Training Recommender Systems using Serverless Architecture with Data Parallelism

Eric He
UC San Diego
zuhe@ucsd.edu

Abstract

The advent of serverless computing has brought convenience and efficiency to many applications with its lightweight management and pricing model. As platform like AWS Lambda is gradually increasing its capacities with larger memory and greater package support, recent research have started to experiment serverless architecture with more data-intensive workloads such as Machine Learning training. In this paper, we ported a ML-based Recommender System workload onto AWS Lambda and profiled its resource consumption. Using the concurrency of Lambda functions, we re-implemented the program with data parallelism and improved the training speed by 51% without significant overhead.

1 Introduction

1.1 Problem Statement

In this paper, we will first manually port machine learning based recommender system to the AWS serverless platform. After the code is ported and runs smoothly, we will rewrite the code as DAG of Lambda functions and parallelize computation to improve the performance of the program. we will also analyze the trade-off between speed and cost by profiling the resource utilization and experimenting with different parameters such as number of child workers, memory size, and data size.

1.2 Motivation

Recommender System has been one of the most popular applications of Machine Learning as it sits at the center of many social media apps and e-commerce websites. Given the trend of big data, the companies need to frequently re-train their models to give better predictions for existing and new users. Currently, machine learning and deep learning workloads have been mostly deployed using clusters of virtual machines[8]. However, there are several downsides [1] of using IaaS based training.

1. **Resource Management**: IaaS approach requires users to provision, configure and manage low-level VM resources such as memory and CPU for their ML workloads which might be very time-consuming and error prone. For instance, there are many combinations of instance types and settings that users can choose from on AWS EC2 for their workloads.
2. **Cost**: Since users have to pay for every second the cluster is running, there could be a large amount of time where the virtual machines are left idle when they are not training.
3. **Over-provisioning:** Since ML workloads can vary greatly depending on the model and dataset. Users might over-provision the resource for peak consumption and end up wasting significant amount of data center resource. The rigidity of VM clusters and ML framework make users hard to find a balance when provisioning.

Serverless Architecture(FaaS) was initially introduced in 2014[5] and it has been adopted by applications such as web servers, video processing and data analytic etc. The design principles of serverless make it a neat solution for the problems mentioned above. Severless framework like AWS lambda lifts the burden of provisioning and management and follows a ”pay by usage” pricing model. In other words, there is no cost incurred when the functions are not running. At last, it is highly scalable as users have the flexibility of launching up to 3000 concurrent functions.

With reasons provided above, we believe serverless architecture provides a good solution for training ML-based Recommender System workloads. In particular, it allows us to exploit techniques such as data parallelization which greatly speeds up the training process.

1.3 Background

Latent factor models are popular for recommender system as they represent the items and users by vectors of latent factors inferred from item rating patterns [6]. These models are based on matrix factorization and can be derived using Singular Value Decomposition. However, given the rating matrix is often large and sparse, the computation for SVD becomes very costly as datasets get large. The common practice is to compute the latent variables using gradient descent. The weight update is shown as follows:

$$w_{k+1} = w_k - \alpha \nabla l(w_k; X, Y)$$

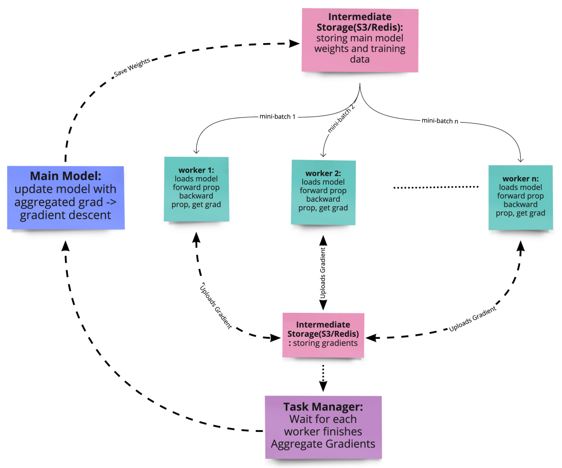

Intermediate Storage(S3/Redis): storing main model weights and training data
Save Weights
mini-batch 1
mini-batch 2
mini-batch n
Main Model: update model with aggregated grad -> gradient descent
worker 1: loads model forward prop backward prop, get grad
worker 2: loads model forward prop backward prop, get grad
worker n: loads model forward prop backward prop, get grad
Uploads Gradient
Uploads Gradient
Uploads Gradient
Intermediate Storage(S3/Redis) : storing gradients
Task Manager: Wait for each worker finishes Aggregate Gradients

Figure 1: Data Parallelism Design Flow

The main idea for gradient descent is to first compute the gradient and then move the weight in the opposite direction of its gradient to minimize the cost function. In this paper, we will be using Batch Gradient Descent which takes the average of the gradients of all training data and use it to update the weights in each epoch. Another more approach is Stochastic Gradient Descent which splits the training data into smaller batches in each epoch. However, our method will generalize to both approaches easily.

The recent paradigm in Machine Learning/Deep Learning is that the increase in model size, dataset size and training length often lead to better model performance. Therefore it becomes impractical to put all training data into the memory. One of the common practice used in training ML workloads is data parallelism which splits up the training data into smaller batches and assign each batch to a worker to train in parallel.

2 Data Parallelization

In this section, we will talk about the design of data parallelism using the serverless architecture and some of the implementation details that are important for the performance of the program.

2.1 Design Overview

Figure 1 is a visualization of training with distributed gradient descent as mentioned in earlier section. In each epoch, the program executes in the following steps:

1. **Data Distribution:** Training data splits into n smaller batches and send to workers from 1 to n.

2. **Compute Gradients:** Worker loads the data and main model from intermediate storage. Forward and backward propagates the data and compute the gradient. When finished, each worker saves the gradients to intermediate storage. (Note: all workers are executing in parallel)

3. **Gradient Aggregation:** Task Manager checks the intermediate storage and aggregates all of the gradients from the workers.

4. **Model Update:** The task manager sums all of the gradient and send it to the main model. The main model then performs gradient descent. The weights of the main model is updated and saved in intermediate storage.

5. **Repeat:** return to step 1

2.2 Implementation Details

2.2.1 Model Detail

The model is called Biased Matrix Factorization [6] and it is implemented using PyTorch. The model learns the latent vector for user, item and mean through gradient descent and predicts the rating as follows given a (user,item) pair:

$$rating = mu + user\_bias + item\_bias + user\_vec@item\_vec$$

The loss function is MSE and the model is trained with learning rate 0.01, momentum 0.9 and hidden dimension 3.

2.2.2 Task Manager

Algorithm 1 Gradient Aggregation

```
Require: $S3BucketAccess, workers$
  $numAdded \leftarrow 0$
  $grad\_arr \leftarrow []$
  while $numAdded < len(workers)$ do
      $response \leftarrow s3.listObjects$
      if $response['count'] > 0$ then
          for $grad$ in $response['content']$ do
              read and decode $grad$
              $grad\_arr += grad$
              delete $grad$ in S3
              $numAdded$ += 1
          end for
          if $numAdded == len(workers)$ then
              break
          end if
      end if
      $time.sleep(2)$
  end while
  $optimizer.zero\_grad$
  for $param$ in $model.parameters()$ do
      update $param.grad$ with aggregated gradients
  end for
  $optimizer.step$
  $save\_weights(model, bucket, key)$
```

Task manager is one of the most important components of the workflow. It is responsible for data transmission and aggregation and making sure the main model is updated correctly with all of the gradients. In addition, since the child workers execute asynchronously and can finish in any arbitrary order, the task manager needs to efficiently checks and collects gradients without too much delay or computation overhead. Algorithm 1 above is a pseudocode for Task Manager.

Note that s3.listObjects returns the information of objects that are currently stored in the s3 bucket. We check periodically to see if there is any gradient file saved to the bucket from the workers. If one or more files is returned, we read and append the gradient to the grad_arr. The file is immediately deleted after read since we don't want to read the same file more than once in later epochs. The while loop terminates when we have gathered all of the gradient files from the child workers.

3 Experimental Setup

3.1 Dataset and Layers

The dataset used for training is MovieLens[2] which contains 1,000,209 anonymous ratings of around 3900 movies and 6040 users. The rating file is roughly 30.7MB and is stored on S3. In later section we also tested smaller subsets of the data to see the how the performance differs.

We added one layer of PyTorch version 1.1.0 for each function since the current version is too big to fit within the 250MB limit. We also used numpy, json and csv for processing data.

3.2 Function Setup

We used AWS Lambda as our serverless platform. The original code runs in a single lambda function. It loads data from S3 and directly trains the model. The implementation for data parallelism splits the function into 1+n functions where n is the number of child workers. The main function creates the main model and handles the main training logic as well as the distribution and aggregation of model weights and gradients.

4 Results and Discussion

In this section, we conducted experiments on using different memory limits, number of child workers, and dataset size. We compared the results by their execution time and resource usage. Note that all of the statistics are collected after the functions have been "warmed up".

4.1 Memory Size

In this section, we analyzed the effect of provisioning different memory size for the lambda function. We tested on one lambda function trained on 1 million input(full dataset) and compared the performance by setting the memory to 800/1000/1500/1580/2000 MB with 3 minute time-out. The most obvious pattern is

Table 1: Memory Size Testing

workers	mem_lim	mem_used	time_used
1	800 MB	N/A	N/A
1	1000 MB	1000 MB	99.148s
1	1500 MB	1500 MB	72.742s
1	1580 MB	1545 MB	69.972s
1	2000 MB	1546 MB	60.044s
1	3000 MB	1545 MB	52.611s

that bigger memory generally gives better runtime performance. As we can see from the table, even though the max memory used is around 1545 MB, the larger memory allocations still gave faster computation over sufficient memory allocation(1580 MB). However, if we under provision memory by 500 MB, we would need to train the model for 30s more. When we have 800 MB, the function simply timed out after 3 minutes.

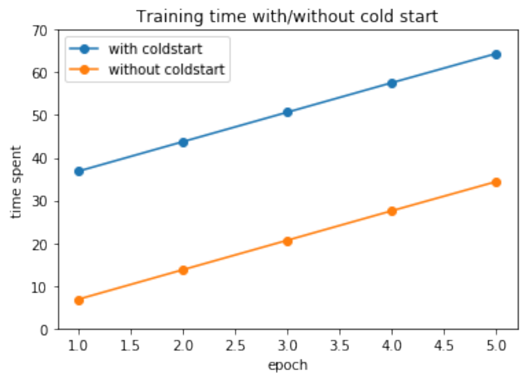
4.2 Number of Workers

Table 2: Number of Workers Testing

workers	dataset	mem_lim	mem_used	time_used
1	1m	1500 MB	1500 MB	72.742s
3	1m	1500 MB	662/958MB	35.222s
5	1m	1500 MB	662/839MB	34.685s
1	120k	1000 MB	757MB	25.709s
3	120k	1000 MB	657MB	12.614s

In this section, we compared training the model with 1/3/5 workers where the worker(s) split the training data equally. For instance, 1 million data for 5 workers would result in each worker training 200k data. Overall, we found that having 3 workers training in parallel greatly speeds up training and return diminishes with more workers. For 1 million dataset, 3 workers is 51.58% faster than using 1 worker. For 120k dataset, 3 workers is 54.26% faster than using 1 worker. As one can see, increasing the parallelization to 5 workers does not have any significant improvements over having 3 workers. We believe the communication and data loading overhead outweighed the benefit of more parallel training power. In addition, given the similarity of improvements between using 120k and 1 million datasets, we believe the the efficiency of parallel training can be scaled to much larger datasets.

4.3 Cold Start



Cold start is the problem when a function takes longer to execute due to long periods of idle time or that it has just been initialized. The Lambda service needs to find a space in its EC2 clusters to allocate the workers. This problem is also observed during our experimentation where the first epoch takes much longer to execute in the first run as seen in the figure above with 5 parallel workers. Once the parallel workers have been initialized and loaded, the rest of the epochs have almost a constant training time.

5 Challenges and Limitations

In this section, we listed some of the challenges we encountered when implementing the system as well as some of the limitation with the current design.

1. **Deployment Size Limitation:** Since the deployment package size can not exceed 250 MB when unzipped, It was difficult to upload the current version of PyTorch as it greatly exceeds the limit. The solution was to use an earlier version (1.1.0) and remove some unnecessary components to fit within the limit. In addition, functions can only have up to five layers.

2. **Package Compatibility:** We also attempted to import other packages such as pandas into the layers. However, the addition of pandas breaks the code due to some compatibility issue with numpy.

3. **Lambda Timeout:** Lambda functions time out after 900 seconds (15minutes). Therefore, we might need to monitor the function and preemptively save the model in case it runs overtime for larger workloads.

4. **Intermediate Storage and Communication:** Since lambda functions are stateless, we need to store the weights and gradients in intermediate storage. We want the communication to be low-latency and optimized for ML workloads. However, none of the existing solutions (S3, Redis, Pocket...) is specialized for these tasks.

5. **Lack of GPU support:** The growing paradigm of deep learning requires GPU to speed up model training. However, current serverless architecture does not support GPUs which makes training large models impossible. There needs to be better solutions for multiplexing GPUs and build corresponding tech stacks to integrate GPU training into the serverless architecture.

6 Related Work

Although using serverless architecture for model inference has been commonly used in practice, data intensive applications like model training are not quite as common and remain an active area of research.

In the paper Serverless Computing: One Step Forward, Two Steps Back [3], the authors evaluated AWS Lambda's performance for machine learning and showed that it suffers drastically from the data-shipping architecture of Lambda. In particular, the author configured TensorFlow and trained a neural network on 90GB of data to predict customer rating. Each function was allocated max lifetime and 640MB RAM. They compared the result with training the same model on an EC2 instance and found that EC2 was 21 times faster (22minutes v.s. 465 minutes) and 7.3 times less expensive than the former. One of the key differences is the that Lambda requires much longer to fetch data and run optimizer, and that it times out after 15 minutes. The bottleneck of loading training data and function initialization was also observed in my experiments, although it is not as obvious given we used a much smaller dataset.

A case for Serverless Machine Learning[1] was a study published by UC Berkeley researchers. In this paper, the authors analyzed the resource management problem in the context of Machine Learning workloads and proposed a serverless ML framework to automate the management. The authors achieved low time per iteration by utilizing two mechanisms: 1) efficient prefetching and buffering of remote minibatches. 2) communicating with data store whenever possible. Prefetching allows functions to get data from s3 during computation, thus reducing time wasted from data loading. The second point emphasized on the careful asynchronous communication with the data-store since the time to send gradient is negligible. This mechanism was also used in our gradient aggregation implementation as it frequently checks the intermediate data storage for gradients to reduce function idle time.

SIREN[7] is a asynchronous distributed machine learning framework based on serverless execution. The authors of SIREN introduced a scheduler based on Deep Reinforcement Learning to dynamically control the number and memory size of Lambda functions used in each training epoch. The result showed that SIREN can reduce model training by up to 44% comparing to ML training benchmarks on AWS EC2 at the same cost.

More recently in 2021, ETH Zurich and Microsoft Research introduced a new platform named

LambdaML[4] that analyzed the design choices for algorithm optimization and synchronization protocol on serverless platform. More importantly, it enabled a fair comparison between FaaS and IaaS and provided an analytical model for the trade-offs that must be considered when opting for serverless architecture.

7 Conclusion

In this paper,we ported a recommender system to the AWS serverless platform and rewrote the program to optimize performance. First we designed a workflow for data parallelism which exploits the concurrency and scalability of lambda functions. We also implemented a task manager which handles the asynchronous function execution of child workers and gradient aggregation. At last, we analyzed the model performance by experimenting with number of workers, memory size and dataset size. Our results showed that distributed gradient descent with lambda improves the training speed by 51% with 3 workers and 53 % with 5 workers. In addition, appropriate memory allocated to each function is also critical to the execution time.

In future studies, we could explore further various storage techniques such as Redis, DynamoDB, Pocket etc. since data communication constitutes a significant part of the training overhead with the serverless architecture. In addition, we will port larger models and datasets to the serverless platform and combine techniques such as model parallelization to accommodate bigger models.

References

[1] J. Carreira. A case for serverless machine learning. 2018.

[2] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), dec 2015.

[3] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *ArXiv*, abs/1812.03651, 2019.

[4] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang. Towards demystifying serverless machine learning training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, June 2021.

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Popa, I. Stoica, and D. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 02 2019.

[6] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, aug 2009.

[7] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[8] M. A. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.