with model partitioning are hidden from the user, as *AMPS-Inf* judiciously partitions the model, automatically deploys partitions and coordinates their executions on lambdas. Following the observations of motivation experiments in section 2, to avoid the non-trivial time and cost incurred by state transitions, *AMPS-Inf* does not use the Step functions, and for the same reason removes the need for the driver (EC2 instance).

Optimizer. The Optimizer is the core component of *AMPS-Inf*. The detailed view of the optimizer is shown in Fig. 4. The Profiler calculates all the possible ways for the partition of the given pre-trained model. For example, a pre-trained model with 3 layers has the following possible partition combinations: (3), (1,2), (2,1) and (1,1,1), where (1,2) means that the first partition has the first neural network layer and the second partition consists of the next two layers. The current possible memory blocks of the AWS Lambda

are also available to the Profiler. The Optimizer needs to select the best one from the entire solution space by formulating and solving the serverless inference problem with cost optimization, which has been elaborated in section 3. The problem is translated to Mixed Integer Quadratic Programming (MIQP) and any MIQP solver such as Gurobi [1], CPLEX [26], *etc.* can be used to find the solution. Our solver CVXPY [32] is implemented in Python. The optimized solution is the best configuration incurring minimized cost and achieving acceptable performance, which consists of the neural network layer partitions and memory blocks for each partition. This final output from the Optimizer will be sent to the Coordinator.

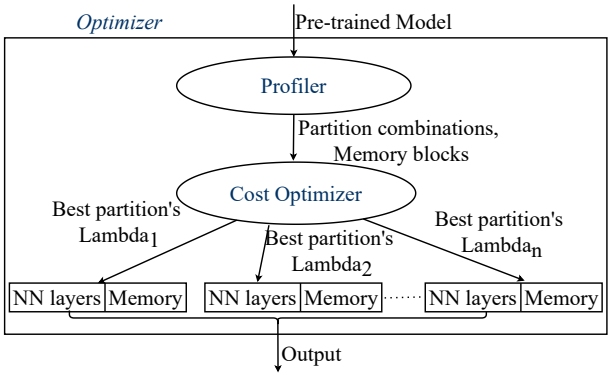

Figure 4: Optimizer component of *AMPS-Inf*.

Coordinator. The Coordinator of *AMPS-Inf* is designed to create the partitioned models' weights as deployment packages and add the partitioned YAML files with neural network dependencies as layers of the deployed lambdas. The Coordinator read and split the pre-trained model as separate NN layers. It lists the necessary parameters (weights,inputs, outputs and parameters) from the model summary. Meanwhile, the Coordinator carefully checks the layers' dependencies (or connection) with other layers when combining them into a partition. Implemented in Python, the Coordinator enforces the partition, deployment, coordination and launch of the given pre-trained model, based on the configuration decision made by the Optimizer.

5 PERFORMANCE EVALUATION

5.1 Experimental setup

AMPS-Inf is evaluated on AWS Lambda platform with four different Keras pre-trained models: MobileNet, ResNet50, InceptionV3, and Xception. Two settings of Amazon SageMaker are compared with *AMPS-Inf*. Since the experiments were performed during October-November in 2020, the calculations and measurements of the used AWS services followed the quotas and pricing schemes for that duration. It is worth noting that AWS Lambda's function memory allocation quota [15] has recently been updated as a maximum of 10,240MB in 1MB increments, while the deployment package size remains 250MB (unzipped). *AMPS-Inf* still works and can be easily extended with the new quota, which will be left as our future work.

We evaluate *AMPS-Inf* compared with two SageMaker settings, three additional baselines and the state-of-the-art [23, 42]. As described in Sec. 2.2, the first SageMaker setting, *Sage 1*, uses instance-based notebook (`ml.t2.medium`) to deploy the model for serving. The second setting, *Sage 2*, uses instance-based notebook (`ml.t2.medium`) to handle the job submission which invokes an `ml.m4.xlarge` instance for model deployment and inference. The uploaded model (JSON) and weights (.h5) are converted and stored as assets, variables, and model.pb in AWS S3. The additional three different baselines for comparison are based on two heuristics and the optimal solution:

Baseline 1: Choose the way of partition randomly and select the memory allocation randomly for all the lambdas.

Baseline 2: Starting from the last layer of a neural network model, include the layers one by one into a partition until the platform limit is about to hit, and continue the procedure to form the next partition. Allocate the maximum memory for all the lambdas.

Baseline 3: The optimal configuration obtained through exhaustive search.

5.2 Comparison with SageMaker

The machine learning inference job runs in steps of loading model and weights, deploying, and making prediction, in both AWS Lambda and Amazon SageMaker settings. Fig. 5 shows the time it takes to load files, respectively, for serving the three models with an image as input. *AMPS-Inf* loads the model and weights from the same triggered lambda since each model partition (YAML) and its weights (.h5) are attached to the corresponding lambda. The first SageMaker setting (Sage 1) loads the model from the package (model.pb, assets, variables) deployed in the same serving notebook instance. In Sage 2, the model is loaded from AWS S3 by the hosting instance. Intuitively, due to the network transfer time, the loading in Sage 2 is longer when compared to the *AMPS-Inf* and Sage 1 which are self-loading. Since *AMPS-Inf* separates the model and weights files into smaller partitions and due to the few milliseconds latency of lambda platform, the sum of loading time over all lambdas is still the minimum, when compared to loading the whole model and its assets and variables within an instance in Sage 1.

Table 4: The overall time spent for deployment and prediction (one image request) in Sage 2.

Pre-trained models	ResNet50	Inception-V3	Xception
Prediction Time(s)	463.482	462.303	401.787

Fig. 6 compares the prediction time per image achieved by *AMPS-Inf* and Sage 1. The same reasons aforementioned and the few milliseconds latency of temporary storage handling the neural network layer outputs within the partitions lead to a smaller serving time of *AMPS-Inf* than Sage 1. Since the prediction time in Sage 2 is not practically measurable, we do not include it in our comparison in Fig. 6. However, the sum of deployment time and the prediction time in Sage 2 can be measured, which is thus presented in Table 4, evaluated over the same three models. A great amount of time in deployment is for creating endpoints and launching the hosting instance, while the prediction is executed by loading model from S3 which can be comparable to *AMPS-Inf* and Sage1.

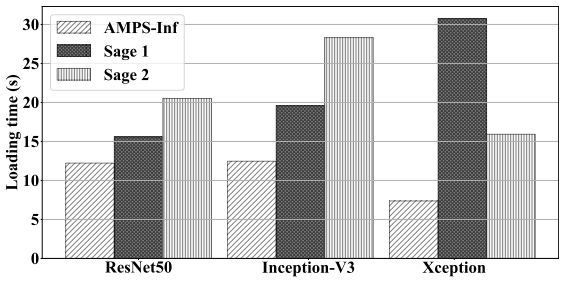

Figure 5: The time for loading model and weights.

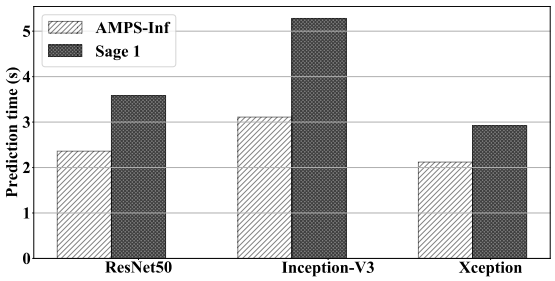

Figure 6: The time for prediction (one image request).

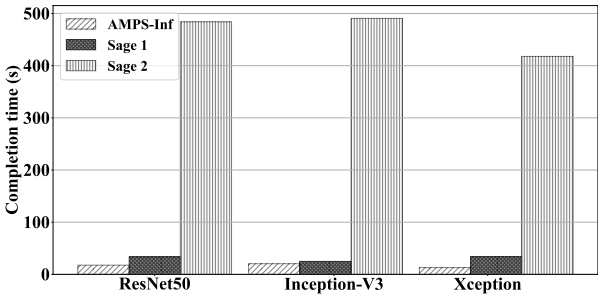

Figure 7: Completion times for serving one image in three different neural network models in two platforms (Lambda and SageMaker) with three different settings.

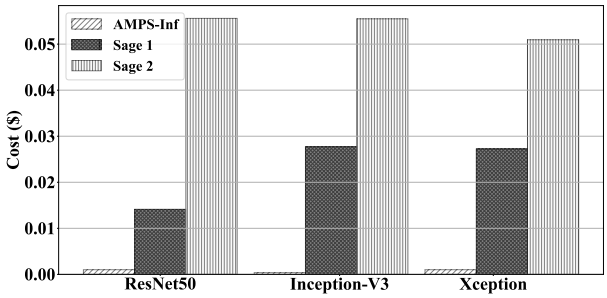

Figure 8: Total cost for serving one image in three different neural network models in two platforms (Lambda and SageMaker) with three different settings.

We measured and compared the response time and overall cost of the whole inference job in Fig. 7 and Fig. 8. ResNet50 completed the serving of an image with four lambdas, each with 1536MB, 1408MB, 1408MB, and 1344MB memory allocations, determined by the Optimizer component in *AMPS-Inf*. For Inception-v3, three lambdas were provisioned for its three partitions, with memory of 640MB, 448MB, and 384MB, respective. Xception was partitioned and deployed on three lambdas with 1536MB, 960MB, 1024MB memory configurations. *AMPS-Inf* took a few milliseconds to accomplish the configuration calculations. As shown in Fig. 7, all three pre-trained models completed the one image serving in fewer completion times compared to the SageMaker settings because of the reasons aforementioned for the steps of loading, deployment, and prediction. Since the intercommunication between lambda functions is currently not well supported, *AMPS-Inf*, with the need for intermediate storage, shows less amount of performance improvement in some cases As *AMPS-Inf* can be extended to use any intermediate storage such as Redis [10] and Pocket [41] that are more sophisticatedly designed, there is opportunity to further increase its performance.

The advantage of the pay-per-use pricing scheme of AWS Lambda and the proportionality of performance to memory configuration result in cost reduction of *AMPS-Inf* for ResNet50, Inception-v3, and Xception by 92.85%, 98.67%, and 96.29%, respectively, when compared to Sage 1, as shown in Fig. 8. Similarly, in comparison with Sage 2, *AMPS-Inf* achieves cost reduction of 98.18%, 99.33%, and 98.02%, respectively, for the three models. The incurred cost in Sage 1 depends on the Notebook instance price, the duration of its running, and the storage cost for model weights (in or out cost). In Sage 2, the cost depends on the running time of both the Notebook instance and the hosting instance, as well as the S3 data transfer cost and storage cost. The VM instance running contributes a lot to the overall cost in both SageMaker settings. In summary, *AMPS-Inf* has demonstrated its superiority on cost-efficiency when comparing to the SageMaker platform.

5.3 Comparison with baselines and the state-of-the-art

We further compare *AMPS-Inf* with three baselines, of which the first two are heuristics and the third one is the optimal solution to cost minimization. Fig. 9 and Fig. 10 illustrate the completion time and overall cost achieved by *AMPS-Inf* and the three baselines for three pre-trained models.

For ResNet50, Baseline 1 randomly selected 1024MB memory allocation for all the 10 lambdas provisioned for a randomly selected way of model partition. The partition heuristic in Baseline 2 resulted in the smallest number of lambdas, and these four lambdas were allocated the largest memory 3008MB (during October-November, 2020). As shown in Fig. 9, with fewer lambdas, each lambda in Baseline 2 needs to handle more data transferred from/to S3 when compared to Baseline 1, which explains its slightly larger completion time. With respect to monetary cost, the usage of maximum memory and the larger completion time resulted in a larger cost of Baseline 2 compared to Baseline 1, as shown in Fig. 10. Baseline 3 adopted the cost-optimal solution and intuitively resulted in the smallest cost. More specifically, the optimal way of partitioning led to 4 lambdas, each with the memory allocation of 384MB, 384MB, 768MB and 832MB, respectively. Though the number of lambdas

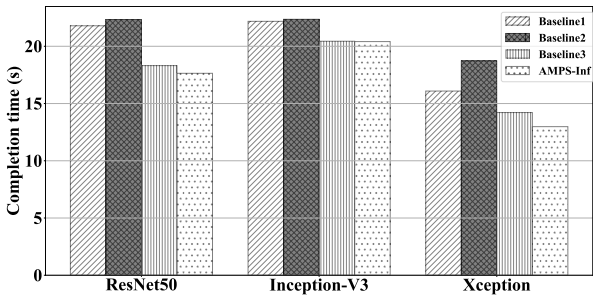

Figure 9: Completion times for serving one image in three different neural network models in four different lambda settings.

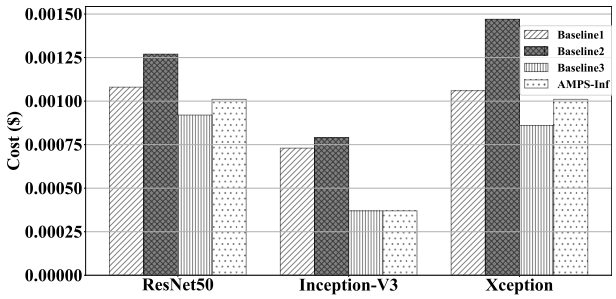

Figure 10: Total cost for serving one image in three different neural network models in four different lambda settings.

is the same with Baseline 2, the optimal solution in Baseline 3 has better way of partitioning and more reasonable resource allocation which contributed to both faster job completion and cost reduction.

AMPS-Inf also achieved shorter completion time and smaller cost when compared to the first two baselines. In comparison with Baseline 3 which is cost-optimal, *AMPS-Inf* showed ≈ 9% increase in cost while achieving ≈ 4% better completion time performance. In particular, *AMPS-Inf* partitioned the model into four parts, deployed on lambdas allocated with 1408MB, 1408MB, 1344MB, 1536MB memories. The larger blocks of memories, compared with Baseline 3, sped up the inference while increasing the monetary cost slightly.

For Inception-V3 and Xception, we have similar observations and analysis on the completion time performance and cost efficiency achieved by *AMPS-Inf* and the three baselines. In particular, *AMPS-Inf* behaved almost the same with Baseline 3 for Inception-V3, regarding both completion time and cost. For Xception, *AMPS-Inf* outperformed Baseline 3 by achieving ≈ 9% faster completion, despite incurring ≈ 14% more cost.

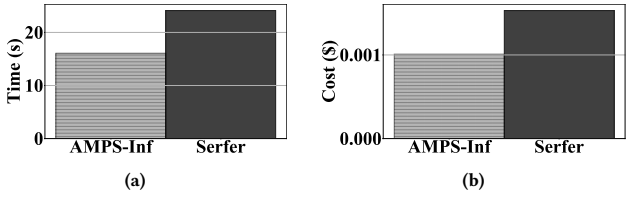

Figure 11: Completion time and total cost of ResNet50 inference (one image request) achieved by Serfer and *AMPS-Inf*.

Next, we present our comparison with the state-of-the-art serverless inference framework, Serfer [42]. Since Serfer [42] does not support automatic partition/configuration and does not give any guidelines, we use the same partition and configuration as *AMPS-Inf* for ResNet50. The differences are: Serfer splits the single image, uses step function, and requires manual model splitting. Figure 11 shows the execution time and cost for image serving achieved by *AMPS-Inf* and Serfer, clearly demonstrating that our work outperforms Serfer in both reducing completion time and cost.

5.4 Small NN, batch inference and discussion

The performance of *AMPS-Inf* for small models.

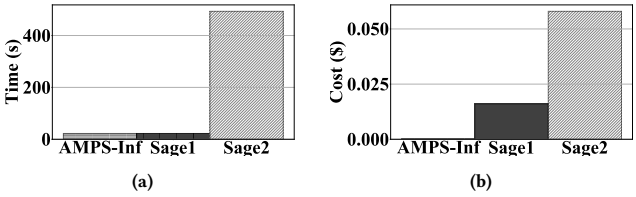

Figure 12: Completion time and total cost of MobileNet inference (one image request) in three different settings: Sage 1 in Amazon SageMaker's Notebook instance, Sage 2 in Amazon SageMaker's hosting instance, and *AMPS-Inf*. *AMPS-Inf*'s cost is $0.00019.

As explained in Section 2, MobileNet's deployment size is less than 250MB, which makes it feasible for single lambda model serving.

Recall that Fig. 2 presented in Section 2 shows the cost-effectiveness and acceptable performance of Lambda (512MB) compared to Amazon SageMaker. With *AMPS-Inf*, two lambdas with 1024MB and 960MB memories, respectively, will be provisioned for the deployment and serving of MobileNet.

Fig. 12 presents the results of completion time and cost achieved by *AMPS-Inf* in comparison with two settings of Amazon SageMaker. Still, *AMPS-Inf* shows much improvement in terms of both completion time and monetary cost over Amazon SageMaker.

The extension of *AMPS-Inf* to batch inferencing. As batch inferencing is likely to further improve the efficiency of model serving, we further conduct a preliminary investigation on the feasibility and promise of *AMPS-Inf* to be extended for batched parallel inference. In particular, we consider the model serving with ten different images in parallel. The images are loaded as .pkl files. The completion time and total cost of the whole inference job for ten parallel servings are presented in Table 5 As shown in Table 5, *AMPS-Inf* achieves cost reduction of at least 53%, 66%, and 60%, with at least 7%, 19%, and 29% performance improvement, compared to SageMaker for ResNet50, Inception-V3, and Xception, respectively. This clearly demonstrates the potential of *AMPS-Inf* for batch inferencing in a more general setting.

We continue to conduct experiments to compare with BATCH [23], a serverless inference solution with batching, for processing 100 images in 10 batches on MobileNet. The configurations calculated by *AMPS-Inf* are two lambdas(/partitions) with 2048 MB and 2176 MB memory. BATCH does not support model splitting and

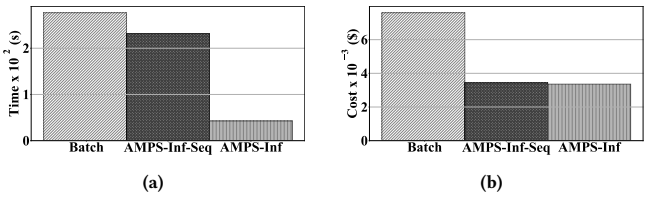

Figure 13: Completion time and total cost of MobileNet batch inference achieved by Batch and *AMPS-Inf*.

we assigned 2048 MB for its single lambda. Given the batch size of 10, BATCH sequentially invokes a lambda per batch for the 10 batches. As shown in Figure 13, the resulted completion time and cost are 276.84s and $0.0095. Although *AMPS-Inf* supports invoking lambdas to process all batches in parallel, we let *AMPS-Inf* process sequentially, similar to BATCH for a fair comparison, denoted by *AMPS-Inf-Seq* in Figure 13. The resulted time and cost are 231.36s and $0.0043, outperforming BATCH. When further using parallel invoking for the 10 batches, AMPS-Inf finished within 42.61s at the cost of $0.0042. These results demonstrate the advantages of AMPS-Inf still hold for batch processing.

Table 5: Completion time and cost for a batch serving with 10 images.

	Settings	ResNet50	Inception-V3	Xception
Time(s)	*AMPS-Inf*	23.94	27.03	10.35
	Sage1	25.77	33.59	14.59
	Sage2	432.10	432.62	423.05
Cost($)	*AMPS-Inf*	0.0070	0.0047	0.0052
	Sage1	0.015	0.014	0.013
	Sage2	0.053	0.052	0.051

Discussion and future work. The running time overhead of *AMPS-Inf* incurred by the MIQP solver is within a few seconds on a laptop (Intel®Core™i7-8750H CPU@2.20GHz×12,2×8GiB memory). It is expected that the overhead is negligible (in milliseconds) on a more powerful commodity server. Though implemented in AWS Lambda, *AMPS-Inf* can be adapted to Google Functions and Azure Functions with minimal modifications in our model and problem formulation. Extension to other platforms will be left as our future work. Since advanced neural network models (such as BERT [31]) keep growing in size and complexity, it may be possible that even a single layer is too large to fit into a lambda function in the future. To overcome this issue, we will consider automatically quantizing [34] the weights before the deployment on a serverless platform.

In our future work, we will extend the design for batch inference serving at a very large scale. We will also evaluate *AMPS-Inf* for the models in other frameworks such as Tensorflow, PyTorch, and *etc.*

6 RELATED WORKS

A number of existing efforts have studied the employment of serverless platforms in machine learning. Seneca [52] leverages stateless functions for hyperparameter tuning. Each function is used to train and evaluate the machine learning model, given a set of hyperparameters. Cirrus [28] focuses on the iterative model training process, which utilizes lambda functions to efficiently handle computation workloads in each training iteration. It addresses the unnecessary large memory provisioning by streaming training mini-batches from a remote storage and revises the training algorithms to work precisely. SIREN [47] applies deep reinforcement learning to determine the number and memory size of the stateless functions for each training epoch, with the objective of minimizing the training time given a budget.

Apart from the existing efforts aforementioned with the focus on training, another category of work targets at provisioning the workloads of machine learning inference or model serving ([23, 24, 30, 50], *etc.*). The production serving systems provided by cloud providers ([7, 8, 22, 44], *etc.*) facilitate the deployment of trained models in containers. Amazon SageMaker [12] supports model serving over EC2 instances. MArk [50] studies the capabilities of stateful and stateless architectures to support batch inferencing requests, given the objective of meeting the service level requirements. It primarily relies on Infrastructure-as-a-Service (IaaS) provisioning for model serving, while leveraging Function-as-a-Service (FaaS) for horizontal/vertical scaling to adapt to increasing workloads. BATCH [23] is prototyped on AWS Lambda for inference serving, where requests are buffered to be later processed in a batch. Its performance optimizer provisions lambda functions based on the distribution of the requests in the buffer. In all the existing serving systems in the serverless environment, the model (each copy) will be deployed on a single lambda function by default, without considering the infeasibility issue when a model is larger than the deployment size limit. To fill this gap, Gillis [48] and our *AMPS-Inf* are two concurrent works on automatic model partitioning and resource provisioning for large model inference in the serverless environment, with awareness of both SLO and cost. While Gillis adopts reinforcement learning based approach for model partitioning and further enables parallelism within a partition, *AMPS-Inf* focuses on finer-grained problem modeling and employs optimization-based solution.

7 CONCLUDING REMARKS

Serverless computing has exhibited great promise in recent years, attracting research attention on migrating machine learning workloads towards serverless platforms. In this paper, we examine the challenges of serverless provisioning for machine learning inference due to the increasing size of advanced models and the limit on the deployment size of serverless function. To address the challenges on splitting model and coordinating partitions, and to hide these complexities from users, we design and implement *AMPS-Inf*, an automated framework for serverless machine learning inference towards cost-efficiency and timely-response. In particular, *AMPS-Inf* solves the Mixed-Integer Quadratic Programming problem for model partitioning and resource provisioning, with the objective of cost minimization while satisfying the response time service level requirement (SLO). Deployed in AWS Lambda, *AMPS-Inf* is evaluated with four pre-trained models, in comparison with Amazon SageMaker and three different baselines. Results demonstrate that *AMPS-Inf*, by finding the best configuration of lambda resource type and model partitions, achieves cost saving of up to 98% without degrading the response time performance.

ACKNOWLEDGMENTS

This work was supported in part by the Louisiana Board of Regents under Contract Numbers LEQSF(2019-22)-RD-A-21 and LEQSF(2021-22)-RD-D-07, in part by National Science Foundation under Award Number OIA-2019511, in part by the NSFC under grant No.61972158, in part by RGC RIF grant R6021-20, and RGC GRF grants under the contracts 16207818 and 16209120.

REFERENCES

[1] *GUROBI Optimization.* Retrieved December 20, 2020 from https://www.gurobi.com/
[2] *Keras.* Retrieved January 5, 2021 from https://keras.io/
[3] *Pillow.* Retrieved January 5, 2021 from https://pillow.readthedocs.io/en/stable/
[4] *TensorFlow.* Retrieved January 5, 2021 from https://www.tensorflow.org/
[5] *VGG16 Function.* https://keras.io/api/applications/vgg/#vgg16-function
[6] *VGG19 Function.* https://keras.io/api/applications/vgg/#vgg19-function
[7] 2018. *PredictionIO.* https://predictionio.apache.org/
[8] 2018. *RedisML.* https://github.com/RedisLabsModules/redisml
[9] 2020. *Amazon EC2.* https://aws.amazon.com/ec2/
[10] 2020. *Amazon ElastiCache.* https://aws.amazon.com/elasticache/
[11] 2020. *Amazon S3.* https://aws.amazon.com/s3/
[12] 2020. *Amazon SageMaker.* https://aws.amazon.com/sagemaker/
[13] 2020. *Amazon SageMaker Pricing.* https://aws.amazon.com/sagemaker/pricing/
[14] 2020. *AWS Lambda.* https://aws.amazon.com/lambda/
[15] 2020. *AWS Lambda Limits.* https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
[16] 2020. *AWS Lambda Pricing.* https://aws.amazon.com/lambda/pricing/
[17] 2020. *AWS Step Functions.* https://aws.amazon.com/step-functions/
[18] 2020. *Azure Functions.* https://azure.microsoft.com/en-us/services/functions/
[19] 2020. *Cloud Functions.* https://cloud.google.com/functions
[20] 2020. *Deploy Python Lambda functions with .zip file archives.* https://docs.aws.amazon.com/lambda/latest/dg/python-package.html
[21] 2020. *Lambda Layers.* https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html
[22] 2020. *Multi Model Server.* https://github.com/awslabs/multi-model-server
[23] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 972–986.
[24] Anirban Bhattacharjee, Zhuangwei Kang Ajay Dev Chhokra, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. Barista: Efficient and Scalable Serverless Serving System for Deep Learning Prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 23–33.
[25] Alain Billionnet, Sourour Elloumi, and Marie-Christine Plateau. 2008. Quadratic 0–1 Programming: Tightening Linear or Quadratic Convex Reformulation by Use of Relaxations. *RAIRO-Operations Research* 42, 2, 103–121.
[26] Christian Bliek1ú, Pierre Bonami, and Andrea Lodi. 2014. Solving Mixed-Integer Quadratic Programming Problems with IBM-CPLEX: a Progress Report. In *Proceedings of the twenty-sixth RAMP symposium*. 16–17.
[27] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. 2018. Learning a Classification of Mixed-Integer Quadratic Programming Problems. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 595–604.
[28] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.
[29] François Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.
[30] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.
[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).
[32] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.
[33] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[34] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149* (2015).
[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[36] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).
[37] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. May 17-21, 2021. Astra:Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness. In *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2021)*.
[38] Vaithilingam Jeyakumar, Alex M Rubinov, and Zhi You Wu. 2007. Non-Convex Quadratic Minimization Problems With Quadratic Constraints: Global Optimality Conditions. *Mathematical programming* 110, 3 (2007), 521–541.
[39] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451.
[40] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. In *11th International Conference on Cloud Computing (CLOUD)*. IEEE.
[41] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 427–444.
[42] Mohan Kodandarama, Mohammed Shaikh, and Shreeshrita Patnaik. 2020. SerFer: Serverless Inference of Machine Learning Models. (2020). https://divatekodand.github.io/files/serfer.pdf
[43] Benjamin D Lee, Michael A Timony, and Pablo Ruiz. 2019. DNAvisualization. org: a Serverless Web Tool for DNA Sequence Visualization. *Nucleic acids research* 47, W1 (2019), W20–W25.
[44] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-Serving: Flexible, High-Performance Ml Serving. *arXiv preprint arXiv:1712.06139* (2017).
[45] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[46] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, and Jonathon Shlens. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.
[47] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE Conference on Computer Communications,INFOCOM 2019*. IEEE, 1288–1296.
[48] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving Large Neural Networks in ServerlessFunctions with Automatic Model Partitioning. In *41st IEEE International Conference on Distributed Computing Systems*.
[49] Diego Zanon. 2017. *Building Serverless Web Applications.* Packt Publishing Ltd.
[50] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, Slo-aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.
[51] Fuzhen Zhang. 2006. *The Schur Complement and Its Applications.* Vol. 4. Springer Science & Business Media.
[52] Michael Zhang, Chandra Krintz, Rich Wolski, and Markus Mock. 2019. Seneca: Fast and Low Cost Hyperparameter Search for Machine Learning Models. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 404–408.