Spring 5-24-2021

Machine Learning Using Serverless Computing

Vidish Naik

Machine Learning Using Serverless Computing

A Project Report

Presented to

Prof. Robert Chun

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Vidish Naik

May, 2021

The Designated Project Committee Approves the Project Titled

Machine Learning Using Serverless Computing

by

Vidish Naik

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

San Jose State University

May 2021

Dr. Robert Chun	Department of Computer Science
Dr. Fabio Di Troia	Department of Computer Science
Dr. Soon Tee Teoh	Department of Computer Science

Abstract

Machine learning has been trending in the domain of computer science for quite some time. Newer and newer models and techniques are being developed every day. The adoption of cloud computing has only expedited the process of training machine learning. With its variety of services, cloud computing provides many options for training machine learning models. Leveraging these services is up to the user. Serverless computing is an important service offered by cloud service providers. It is useful for short tasks that are event-driven or periodic. Machine learning training can be divided into short tasks or batches to take advantage of this. Due to the nature of serverless computing, there are certain limitations imposed by the cloud service provider such as execution time and memory. This research proposes standalone solutions to overcome the challenges faced by serverless computing in training machine learning models. The research further combines these individual solutions and proposes a system for leveraging serverless computing for training a machine learning model that incorporates distributed machine learning.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Machine learning and artificial intelligence have been the latest trend in computer science for some time. Machine learning enables computers to learn trends and patterns in data that are provided during the training phase. Computers are trained to make decisions on new incoming data based on past data. These decisions are made without having to explicitly code for the decision-making process and are based only on the knowledge gained by the computer during the training. The training process is time consuming and resource intensive. Resources such as GPU, RAM, and CPU are the main components that are needed for resource intensive training. Owning these resources individually is expensive.

Cloud computing has become a viable solution to tackle this problem. Cloud computing allows users to use resources for a fraction of the cost by leasing them for the required duration. As the adoption of cloud computing increases, more and more models are being trained on the cloud. As a result, researchers and developers are using more and more cloud instances that require constant monitoring and upkeep. It is also important to note that the cloud service provider is only responsible for the underlying hardware in such services. The operating system management, updates, patches, and other software responsibilities belong to the user. This also includes managing the libraries and their versions. Serverless computing is a type of service that allows the user to focus their efforts on developing solutions. Cloud service providers abstract the details of underlying hardware and software infrastructure and provide the user with a handful of options that are relatively easy to fine tune according to the need. This becomes useful for the developer as they can invest more

of their time into focusing on the problem rather than solving redundant problems related to the server. Additionally, serverless computing can be done in isolated environments. This can be leveraged to test multiple hypothesis in parallel which becomes an advantage for the developer.

1.1 Problem Statement

The goal of this project is to develop a solution to train a machine learning model using serverless computing. Using serverless computing sounds compelling but comes with its own set of challenges. Firstly, since the underlying hardware and software is the responsibility of the cloud service provider, the execution time of the code is capped at a few minutes. This adds challenges for training that exceed this time limit. Secondly, the amount of storage that this service provides is limited to Megabytes (MB). This makes the use of libraries such as PyTorch and TensorFlow, which are well known for machine learning and artificial intelligence, difficult to use as they occupy space that can be used for other purposes. Another problem that limited storage creates is that the trained model size can exceed the memory limit. Thus, solving these challenges is critical in the adoption of serverless computing to train machine learning models.

2. Motivation

Machine learning has been part of almost all possible applications involving computers. It has helped in forecasting prices of commodities [1] as well as predicting the possibility of cancer in a medical body scan [2]. These kinds of applications require sophisticated machine learning models that can accurately fulfill their desired goal. Not doing so can result in a big financial loss or lead to a loss of human life. As a result, developers and researchers who train these models must train and choose multiple models to identify the best performing model. Training multiple models requires compute resources that are expensive to purchase. The recurring cost of electricity and maintenance is costly too. As a result, it becomes difficult to test multiple hypotheses with such restrictions. Cloud service providers that own and manage these resources offer them in a variety of forms at a fraction of the original cost. The services offered by the cloud service providers vary in the responsibility distribution between them and the user. These services can be grouped under three categories namely, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

The services that fall under IaaS allow greater flexibility as the user has a wide variety of customization options that can be chosen. Leveraging the full potential of these services for practical applications is the responsibility of the users. Creating a virtual machine in the cloud is an example of IaaS. User has the option to choose the operating system, the disk space, CPU, and RAM size among other settings. PaaS allows the users to focus more on the application rather than the underlying infrastructure. The choices offered to the users are restricted to the language and the

compute capacity required to run the application. The cloud service provider manages the remaining infrastructure. An example of this service can be hosting services that allow the users to host websites on their infrastructure. The users provide the code required to run the application and the remaining part is handled by the hosting service. SaaS offers readymade software for the users to use. Users do not have to write any code or manage any infrastructure. Gmail is an example of SaaS where the users do not have to manage any mail servers, nor do they have to manage any infrastructure. They can directly send and receive emails through their accounts.

Serverless computing, or Function as a Service (FaaS), is a service that was designed for short-running tasks. These tasks would have been periodic, or event driven to be able to trigger the code. An application of serverless computing is the backend of a web application [3]. Web requests are event-driven and short-lived and perfect for such a use case. With serverless computing, one can integrate other cloud services and provide a robust application for users. However, the advantage of serverless computing is that it hides the server management from the user and allows them to focus on the task at hand like PaaS. This advantage is beneficial to developers who can focus on the application without worrying about server management. Bringing this event-driven property of serverless computing to training machine learning models is challenging. As discussed above, serverless computing brings time and memory challenges that need to be tackled, but otherwise seems to be a viable alternative to the mainstream training process.

3. Related Work

Ishakian et al. [4] have already tested the idea of using serverless computing in machine learning. Their approach used serverless computing only in the inference phase. Since the inference phase uses less computation power than the training phase, their study finds it is suitable to use serverless computing for serving inference requests although it faces the issue of cold start. Serverless computing services run in a container and usually have a latency associated with starting the container which is called cold start. Subsequent requests of the task reuse the same container reducing the latency and speeding up the process.

Feng et al. [5] have used serverless computing for training a neural network. Their approach uses a data-parallel approach to serverless computing. They divide up the data into multiple chunks and each serverless instance works on the set of data and updates the parameters accordingly. Their approach only involves training models that go beyond the time constraints of serverless computing. [6] shows the model that they chose. They train a Convolution Neural Network to classify images. The size of the model turns out to be only a few megabytes which is not a storage challenge even for serverless computing. Their approach uses a parameter server which is a serverless instance of its own that serves the parameters of the model to the worker serverless instances. To reduce the number of transfers of parameters, the authors combine some of the worker nodes to be the parameter server. However, in case one of the instances is a parameter server and exceeds the time of training then the remaining workers might not be able to send their updated values. This might result in a model that was not trained.

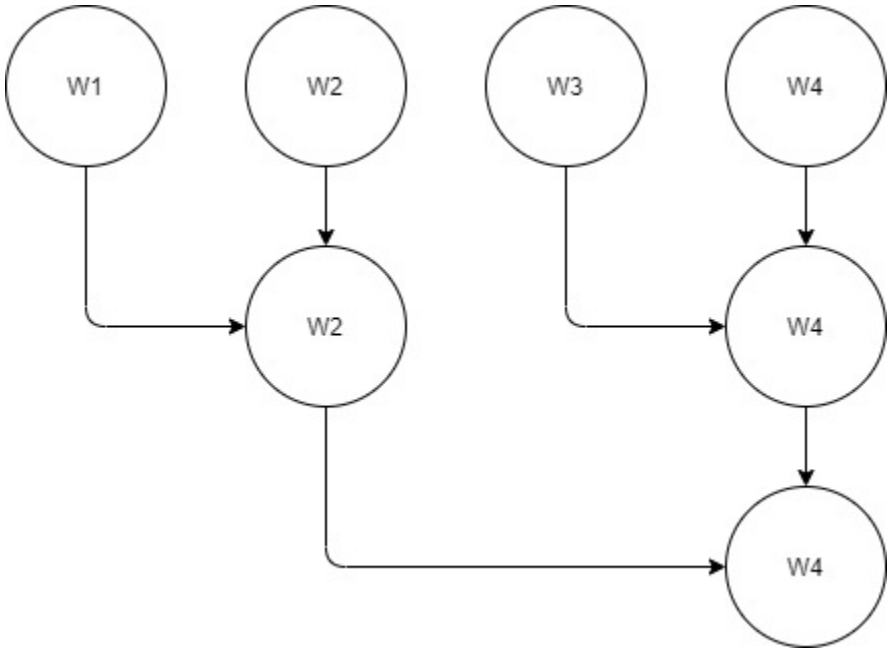

Fig 1. Understanding the approach of Feng et al [5]

Fig. 1 shows the approach taken by the authors. Here, W1, W2, W3, W4 are worker serverless instances that perform the computation on their respective datasets. After the computation is done, W2 and W4 act as the parameter server and accept updated parameters from W1 and W3, respectively. Once they receive the updated parameters, they update the parameters with the ones that they have calculated. Following this, W4 acts as the parameter server and W2 sends the updated parameters of W1 and W2 to W4 which has the updated values of W3 and W4(itself). After receiving the parameters, W4 updates the parameters and we have the updated parameters from all worker nodes. However, in this case, if either of the parameter server instances fails, the work will have to be done again leading to extra efforts.

4. Methodology

Successfully training a machine learning model using serverless computing will involve overcoming the challenges of memory and time constraints. These are the main challenges faced by serverless computing apart from which there is compute capacity. For this project, we will focus on solving this challenge on AWS Lambda [7] which is the serverless computing service offered by Amazon Web Services. The compute capacity of AWS Lambda increases with an increase in its memory configuration. Hence, having a higher memory serverless function might benefit from the higher compute power at its disposal. We address the service AWS Lambda as Lambda, with an uppercase L, and the individual functions in the service as lambda, with a lowercase L.

4.1 Storage

The challenge of storage arises from the fact that AWS Lambda only has 512 MB of non-persistent memory during runtime in the “/tmp” directory. Any data stored during runtime will not be available during subsequent executions. As a result, it becomes impossible to dynamically load the libraries such as PyTorch [8] or TensorFlow [9] during runtime.

4.1.1 Customizing AWS Lambda

Perez et al. [10] propose a solution to custom create a Docker image and upload it to AWS Lambda. This method allows us to load the required library and, if needed, the dataset in the image. This reduces the time and latency in fetching the

data as the data is locally available. However, with this approach, we also must add the AWS Software Development Kit (SDK) as well as the library that we will use for training. This also required the use of an AWS owned repository for container images called Elastic Container Registry (ECR) [11]. AWS ECR is like Docker Hub [12] in the sense that it hosts the container images created by the user.

Another approach we can take is adding layers to AWS Lambda. Layers are zip files that are added to the lambda function. The layers can hold the libraries in the layer. The contents of the layers are then available in the “/opt” directory from where we can use the libraries. With this approach, we can quickly develop the code and make minor changes without uploading large amounts of data for small changes.

4.1.2 AWS Simple Storage Service

Another approach will be to access AWS Simple Storage Service, more commonly known as S3[13]. It allows object storage which can be easily accessed by using AWS SDK. AWS SDK’s are readily available in all AWS Lambda runtimes. We can store the dataset in an S3 bucket and then retrieve it as and when needed. This allows us to free the space for the dataset. S3 also supports byte streaming which can be used to store the model. However, we will need to create a file or object for every worker and then have additional workers combine those files. This adds overhead to the process. S3 does not allow object locking where one process can update the model. Hence it becomes difficult to use the same object amongst all workers.

4.1.3 AWS Elastic File Storage

Alternatively, rather than having external storage, we can leverage the services offered by AWS to solve the issue of storage. As pointed out by Sindi et al. [14], we can extend the storage by using Elastic File Storage (EFS) [15] which is a network file system that can be mounted in AWS Lambda. EFS is fully managed by AWS and is scalable meaning that it is serverless and can grow and shrink according to the need of the user. AWS Lambda uses a mount point in EFS to allow the file system to be mounted. This gives lambda the required additional storage for the model training. EFS also supports file locking including both shared and exclusive locks. This allows multiple workers to update the file, while locking, without having to worry about the consistency of the file. As a result, we can update the contents of the model without having the overhead of combining multiple files into one.

4.2 Time

Another major factor affecting the adoption of serverless computing is the time constraint on the execution of code. Since the user is not managing the underlying server, the user does not have access to configure the time for which the code should run. This is done for security purposes such that any malicious code cannot run for long durations on cloud service provider managed servers. The cloud service provider, AWS in this case, does allow the user to configure the maximum time the code can be executed. However, that is limited to a maximum of 15 minutes.

4.2.1 Sequential Execution

We can use serverless instances to call one another and pass the current state of training. The caller instances pass the parameters of the model as an event and the called instance receives them and continues training from that point. Once the caller has successfully called the other function, it can be terminated. The called instance now has the responsibility of continuing the training. This can be continued till the result is achieved. This means that we will be training in serial fashion meaning one instance after another. As a result, we will take similar or more time depending on the overhead of calling instances successively. Fig 2. shows sequential execution of lambdas for machine learning.

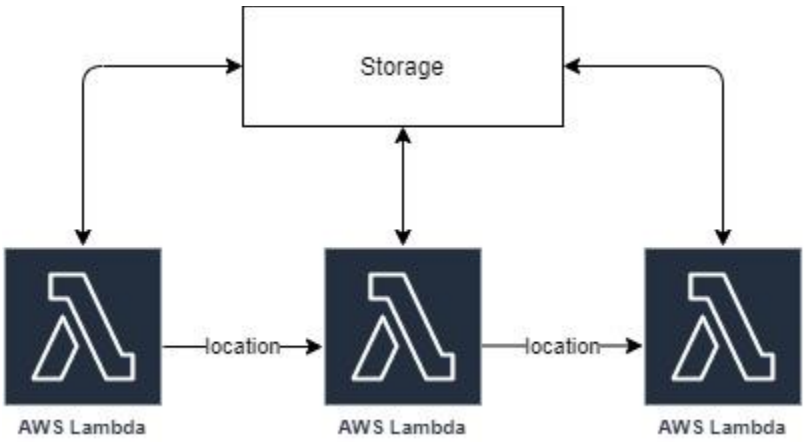

Fig 2. Demonstrating the sequential execution method

4.2.2 Orchestrated workflow

Orchestration systems build orchestrated workflows based on business logic. These workflows control the execution of FaaS services provided by the cloud service provider. AWS has an orchestration system called AWS Step Functions [16]. AWS Step Functions allows conditional execution as well as parallel execution of

AWS Lambda based on certain conditions. Lopez et al [17] compare the different orchestration systems for FaaS by their respective cloud service providers. Their study also finds that state can be transferred between instances up to 32KB. Models cannot be passed in such short memory constraints. However, we can store the model into an S3 bucket or EFS as discussed in section 4.1.2 and section 4.1.3 respectively. We can then send the location of the model while transferring the state. This can be achieved with the 32KB limit. Another point to be noted here is that the charges for using AWS Step functions might be steep when training large models.

4.3 Training Methodologies

Selecting the appropriate methodology for training is key to getting the result faster. Serverless computing poses challenges to traditional training methods. Traditionally, training a model involves all the data and required model attributes available to the training process. With serverless computing, based on the computing method, when the computing transitions from one instance to another, these attributes need to be replicated for the training to proceed. This poses a challenge to the serial mode of execution discussed in section 4.2.1. Also, the serverless computing instances are stateless, meaning that no information about the current execution will be retained by the next execution. Taking advantage of this property, we can focus on the distributed training of the model.

4.3.1 Distributed Training

Training of the model involves iterating over the dataset and updating the

weights of the model. The frequency at which the weights are updated depends on the algorithm that we are using to update the weights. Gradient descent is a popular choice for such problems. Gradient descent has three variants, namely Batch Gradient Descent (BGD) [18], Stochastic Gradient Descent (SGD) [19], and Minibatch Gradient Descent (mini BGD) [20].

BGD iterates over the entire dataset to update the weights of the model. It considers all the data points available in the given dataset before adjusting the weights. For large datasets having millions of data points, it takes a long time for one iteration. This process must be repeated over and over each time on all the data points to reach the desired result. As a result, this process is time-consuming. Considering the time challenge on serverless computing, the process of batch gradient descent might go beyond the permissible execution time.

SGD provides a faster way of updating the weights. In contrast to BGD, it considers each data point as a whole dataset and updates the weight after each data point. This gives instant feedback to the developer. It may seem like this approach is the best since it gives instant feedback and tunes the weights based on individual data points, but that is not the case. Since it is considering all the data points equally, it is also considering the outliers. Outliers are the data points that do not follow the general trend of the entire dataset. This allows outliers to distort the weights of the model and can harm the training process. Although SGD gives instant feedback, the outcome of BGD is better than SGD. SGD also proves to be difficult to implement in a distributed environment. Every update requires locks for updating the parameters which lead to overhead. It is found that the process of updating parameters with locks

slows down the process as processes end up waiting for the lock rather than doing actual computation. If a serverless instance ends up waiting for a lock, then it might cross the maximum permissible time and we might have to repeat the process. This adds extra work for computing and wastes resources.

Mini BGD takes the best of both worlds and combines them into one algorithm. It randomly samples the dataset into smaller groups. It can be noted that each smaller group will represent the entire dataset when divided at random. Each batch will now be processed as an entire dataset, and the weights will be updated after one batch rather than the entire dataset. The process, however, involves iterating over the entire dataset. This process helps reduce the memory requirements of AWS Lambda. The entire dataset does not have to be in the memory. Only the smaller batch that is currently being used can be in RAM while others can be in persistent storage. This allows serverless instances to work only on part of the data and can be executed within the permissible time limit. However, choosing the correct size for the mini batches becomes a trivial problem.

4.3.2 Parameter Server

Li et al. [21] have discussed the idea of having a centralized update and distribution of the model weights by using a parameter server. This server is responsible for updating the parameters when new parameters are received from the worker nodes and distributing the parameters when the worker nodes ask for them. Each server maintains a master copy of the parameters that it is responsible for and then also maintains duplicates of the parameters from other servers for fault

tolerance. In the context of serverless computing, we can have a serverless computing instance act as the parameter server and update and distribute the values of the updated weights. However, this approach means that we will always have a serverless instance running for that purpose. Instead, we can distribute the responsibility of updating parameters to the individual serverless instances. The instances do not have to propagate the weights as all the updates will occur on a central copy of weights that can be done via locks. The instances will read the parameters and then work on updating the parameter and update the parameters when the computation is done. While it is computing, it will release the locks and allow other instances to read or update the parameters.

Contrary to the general intuition of not allowing updates while working on the current data that is currently being processed, it is safe to allow updates to the model weights in machine learning. Niu et al [22] propose a novel idea of parallelizing SGD while not having locks when updating the weights. Their research shows that most updates to the weights of the model are sparse. The update only changes a set of parameters. As a result, their algorithm, called Hogwild, achieves results like the serial version of execution. Fig 3. explains their approach of using parameter server for training models.

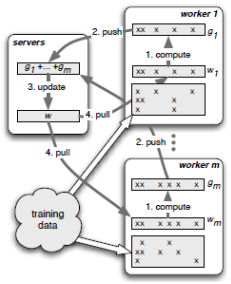

Fig 3. Explanation of training using parameter server.
Source: Niu et al. [22]

5. Proposed System

5.1 Architecture

Looking at the different approaches mentioned in section 4, we can see that some options are well suited for certain tasks, whereas other options are well suited for other sets of tasks. By combining the required properties of each approach that suit our tasks, we can achieve a system that is well suited for achieving the desired result. To tackle the issue of storage, we can create a system with a combination of EFS and Lambda. We eliminate the use of S3 as it adds an extra overhead of managing multiple workers. Additionally, S3 follows a mechanism of write once read many (WORM) meaning that data can only be written once. For making changes to the file, the entire file must be over-written. As a result, for solving the issue of storage, we will use a combination of customizing the lambda and adding storage using EFS. We will store the dataset, libraries, and model on EFS.

For tackling the time constraints, we can make use of a combination of serial and distributed workers. Initially, we start with a set of workers working parallelly in a distributed fashion. Each of them will work on their own set of data. Once the training process is done, the workers will store the gradients in a directory in the EFS. These gradients will then be picked up by the lambda function that combines the gradients and updates the weight. Once the weight is updated, the combining lambda again invokes the workers, and the process continues. The number of workers working in parallel will be decided by parameter values that we will pass at the start of training.

For the training methodology, we will be using a distributed training approach using minibatch gradient descent (mini BGD). We will distribute the dataset

amongst the number of worker nodes and each node will process and update the weight of the model. We plan to incorporate the portion of the parameter server into each worker node thereby eliminating the need for a standalone server. This reduces the networking overhead of passing and retrieving parameters and can be directly fetched from the mounted storage.

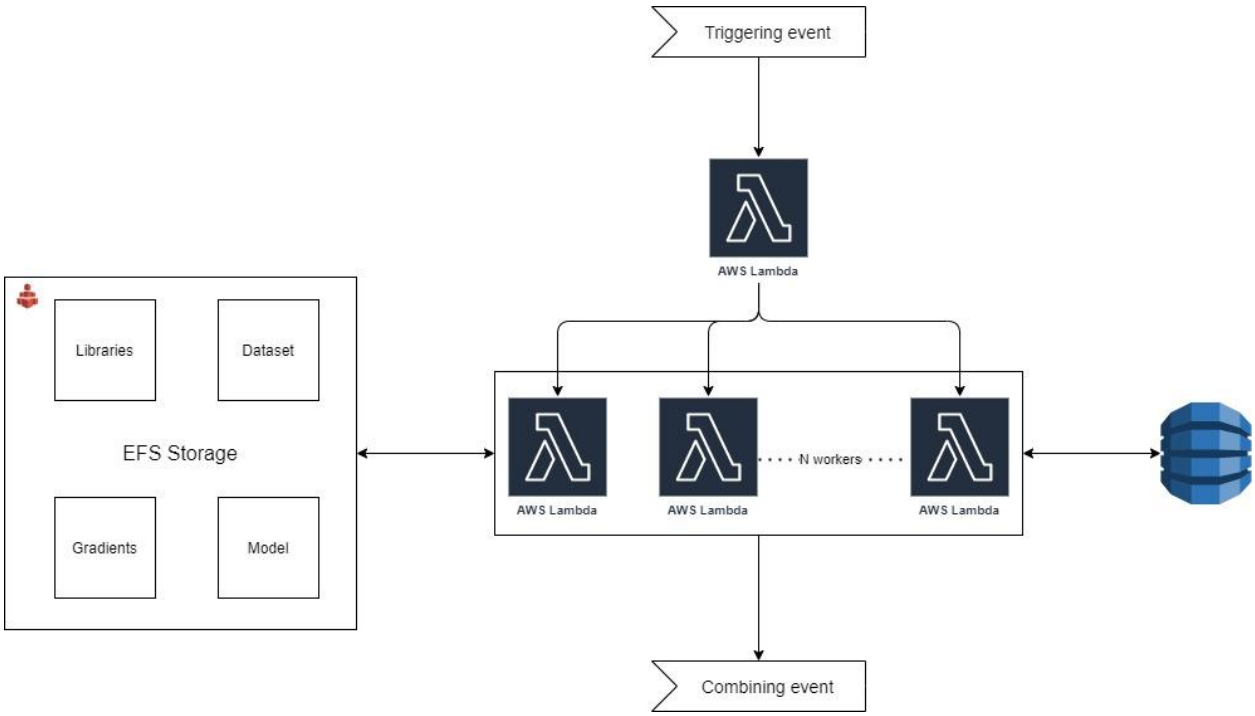

Fig 4 (a). Architecture for starting the training process.

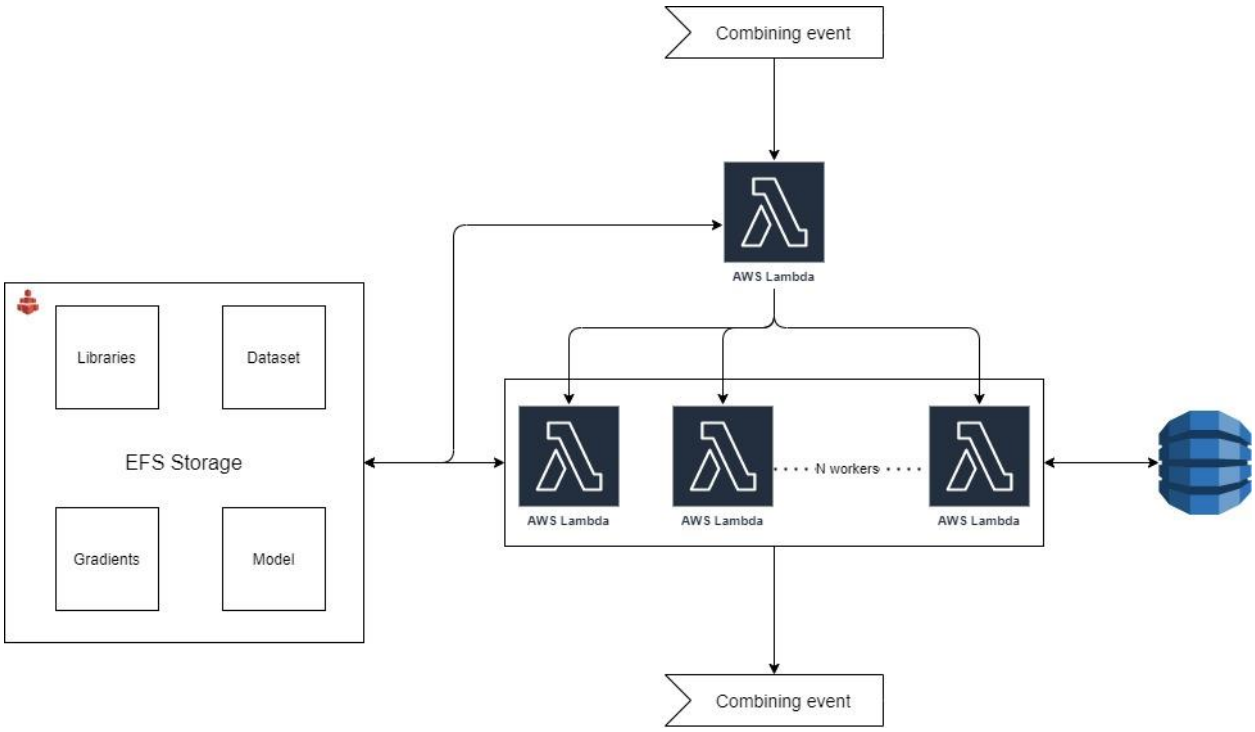

Fig 4 (b). Architecture for combining the gradients.

Fig 4 (a) and (b) show the architecture of the proposed system. Fig 4 (a) focuses on the architecture for initiation of the training process. Fig 4 (b) focuses on the architecture for combining phase and continuing the training process. The architecture consists of an AWS Elastic File System, Lambda, and DynamoDB. EFS stores the libraries used for machine learning, the data set required for training, the gradients, and the model while the training is in progress. The entire code for the lambda is written in the same lambda function. The same lambda function is re-used as a parameter server for calculating the gradient and combining the gradients once the calculation step finishes. Each lambda function updates the DynamoDB table to give updates to the user. Users can track the current state of the training process through the data inserted into DynamoDB.

5.1.1 Training

To start the training process, we need to invoke the lambda for the first time. The function takes an event to start, and we can provide this event manually through the console. This event is a JSON object and should contain the key “start” to begin the training process. Additionally, it should also contain the number of workers that need to be used for the training.

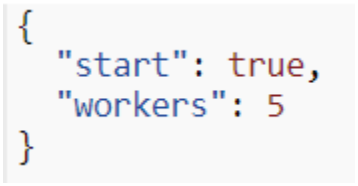

Fig 5. Initial invocation event

Fig 5. shows the invocation event for the training process. Once the lambda receives the event, it invokes the required number of workers and assigns each of them with a worker ID. Workers IDs are assigned from 0 to N-1 where N is the number of workers requested. To start the workers, we create an event in the initial lambda itself. The worker lambdas require their worker IDs in the event along with the total number of workers in the training process. We also pass the information regarding the number of epochs that are done.

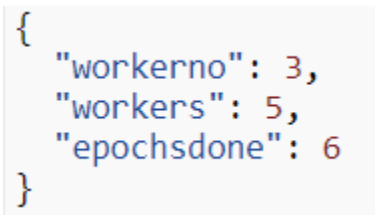

Fig 6. Worker invocation event

Fig 6. shows the invocation event for a worker. The worker lambda receives this event and starts by loading the data that is saved in the EFS. Initially, it loads the weights of the pre-trained model and then starts running the training on the dataset. It saves the gradients on the EFS for each step. Before finishing execution, the lambda function combines all its gradients to reduce the workload of the combining lambda. After the training is complete, only one lambda function initiates the combining phase. Each worker checks for the number of gradient files present on EFS. If they are equal to the number of workers, then it creates an event for the combining phase.

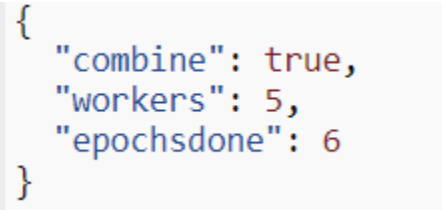

Fig 7. Combining phase invocation event

Fig 7. shows the event for starting the combine phase. The event phase contains the key “combine” to let the lambda function know that it is in the combine phase. In this phase, it sums all the gradients generated by the worker lambdas and then proceeds to update the weight based on the formula given by Li et al. [21]. The combine phase also determines if the training needs to proceed or can be stopped.

5.2 Setup

To setup the entire architecture, we use CloudFormation which allows us to

create resources in AWS using templates. Users can use the templates to deploy the same infrastructure in multiple accounts while ensuring that the architecture remains the same. Through the template, we create resources, install libraries and preprocess the dataset.

5.2.1 Resources

We need to create resources in addition to AWS Lambda and AWS EFS to be able to run our experiments. These resources are provided by AWS. While creating the resources, we keep security in mind and follow guidelines provided by AWS to have a secure environment. We also have Identity and Access Management (IAM) roles that permit only selected entities to access the filesystem. Fig 8. shows the full architecture diagram which involves other AWS services. The resources needed for this experiment are as follows:

- Virtual Private Cloud (VPC)

  VPC is a logical boundary for isolation of resources in an AWS account. VPC acts as a private network for resources to interact. Each user can have their VPC and resources inside them to allow independent work or work in the same VPC to collaborate.

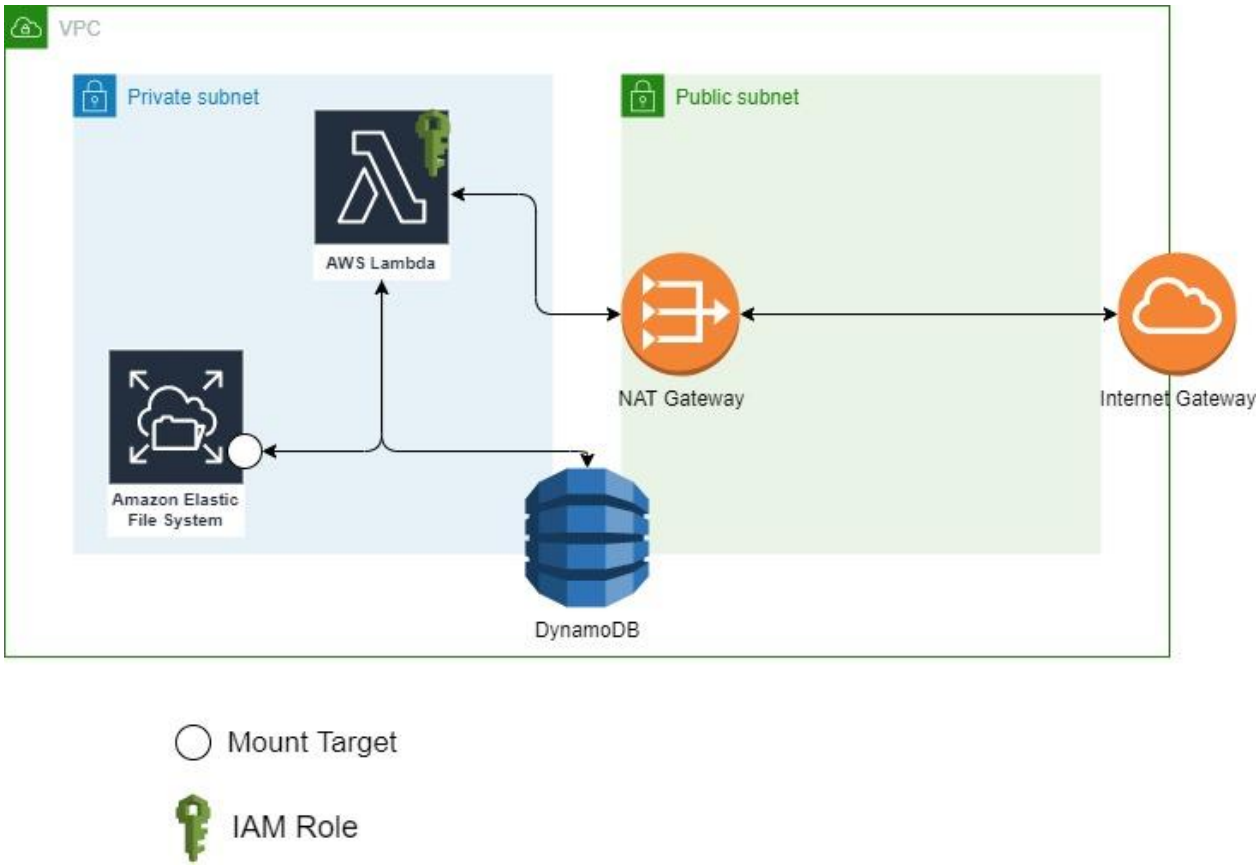

Fig 8. Full architecture diagram

- Subnet

  Subnets are virtual subnetworks within a VPC. These are smaller logical partitions of the VPC and are used by resources to interact. We will need 2 types of subnets:

  - Public subnet

    Resources in these subnets have access to the internet.

  - Private subnet

    These subnets do not have access to the internet. Internet access can be provided by routing the requests through a public subnet. Primarily EFS and Lambdas will be deployed in this subnet for security purposes.

- NAT Gateway

  Network Address Translation (NAT) Gateway is used to provide internet access to private subnet. It forwards the request to the destination while masking, or translating, the IP of the resource requesting it.

- Internet Gateway

  Internet Gateway provides the access to the entire VPC. It allows access to public subnets and the NAT Gateway to connect to the internet.

- Elastic File System

  The filesystem that will be used for storing the dataset, libraries, and the model. It will also store the gradients while the training is in progress.

- Mount target and security groups

  Mount targets are logical mounting points of EFS that allow other resources to mount the filesystem.

  Security groups act like firewalls and allow the whitelisted sources while denying any other traffic.

- DynamoDB Table

  DynamoDB table is used to monitor the progress of the ongoing training process. We can store the epoch information as well as the loss values.

- Lambda and IAM Role

  Lambda is used to train the model. Any lambda that is trying to access the filesystem requires permission to do so. IAM role has permissions that allow lambda to access the filesystem. To keep the architecture secure, we only allow Lambda to assume the role and no other service can use the role.

5.2.2 Libraries

While setting up the resources, we also install the libraries into the EFS for the lambdas to access. We install the libraries onto the EFS which allows us to go beyond the 512MB limit of Lambda. We can install popular machine learning libraries such as PyTorch [8] and Tensorflow [9]. These libraries can be shared amongst multiple developers through the same EFS thereby reducing dependency conflicts. All the developers use the same version and have a uniform development environment. The libraries installed for this project are mentioned below:

- Boto3 (v 1.17.39)
- Keras (v 2.4.3)
- Matplotlib (v 3.4.0)
- NumPy (v 1.19.3)
- Open CV (v 4.5.1)
- Pandas (v 1.2.3)
- Pickle (v 0.0.11)
- Pillow (v 8.1.2)
- Scikit Learn (v 0.24)
- SciPy (v 1.6.2)
- TensorFlow (v 2.4.1)
- Urllib3 (v 1.26.4)

5.2.3 Dataset

For this experiment, we use the CIFAR10 dataset [23]. The dataset contains 60,000 images divided into 10 distinct classes. Each class has 6000 images. The dataset is divided into 50,000 images for training and 10,000 images for testing. Fig 9. shows 30 sample images from the dataset.

Fig. 9 Sample dataset.

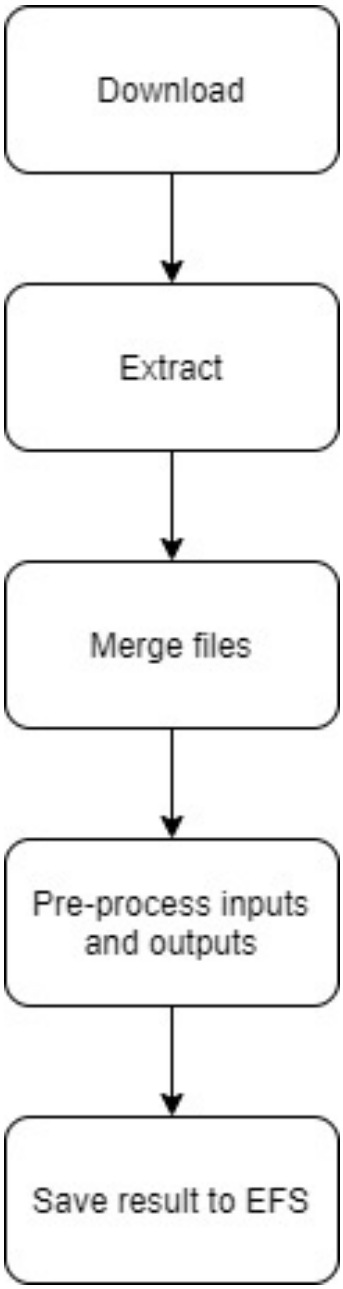

Fig 10. Preprocessing of the dataset.

Fig 10. shows the preprocessing of dataset. The dataset is downloaded and extracted in tar format to the EFS directly. The extracted data is divided into 6 files: 5 files for test data and 1 file for validation. These files are stored in pickle format and requires the pickle library for reading the data. To save time during each lambda

execution, we read all the data during the setup process and preprocess the data. Preprocessing includes resizing the images to fit the input of the model as well as one-hot encoding the output labels. The train and test images are then converted to NumPy arrays for more efficient storage. We then store the data in four separate files. One file each for training data, training labels, test data, and test labels. These files can be read faster and save time since the lambda must read fewer files each time. Table 1. shows the time taken to read files from untarred data and NumPy files.

Table 1. Time delay in reading files

	UNTARRED DATA (in seconds)	NUMPY DATA (in seconds)
RUN 1	4.07	2.04
RUN 2	3.43	1.76
RUN 3	4.80	2.17

6. Experiments

6.1 Requirements

6.1.1 Hardware Requirements

We use 2 different computing machines during the experiments. They are:

- Machine 1: AWS Lambda (RAM: 1GB – 10GB; 1 GB increments)
- Machine 2: AWS EC2: t2.xlarge (4 vCPU, 16 GB RAM)

6.1.2 Software Requirements

We use python3.7 as the programming language for the experiments. The libraries discussed earlier are installed using python's package installer pip. The libraries are installed on EFS and shared amongst all the machines to have identical training and testing environments to prevent any undue advantage.

6.2 Model

We use VGG19 [23] for experimenting with the proposed system. VGG19 has a trained model size of 549MB [24]. This size goes beyond the 512MB limit of AWS Lambda and is used for this purpose. We decide to perform transfer learning of the VGG19 model. Transfer learning takes more than 15 minutes which is the maximum permissible runtime of AWS Lambda.

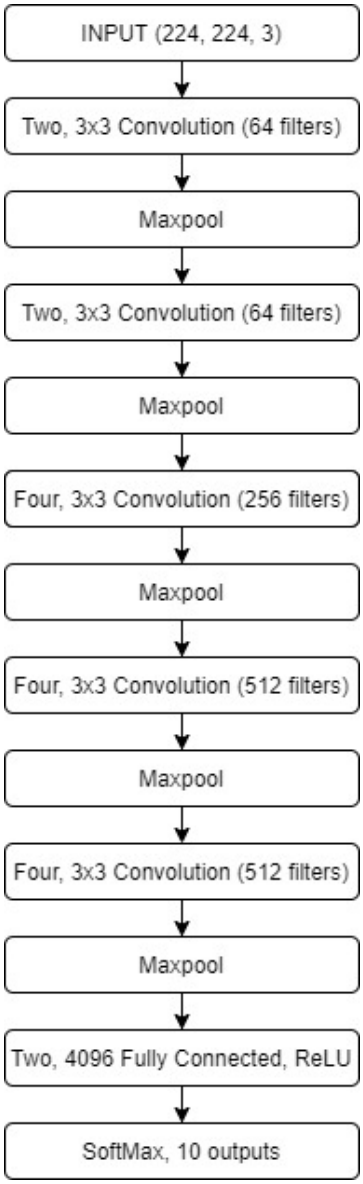

Fig 11. VGG19 [23] with new SoftMax layer.

Fig. 11 show the layers of the VGG19 model with the modified SoftMax layer. The input size is an image or array of dimensions (224, 224, 3). The first two values are the width and height of the image and the third value is the number of color channels. Here 3 indicates that there are three color values per pixel. The CIFAR10 input, for this experiment, must be resized from 32 x 32 to 224 x 224. The

images are already in three channels and no further modification is required here. The resized images are stored as NumPy arrays in EFS. After the input stage, there are two 3 x 3 convolution layers with 64 filters. The output of the convolution layers goes into a max-pool layer. Max-pool layer has a filter of size 2 x 2, and it generates an output of size 112 x 112 x 64 from input size from 224 x 224 x 64. This output goes into two 3 x 3 convolution layers with 128 filters and a max-pool layer to get 56 x 56 x 128 output. This is repeated for four 3 x 3 convolution layers with 256 filters with max-pool layer and twice for four 3 x 3 convolution layers with 512 filters and max-pool layer to get the final output from the convolution neural network part to get an output of 7 x 7 x 512. This input is flattened fed into a fully connected layer with 4096 outputs having ReLU activation. This is again passed through a fully connected layer before passing it through the final SoftMax classifier which classifies the input image into a category.

6.2.1 Transfer learning

Transfer learning is a method where a model previously trained on a dataset is customized to work on our dataset. The datasets must be similar in features for this approach to work. The pre-trained model that we use for our experiment is a model trained on ImageNet dataset [25]. ImageNet consists of more than 21,000 classes. For the CIFAR10 dataset [23], we need the feature extraction part of the pre-trained model. As a result, we swap out the final SoftMax classification layer with 1000 outputs and introduce a new SoftMax classification layer with 10 outputs which needs to be trained. We freeze the weights of the previous layers such that they are

not affected during the training process.

6.2.2 SoftMax layer

We swap out the SoftMax layer of the pre-trained model for a SoftMax layer that outputs 10 classes. SoftMax classifier converts the input into probabilities and then normalizes them. The normalized output lies in the range of 0.0 to 1.0 and the sum of all the outputs is 1.0. The output index with the highest value is the predicted class. We need to train this layer during our transfer learning so that we can predict the output based on the 10 classes that we have in our dataset.

7. Results

7.1 Lambda Configuration – Time

The first configuration that is adjustable is the runtime duration of the lambda function. The minimum runtime is 1 second and the maximum is 15 minutes or 900s. We keep the RAM at 10240 MB which is 10 GB to avoid any conflicts due to memory. We experiment the setup on 1-minute intervals to find the optimum runtime configuration. We set 20 steps per epoch for this experiment. Table 2. lists the parameters for the experiment.

Table 2. Parameters for finding optimal runtime.

Parameter	**Value**
Language	Python3.7
Memory	10 GB
Steps	20 per epoch
Epochs	5
Batch size	8
Time duration	1 min increments (Min: 1s, Max: 15m)
Number of workers	1

Table 3. Results for optimal runtime for AWS Lambda.

Runtime Duration (in minutes)	Epochs	Steps	Total steps
1	0	0	0
2	0	4	4
3	0	9	9
4	0	15	15
5	1	0	20
6	1	5	25
7	1	11	31
8	1	17	37
9	2	2	42
10	2	7	47
11	2	11	51
12	2	15	55
13	2	19	59
14	3	4	64
15	3	8	68

Table 3 shows us that we can achieve 3 epochs and 8 steps or 68 steps of training per execution. This is achieved with the parameter values in Table 2. For each additional minute added to the execution, we get additional four to five steps. To maintain a consistent epoch count, we will set the epochs per worker to 3 and leave additional time aside in case a batch takes longer than expected.

7.2 Lambda Configuration – Memory

The next configuration that can be changed for the lambda function is the memory configuration. The minimum amount of memory that a lambda function can have is 128 MB and the maximum is 10240 MB or 10 GB. We experiment with the setup on 1 GB increments and use the maximum memory used log provided by AWS for each execution of the function. We keep the runtime at 15 minutes to test the memory requirements during the entire process. We take the maximum number of epochs. Table 4. lists the parameters for the experiment.

Table 4. Parameters for finding optimal memory value.

Parameter	**Value**
Language	Python3.7
Time duration	15 minutes
Steps	20 per epoch
Epochs	3
Batch size	8
Memory	1 GB increments (Min: 128 MB, Max: 10 GB)
Number of workers	1

Table 5. Results of optimal memory value for AWS Lambda.

Memory (in GB)	Epochs	Steps	Total Steps	Time
1	0	0	0	Out of Memory
2	0	0	0	Out of Memory
3	0	0	0	Out of Memory
4	0	1	1	Out of Memory
5	2	16	56	15 minutes
6	3	0	60	14min 26s
7	3	0	60	13min 48s
8	3	0	60	13min 05s
9	3	0	60	12min 20s
10	3	0	60	11min 42s

From Table 5, we can see that any memory configuration above 5GB can give us training results for 3 epochs under 15 minutes. For 5GB memory, we can train the model, but we are not able to complete 3 epochs. We consider the 8GB memory option as an optimal memory configuration as we can get 2 minutes of buffer time in case, we were to exceed the observed time.

7.3 Number of workers

We will be running multiple workers in parallel to achieve faster training times. To get the most out of the parallelism, we need to decide the optimal number of workers. In this experiment, we will run multiple workers in parallel and look at the

total time required by setup to complete the training process. We start by running only one worker which is a linear approach where the lambda keeps calling itself linearly. We then increase the number of workers exponentially up to 32 workers in parallel. The epoch count here is the total of all the workers combined.

Table 6. Runtime vs Number of workers

Number of Workers	**Epochs per worker**	**Epochs**	**Time (in minutes)**
1	45	45	221
2	36	72	193
4	30	120	157
8	27	216	139
16	21	336	118
32	18	576	132

We can see from the observations that initially as we increase the workers, the time taken for the training decreases. However, as we go beyond 16 workers, the time taken starts to increase. For 32 workers, the time taken is almost equal to having 8 workers. Having additional workers does not benefit us in terms of time. As a result, the optimum number of workers lies between 16 workers to 20 workers.

The reason behind the training taking longer even though we have additional workers is the fact that the combine stage must collect gradients from additional workers and compute the sum on all the gradients. This process adds overhead in

the entire training phase and diminishes the advantages of having additional workers. This is because we have more than a million weights and each weight will have a corresponding gradient. Every worker generates these million gradients, and the combining function must collect these weights and process them before updating the weights. Fig 12. plots the graph of the results of table 6. It shows the time taken by the workers to complete the training process.

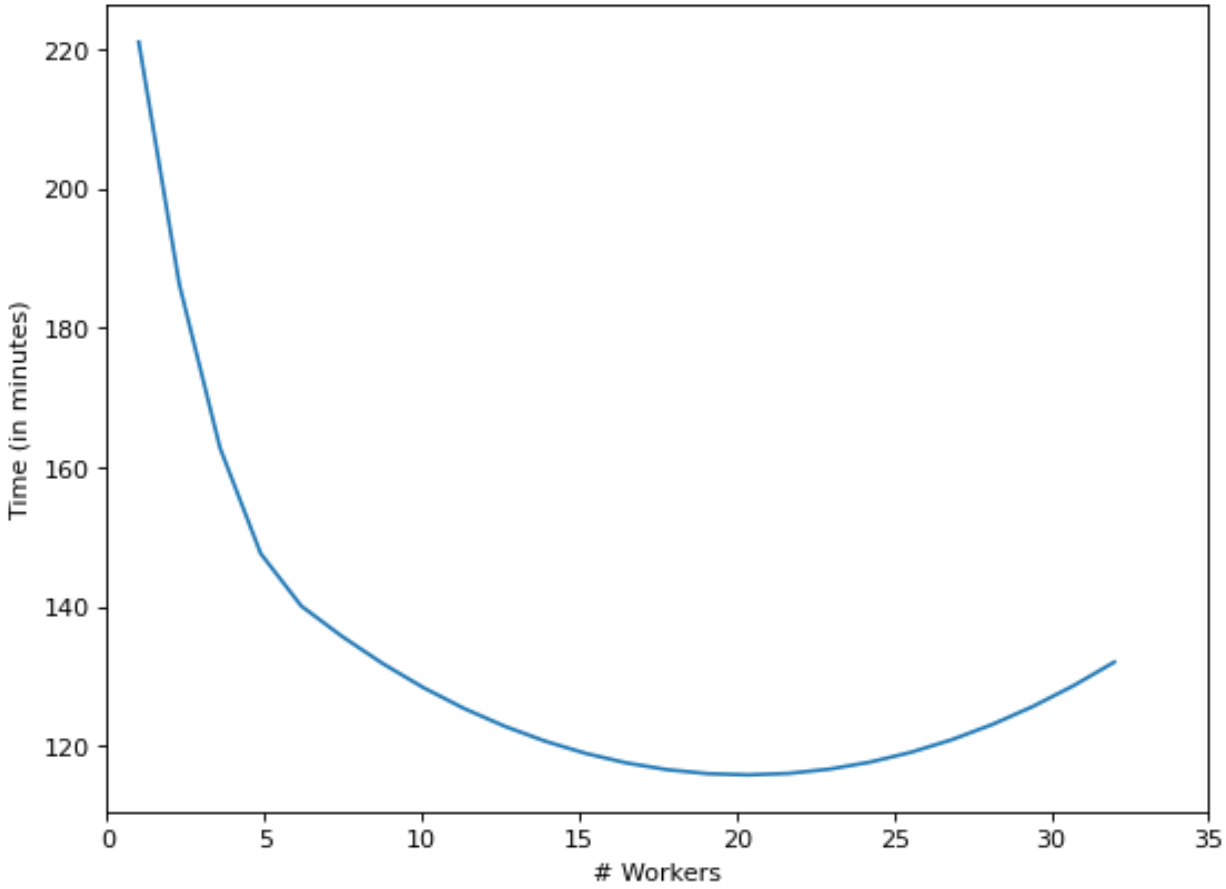

Fig 12. Compute time vs Number of workers

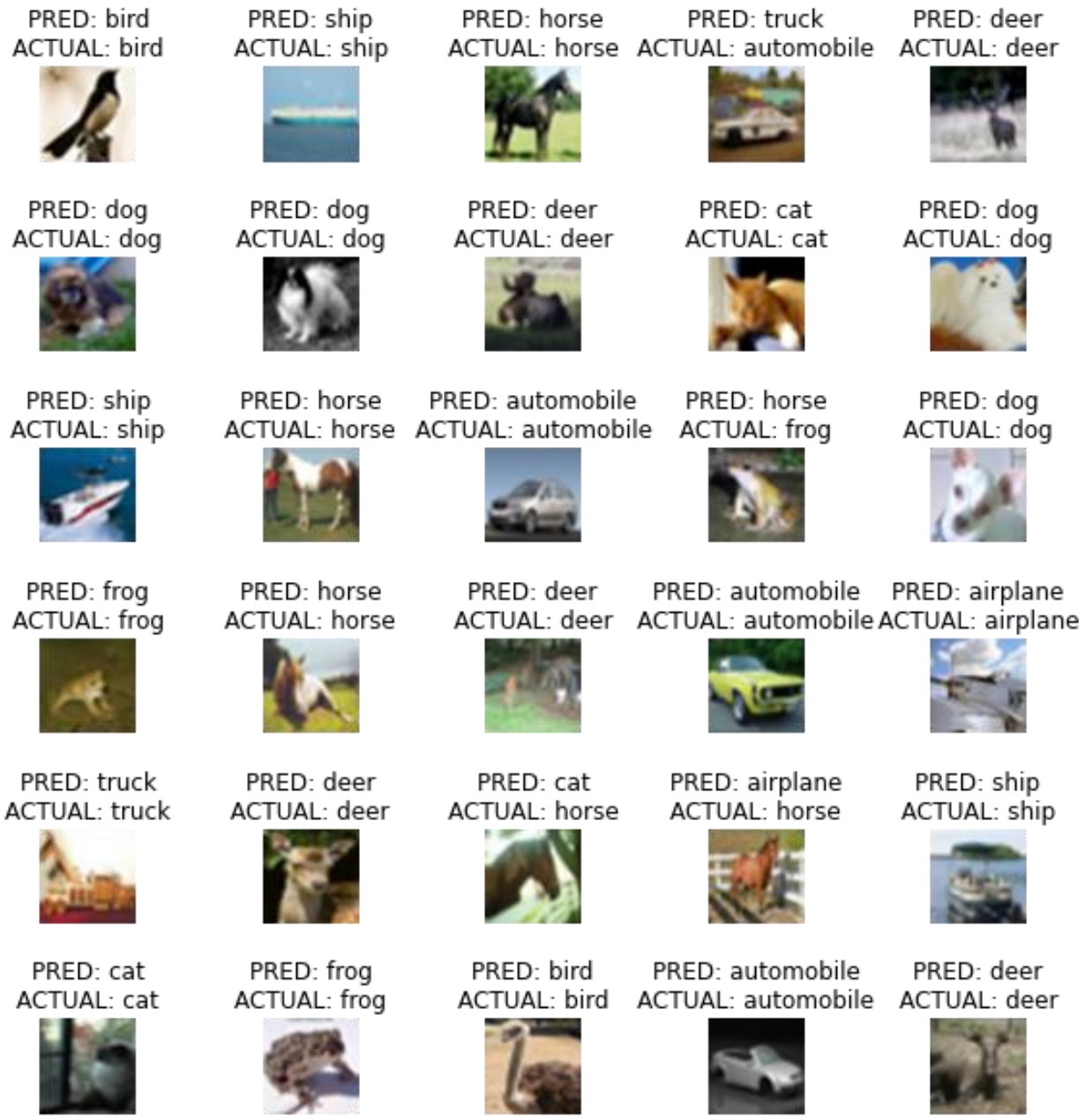

Fig 13. Predictions by the trained model

Fig 13. shows the prediction of the model trained by 16 workers. We can see that the model accurately predicts 26 out of 30 test images.

7.4 Comparison with traditional VM instance

For comparison with a traditional VM instance, we use an EC2 instance in AWS which provides virtual machines in predefined as well as custom configurations. We use machine 2 which has 4 vCPUs and 16 GB RAM. We train the model using the libraries installed on EFS. This allows us to have a uniform environment across the machines. We make use of the same code used on AWS Lambda to run on the instance with modifications to make it run on the instance.

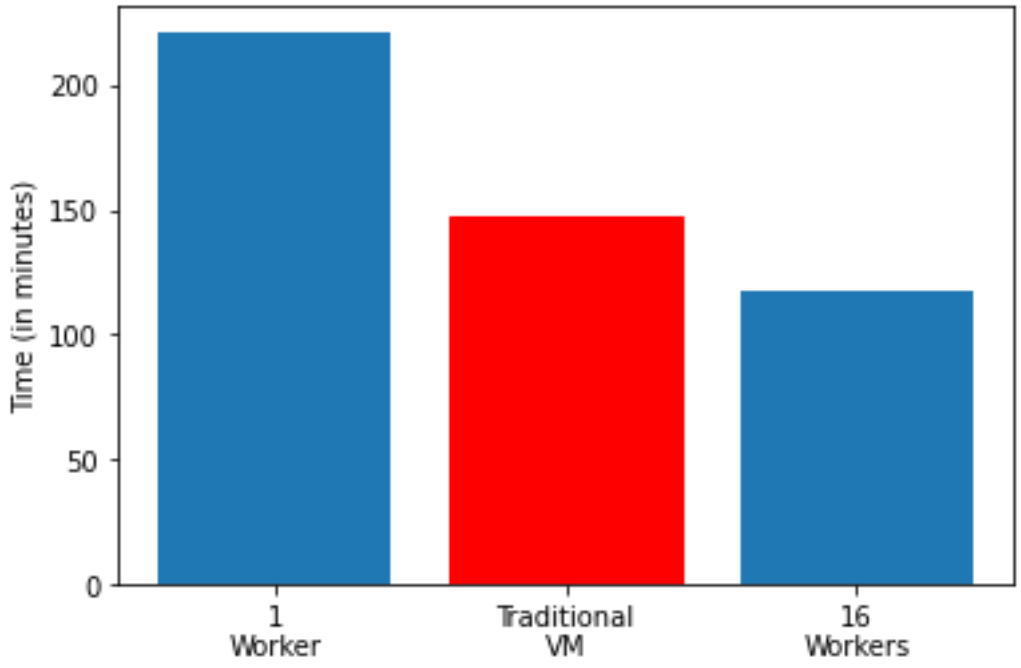

Fig 14. Runtime comparison with traditional VM

From Fig. 14, we can see that traditional VM is faster than a single worker lambda but slower that 16 workers working in parallel. Single worker lambda acts as a traditional VM but has overhead of combining stage making it slower than the traditional VM. For the lambda with 16 workers, the advantages of parallelism overcome the overhead of combining the gradients. The workers compute the gradients in parallel leading to faster training times. We can see that having more

workers results in faster training times than traditional VMs. However, there is a limit to the number of workers we can have before the overhead of having these additional workers outweighs their advantages.

The main bottleneck of the training in parallel is the combine phase. The combine phase must wait for all the worker lambdas to finish the computation and then only can it start processing the gradients. Additionally, the combining phase must read all the gradients and sum them up which delays the training process. As a result, the more workers we add to the training process, the longer is the wait time for the combine phase to start. Also, the combining phase must run longer to compute all the gradients. Hence, adding additional workers beyond a point leads to additional overhead that is not overcome by the parallelism of the worker lambdas.

Another constraint of the process is the synchronization of the lambdas to make sure we only have one event triggering the combine phase. It might happen that multiple lambdas satisfy the condition for the combine phase. If we have multiple lambdas triggering the combine phase, we might end up having more than N workers as each combining lambda will start N workers of its own. As a result, it becomes necessary to control the initiation of the combine phase and to make sure that one and only one of these lambdas is started.

8. Future Scope

Serverless computing has seen an increased demand in recent times. This goes to show that it has lasting consequences on the current workloads. We have seen in the experiments that serverless computing can be used for machine learning workloads. Given that machine learning primarily benefits from the use of GPUs shows us that serverless computing with GPU can be considered a possibility in the future.

ECS Fargate is another serverless service that provides a cluster of compute resources and uses containers to perform tasks. We can have machine learning tasks run inside these containers and the service can scale as and when needed. This allows us to eliminate the time constraint as there are no time limits for ECS.

9. Conclusion

The application of machine learning is increasing by the day. New models are being trained at a rapid pace and the need for having newer and faster methods of training is increasing by the day. Cloud service providers have a variety of services for computing. Serverless computing, although with its limitations, proves to be a vital service in the cloud domain. The limitations of serverless computing can be leveraged to our benefit and can be turned into a resource. The limitation of time and storage combined gives rise to a novel approach to distributed training. The architecture and the system proposed in this research can be leveraged to train large models that exceed the limitations of serverless computing for training machine learning models in parallel.

REFERENCES

[1] N. Amjady and M. Hemmati, “Energy price forecasting - problems and proposals for such predictions,” IEEE Power and Energy Magazine, vol. 4, no. 2, pp. 20–29, Mar. 2006.

[2] Y. Khourdifi and M. Bahaj, “Applying Best Machine Learning Algorithms for Breast Cancer Prediction and Classification,” 2018 International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS), Dec. 2018.

[3] N. Savage, “Going serverless,” Communications of the ACM, vol. 61, no. 2, pp. 15–16, Jan. 2018.

[4] V. Ishakian, V. Muthusamy, and A. Slominski, “Serving Deep Learning Models in a Serverless Platform,” 2018 IEEE International Conference on Cloud Engineering (IC2E), Apr. 2018.

[5] L. Feng, P. Kudva, D. Da Silva, and J. Hu, “Exploring Serverless Computing for Neural Network Training,” 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), Jul. 2018.

[6] “Convolutional Neural Network (CNN): TensorFlow Core,” TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn. [Accessed: 29-Mar-2021].

[7] R. W. Hendrix, “Lambda,” Amazon. [Online]. Available: https://aws.amazon.com/lambda/ [Accessed: 29-Mar-2021].

[8] A. Paszke et al., "PyTorch: An imperative style high-performance deep learning library", Proc. Adv. Neural Inf. Process. Syst., pp. 8024-8035, 2019.

[9] M. Abadi et al., "Tensorflow: Large-scale machine learning on heterogeneous systems", OSDI, 2016.

[10] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, “Serverless computing for container-based architectures,” Future Generation Computer Systems, vol. 83, pp. 50–59, Jun. 2018.

[11] “ECR,”. [Online]. Available: https://aws.amazon.com/ecr/. [Accessed: 29-Mar-2021].

[12] “Docker Hub.” [Online]. Available: https://hub.docker.com/. [Accessed: 29-

Mar-2021].

[13] “S3,” Amazon, 2002. [Online]. Available: https://aws.amazon.com/s3/. [Accessed: 29-Mar-2021].

[14] M. Sindi and J. R. Williams, “Using Container Migration for HPC Workloads Resilience,” 2019 IEEE High Performance Extreme Computing Conference (HPEC), Sep. 2019.

[15] “EFS,” Amazon, 2000. [Online]. Available: https://aws.amazon.com/efs/. [Accessed: 29-Mar-2021].

[16] “Step functions,” Amazon, 1975. [Online]. Available: https://aws.amazon.com/step-functions/. [Accessed: 29-Mar-2021].

[17] P. Garcia Lopez, M. Sanchez-Artigas, G. Paris, D. Barcelona Pons, A. Ruiz Ollobarren, and D. Arroyo Pinto, “Comparison of FaaS Orchestration Systems,” 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Dec. 2018.

[18] “Gradient Descent Algorithms,” An Introduction to Neural Networks, 1995.

[19] H. Robbins and S. Monro, “A Stochastic Approximation Method,” The Annals of Mathematical Statistics, vol. 22, no. 3, pp. 400–407, Sep. 1951.

[20] S. Khirirat, H. R. Feyzmahdavian, and M. Johansson, “Mini-batch gradient descent: Faster convergence under data sparsity,” 2017 IEEE 56th Annual Conference on Decision and Control (CDC), Dec. 2017.

[21] M. Li et al., "Scaling distributed machine learning with the parameter server", Proc. OSDI, vol. 14, pp. 583-598, 2014.

[22] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lockfree approach to parallelizing stochastic gradient descent," in NIPS, 2011.

[23] K. Simonyan and A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2014.

[24] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", 2009.

[25] “Keras Applications,” Team Keras, 2020. [Online]. Available: https://keras.io/api/applications/. [Accessed: 29-Mar-2021]

[26] J. Deng, W. Dong, R. Socher, L. -J. Li, K. Li and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", Computer Vision and Pattern Recognition 2009. CVPR 2009. IEEE Conference on., pp. 248-255, 2009.