Analysis of Machine Learning Models in AWS Lambda

Shreyas Setlur Arun
Department of Engineering, Computer Science and Mathematics
University of L'Aquila
L'Aqila, Italy
shreyas.setlurarun@student.univaq.it

***Abstract*—Serverless framework is one of the modern computing techniques. It is very famous in the software industry for the ease of use, dependability and the ability to deploy functions independent of external dependencies. Similarly software engineers are working on machine learning deployment using serverless platforms to study the behaviour, performance, usage of memory and many other parameters.**

The focus of this paper is to demonstrate the behavior of the machine learning models deployed using serverless technology. Some of the well known models have been used for the inference. In order to perform this experiment we have utilised the ML frameworks such as Tensorflow, Pytorch and Caffe. The ML models namely Resnet, MobileNet and Inception were used with the frameworks listed. The results of this experiment show that Pytorch outperforms Tensorflow in terms of response time by utilising the AWS lambda platform. However, Caffe framework outperforms all the other frameworks in all the aspects.

***Index Terms*—Serverless, Framework, Behavior, Performance, AWS lambda, S3, Tensorflow, Pytorch, Resnet,Caffe, AWS**

I. Implementaion Overview

In this section, we will discuss the overview of the implementation of the project. We will also discuss the technologies used in the deployment of the project.

The implementation is based on an architecture that is a fusion of both the Microservices Architecture as well as the Serverless Architecture. A microservice architecture is a variant of service oriented architecture and is based on deploying small independent services that perform a specific functionality of an application. The micro services aremade up of self-contained pieces of functionality designed according to the business requirements. They communicate with each other through interfaces. On the other hand, a monolithic application consists of centralised co-ordination unit which can be a single point of failure. Furthermore, a microservice can be independently maintained and updated without affecting other functionality whereas, in a monolithic application this is not possible. Micro services are adaptable to various environments using containerization, cloud native applications and serverless computing whereas monoliths have to be reconfigured to be deployed in various environments.

Similarly, a serverless architecture is an style of designing and deploying applications using different cloud based technologies without actually working about server configurations. The applications deployed runs on servers but the headache of provisioning, maintaining and scaling is taken care by the cloud service provider. Serverless functionality is commonly known as 'Function as a Service'(FaaS) in the software industry. Another major advantage of using serverless technology is that the cost factor. It follows a pay per use model where payment is made as per the usage. The other factor contributing to the growth of this style of architecture is that the base operating system functionality is taken care by the cloud provider. Some cloud providers provide Platform as a Service (PaaS) which is very similar to FaaS but the need for scaling the application is needed to be configured manually. Serverless computing is different from traditional computing basically in the management and operation of servers.

Taking inspiration from the above architectures, we have proposed our high level architecture as shown in figure 1. The cloud service provider used in this experiment is Amazon Web Services (AWS). The implementation of the experiment by utilising the lambda functionality in AWS. We have worked with AWS lambda to implement the serverless functions. AWS lambda is a computational service in the Amazon cloud which provides capability for running code without the need for provisioning or maintaining servers. It is very efficient and provides the ability to easily to orchestrate complex architectures.

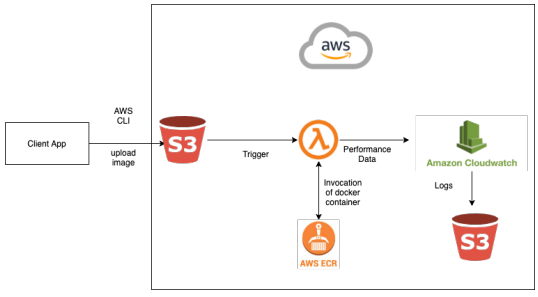

Fig. 1. High level architecture

The proposed architecture consists of a python based 'Client App' function. It is a function which is developed using python and it's functionality is to upload images to the S3 bucket associated with the lambda function. The function is based on a asynchronous library that uploads images without waiting for a response.

Amazon Simple Storage Service (S3) is a storage for cloud

based applications. It is used to store data from various sources for usage in different applications. It is preferred in the Amazon cloud services as it is cost effective, can retrieve any amount of data and at any time using a simple web interface or a simple library from Amazon. In this experiment for each model there is an Amazon S3 bucket associated with it. Once the client app uploads an image, the S3 buckets triggers the lambda function.

The code is organised into functions which are called as Lambda functions. These functions give the user the capability to manage only the code and manage the rest of the parameters like CPU usage, memory requirements and other resources themselves. Variety of languages can be used to write the lambda function. In this experiment we have used the Python language . The code has been dockerised in our experiment. The docker container consists of the code, the libraries necessary and the recent version of AWS python image. The docker container has been stored in the Elastic Container Registry (ECR). The process of dockerisation of the code is shown in figure 2. ECR is a container registry present in AWS for the storage, retrieval, sharing and management of docker containers. It is highly available and supports deployment of high level architectures efficiently.

Each of the lambda functions used in this experiment consist of a specific Machine Learning (ML) model which is used to perform an inference of the image. Each upload of image of S3 triggers a function call to the corresponding lambda function to make the analysis. The lambda functions works in the following manner:

- Series of images are uploaded to the S3 bucket associated with the ML function.
- The image is parsed and downloaded using the boto3 library in the lambda function
- Once the image is passed, the pretrained model is loaded to the function to perform the inference.
- Inference is performed on an input image resulting in a prediction vector.
- The process of loading, prediction and inference take place in an asynchronous way.
- Along with the above listed processes, the execution time for each ML model is recorded in a parallel manner.
- The time taken for loading thee ML model is also recorded similarly.
- Furthermore, when the execution process is completed, the logs are recorded in Cloudwatch for further analysis.
- The Cloudwatch lambda insights provides insights based on the logs generated by a lambda function.
- The logs are exported to S3 in the form of CSV files for future uses.

Another important aspect is the IAM role associated with each functionality in AWS. It gives the user the permission to use functionality and resources for any of the cloud based services. We have configures an IAM role with each of the lambda function. Cloudwatch Application Insights is another service used in this experiment. It provides specifc insights on the usage of memory, execution time, response time of a function and more information. In this experiment we have activated the cloudwatch lambda application for each lambda function.

Cloudwatch is a monitoring service which helps developers, deployment engineers, Devops engineers, managers and IT managers to visualise and analyse logs of an application deployed in AWS cloud services. It provides data and actionable insights for an application. The ability to monitor, analyse, optimise and collect system wide or application based insights plays an important role in the cost management of an application. Cloudwatch even provides actionable insights on system parameters such as CPU usage, memory usage as well as many other insights. It is used in our experiment to generate the logs of the lambda function.

Overall, the architecture consists of the client side application which inserts images asynchronously in a S3 bucket associated with a lambda function. The server side components consist of the lambda function, the IAM roles associated with each function, the S3 buckets and the cloudwatch application insights. The next section describes the experimental setup used in the experiment and the lambda functions in detail.

II. EXPERIMENTS AND EVALUATION

This section illustrates the details of the experiments performed for the study and outcome obtained from these experiments. Several research questions regarding the experiment have been analysed based on the results. Firstly, we discuss the experimental setup and deployment of components. Later, we will discuss the details of the evaluation candidates that have been selected for this experiment. Finally, we will discuss the research queries and visualizations.

A. Experimental Setup

The high level architecture discussed in section I consists of four components namely the client app, the S3 bucket, the lambda function and the Cloudwatch setup. In addition to these components the IAM role, ECR and the configurations of the function are also discussed in this section. We will discuss the role of each of these components in relation with the experiment.

Coming to the core part of the experiment, the code for the inference of pretrained ML models is written in Python and is dockerised along with dependencies of AWS lambda. it is then pushed to the repository created in ECR. There are a total of 8 scripts written for this experiment. Initially, the scripts were written in the local computer and tested if they are working. Why only 8 scripts? This is due to the fact that we have employed three deep learning models **Resnet, MobileNet and Inception** to perform the inference using three machine learning frameworks **Tensorflow, Pytorch and Caffe**. More details on these frameworks are given in the section II - B.

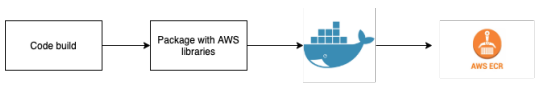

Fig. 2. Containerisation of the code

Figure 2 represents the process of containerisation of the code. The significance of this step is to overcome the limitation imposed by AWS lambda in terms of size of the layers or the libraries. The size permitted for layers in 256MB. As the models and the frameworks used in this experiment exceed this limit, containerisation assists in overcoming this drawback.

The client side application consists of python script whose sole responsibility is to send images to the S3 bucket periodically. We have made use of the 'asyncio' library in Python to time the uploads asynchronously to the S3 buckets associated with a lambda function.

S3 buckets are a form of public cloud storage which are economical and can store various forms of data. In our experiment we are using S3 buckets for storing the images used in the inference of the machine learning models as well as storing the logs of the experiment.

Moreover, for the simulation of the experiment, we have made the client side application to follow a FIFA 1998 World Cup trace logs. The trace logs are obtained from the FIFA 1998 web site [5] through the Internet Traffic Archive (ITA). The simulation is performed for 5 minutes and the following number of images are uploaded **[12,11,11,10,10]**. These values mean that 12 images are uploaded in the first minute of the simulation, 11 images in the second minute of the simulation, 11 images in the third minute and so on. The number of uploads can take place randomly within a minute within the given range. The image upload takes place in an asynchronous manner as stated before itself. A total of 54 images are uploaded for inference for all the models deployed in this experiment.

Furthermore, AWS lambda is a serverless compute service that lets a user run a code without provisioning servers, creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes. The advantage that lambda provides is that it can be used to run any type of code; either frontend based application, a backend based application or a simple machine learning based application.The administrative overhead of the user is greatly reduced by using lambda functions. There are various types of code deployments which can be done in a lambda function. It can either be zip file based, a code typed in the lambda editor itself or a container tool based deployment. Another plus point is that the application can be scaled or descaled instantly depending on the load. The parallel execution of the code is a major boost to application developers. Coming to the cost factor of lambda, it follows a pay per use model which is very economical compared to traditional deployment of server based applications.

The lambda component in the above architecture makes use of the docker environment for executing the ML models using the above specified frameworks. Each lambda function is an independent execution of a specific model using a specific framework. No lambda function depends on the other in our experiment. There are a total of 8 lambda functions deployed for this experiment. Each lambda function is triggered by the input of an image to the S3 bucket associated with it. While the deep learning model loads and performs the inference of the image the model execution time for the inference is recorded for analysis. The time taken to load the model is also recorded. This concludes the functionality of the lambda function. Lambda functions can be called using a web application, a mobile application or even a simple code.

Finally, the logs of the experiment are analysed and the graphs depicting the execution time of the models, the cost incurred for the execution as well as the model loading time for each of the lambda functions studied in this experiment.

In conclusion, this section describes each component along with its functionalities and the details of the deployment. Client side scripts for uploading images are deployed on the local computer. Serverless lambda functions perform the inference of the images by making use of the deep learning framework and the ML model. Analysis of the logs is done using python scripts and the graphs are generated.

B. Evaluation Candidates

In this section the focus is on discussing the baselines of the experimental candidates. We will discuss the candidates, their types and how we have employed these candidates in the experimental setup.

Three different experimental candidates are used for the evaluation of the deep learning models employed in this experiment. The idea is to know the depth of complexity of the model. The models selected in this experiment are from the domain of computer vision. Visual recognition is one of the most utilised research domain in the field of machine learning and deep learning community. These models are used all over the world for various domains of research and development. The deep learning models that are employed in this experiment come from the recognised **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** [7].The idea of this challenge is to encourage researchers to be able to develop various algorithms, machine learning as well as deep learning techniques that are able to recognise a large number of objects that are viewed commonly. This challenge is held every year and improvements to the database are made periodically. We have used the labels of the Imagenet database as part of the image detection in our experiment. Each of the lambda functions load the Imagenet labels to a variable numpy array and the detected object is named from these labels.

Let us now discuss in brief regarding the neural networks. These are set of algorithms that analyse data and interpret them by recognising the relationships in the data the same way a human brain works.The 3 main classes of the neural networks are:

- **Convolution Neural Networks (CNN's)**
- **Recurrent Neural Networks (RNN's)**and
- **Artificial Neural Networks (ANN's)**

In our experiment we have utilised some of the algorithms belonging to the class of **Convolution Neural Networks (CNN's)**. A CNN belongs to the class of deep learning algorithm that takes an input image, assigns importance (weights/bias) to it and then is able to differentiate the image. We have used 3 CNN based deep learning algorithms in

this experiment. The following subsection discusses these algorithms.

1) ResNet50: : ResNet was one of the groundbreaking discovery that assisted researchers in training and deployment of image related algorithms. It was discovered to overcome some of the drawbacks of Alexnet, the VCG network as well as the GoogleNet. The major drawback of the above listed algorithms was the vanishing gradient problem. The increase in the depth of these models was resulting in the accuracy getting saturated and increasing the layers of these models resulted in training error. These factors led to the development of ResNet by 4 researchers of Microsoft in the ILSVRC and COCO 2015 competitions. Theses researchers used the concept of **Residual Network Architecture** to solve the issue. The architecture consists of residual units called identity connections. Identity mappings identify the shortcut connections from one layer to another. ResNet has many variants associated with it. ResNet50 [2] is one of the very well known algorithms in the computer vision community used for image recognition and classification. We have used this variant which consists of 50 deep learning layers. The advantage of using ResNet architecture is that the residual mappings are fitted into layers. The layers are non-linear and the mapping is done in such a way that one layer is able to connect to another layer in a non linear mapping pattern. This process is known as residual learning and shown in figure 3.

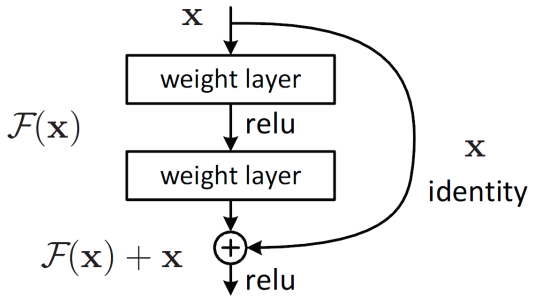

Fig. 3. Residual map of ResNet Architecture, adopted from [2]

2) InceptionV2: : Inception architecture [9] was created to standardize the kernel size for the detection of large scale variation in the salient features of images. The architecture of Inception is different to other CNN architecture as it is highly engineered and has had several improvements over the years. The idea of having multiple filters operating at the same level is the distinct feature of the Inception architecture as depicted in figure 4.

Some sections of images have very large variations. For example, an image of a dog can be centered in an image, it can occupy most of the part of another image as well as occupy a very small part of an image. These kinds of variations play a huge role in the selection of right convolution filter size. Another factor taken into consideration is the distribution of data. For worldwide data a large filter is preferred, whereas for locally distributed data a smaller filter is suitable. Inception architecture handles this issue seamlessly using variations

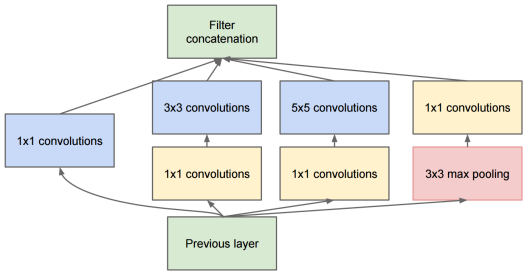

Fig. 4. Naive Inception Architecture, adopted from [9]

of the architecture shown in the figure 4. This architecture makes the network wider rather than deeper. A basic Inception module called Naive Inception Module handles convolution by introducing different filters of various sizes i.e 1X1, 3X3, 5X5. At the end of the module, the max pooling is done using the 3X3 max pooling filter and the output is concatenated. However, there is a problem of **representational bottleneck** which was solved by Google by introducing the Inception V2 architecture [10].**Smart factorization** methods were introduced by Google to increase the computational complexity. These methods distribute the filters having the size 5x5 into two 3x3 sized filters which boosts performance. Moreover, the filters with size nxn are reduced to a combination of 1xn and nx1 sized filters. This method leads to a reduction in representational bottleneck across the network.

3) MobileNetV2: : MobileNet architecture [3] was designed by a group of researchers at Google for optimisation of neural networks in mobile devices. **Depthwise** and **Pointwise** convolutions make up layers of the MobileNet architecture. This architecture follows a widthwise expansion of layers rather than a depthwise expansion.

Firstly, the MobileNetV1 version consisted of depth wise separable convolutions that reduced the model size and complexity. Batchwise normalisation and ReLU approximation are applied after each convolution. These models were deployed in mobile based applications and embedded vision applications. The depth wise convolutions applied a single filter per image input. The point wise convolution then performed 1X1 convolution thereby introducing new features in the network.

Secondly, a majorly revamped version called MobileNetV2 [8] was released to improvise MobileNetV1 and it introduced the concepts of **inverted residuals** and **linear bottlenecks**. The architecture contains 2 Inverted residuals blocks of different sizes. They aid in the removal of non-linearity in the layers. For each type of block, there are 3 layers in the model. However,the layers are inverted. The first convolution layer is of size 1X1 with ReLU6 activation while the second layer is depth-wise convolution layer. The third layer is also a convolution layer but without any non-linearity i.e it does not contain any ReLU activation. This model is also used for classification of images and feature extraction in various small devices.

C. Evaluation Metrics

This section illustrates the evaluation criteria used to measure the performance of the machine learning algorithms mentioned in the previous section. The core of this study is the evaluation metrics. We have used the evaluation to explain the results obtained in this experiment.

Firstly, let us discuss the frameworks that have been used in this experiment. Later we will discuss the performance measure of each of the ML model in each of the framework.

Machine learning has a large number of frameworks for various applications. Many of these frameworks are available open sourced whereas some of them are licensed. Each of the frameworks has its own advantages and disadvantages. Some of them are very easy to learn and deploy ML models whereas others are robust in their capability and performance. One of the aims of this study is to analyse and monitor the performance of various machine learning models deployed on various machine learning frameworks. Therefore, it is important to discuss the frameworks used in this experiment before the discussion of the performance metrics. The following frameworks are used in the experiment:

- Tensorflow: It is an open-source framework developed by Google for deep learning applications [1]. It is a platform independent as well as a language independent platform. It supports traditional machine learning techniques as well as modern deep learning methodologies.
- Pytorch: It is an open source machine learning library based used for computer vision as well as natural language processing applications [6]. Pytorch framework was developed by Facebook's AI Researcher Lab (FAIR) based on Torch library.
- Caffe: It is a deep learning framework written in C++ and having a Python interface [4]. It was developed by the University of California, Berkley for image classification and image segmentation purposes.

Secondly, let us now discuss the performance metrics of this experimental study. The study focusses on the follwing metrics:

- **Lambda Function Response Time**: Response time is one the most important parameters measured in any study related with machine learning models and frameworks. In order to analyse the performance for a lambda functions the **response time** is recorded. The overall time taken by a lambda function from the request to the completion of the execution constitutes the response time. It is the time recorded from the request to the completion of the execution in short. In other words, the time taken by a lambda function to execute and generate the response is taken as the response time. This measure is an important metric that lets us know the speed of the lambda function under the implication of the machine learning model which is being executed with a given and takes time depending on the ML model used and the deep learning framework used in the function. The experiment was performed to understand the flow of the machine learning model using a server less framework. As discussed in the earlier section FIFA 1998 World Cup logs are used in this simulation on lambda functions. The average response time is based on the response time received from the lambda function. The average response time is calculated using the following equation:

$$\frac{\sum_1^n R}{n} \tag{1}$$

where R = Response Time
and n = Total Number of Requests

- **Machine Learning Model Execution Time in Lambda Functions**: Execution time of the machine learning model is another performance metric considered in this experiment. The calculation of execution time is very simple. It is the time taken by the machine learning model to load into the function and perform the inference of the image. In our case the image is classified based on the Image-Net data labels. In other words, it is the speed of the model inside a lambda function. It is important to study the execution time of the model for comparison and analysis of models. The execution time is calculated in a similar fashion of the lambda function response time. The average value is calculated based on all the executions for a single lambda function. The equation for the calculation is given below:

$$\frac{\sum_1^n E}{n} \tag{2}$$

where E = Execution Time
and n = Total Number of Requests

- **Lambda Function Cost**: In this experiment we are using the AWS lambda functions for analysing the behaviour of ML models and ML frameworks. Calculating the cost for using the serverless architecture is one of the most important metric. Cost calculation is very important for 2 reasons, the first is that machine learning models and libraries use a certain amount of storage. The libraries imported in the code need to be packaged in the docker container.Coming to the second reason, machine learning models that take large time for execution are expensive. Every cloud service provider calculates the cost of utilisation in a different manner. Since we are using AWS lambda functions let us analyse the 2 main factors that play a role in the cost calculation.
  1) Resource Consumption: : The billing of an AWS lambda function is based on the number of executions and resource consumption. The calculation of the amount of resource The calculation of observed resource consumption is done by multiplying average memory size by the time taken to execute the function. The unit for memory size is gigabytes and it is rounded up to the nearest 128 megabytes up to the maximum size of 1,536 megabytes. Furthermore, the unit for execution time is milliseconds (ms) which is rounded up to the nearest 1ms. The cost of the resource consumption also varies according to the region where the function is deployed. We have

	Tensorflow			Pytorch			Caffe		
	R	I	M	R	I	M	R	I	M
Cost	0.065	0.053	0.016	0.029	0.017	0.056	0	0.006	0.028

TABLE I
AWS LAMBDA COST CALCULATIONS

deployed the AWS lambda functions in the EU- South-1 (Milan region) as it is the nearest location for us. The observed resource consumption is at the end multiplied by the current price for resource consumption, which is $0.0000195172 per GB-s as of now. The total cost for an AWS lambda function is calculated for a monthly basis by adding up both the resource consumption cost and execution cost. The mathematical formula for calculating the cost for the azure functions is shown in eq 3,4 and 5. The cost for each of the ML models used along with the respective ML framework is shown in Table I.

$$RCCost = MS * Exec * Price(GB - s) \quad (3)$$

$$ExecCost = Exec * Price(Million) \quad (4)$$

$$AWSLambdaFunctionCost = RCCost + ExecCost \quad (5)$$

Where, *RC = Resource Consumption*,
MS = Memory Size,
Exec = No. of Executions
and *AWS Lambda Function Cost*

Our study monitors the performance metrics discussed above. These metrics are analysed and monitored throughout the simulation. In the next section we will discuss the results of the experiments and the analysis of the performance measure values.

D. Results

This section illustrates the results obtained by deploying and executing the various machine learning models on the serverless platform. We will be discuss a number of research questions that will evaluate the performance of the simulation. The answers will be assisted by the graphs and tables for easier interpretation. Taking into account the simulation has been performed with the 1998 FIFA World Cup logs, we will discuss the follwing research queries:

- **RQ1. What is the cost of execution of each framework?** The cost of execution for each framework is calculated by using the equations 3, 4 and 5 which describes the exact methodology used to calculate the cost as described in the AWS documentation. The cost of deploying the services is as shown in figure 4.
  In the cost graph, the cost of various models deployed using the AWS lambda functions is depicted. Cost in dollars is displayed on the y-axis whereas the x-axis

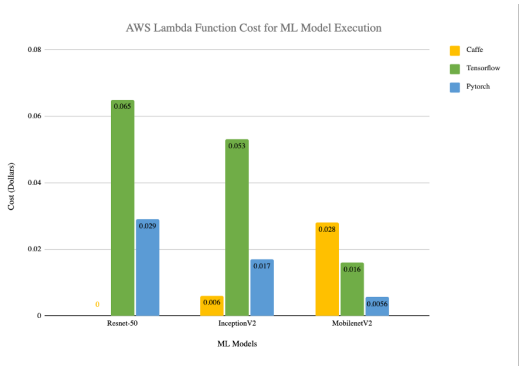

Fig. 5. AWS Lambda Function Cost

consists of the ML models. Different colors are used to denote the frameworks. The individual cost values are shown in Table I. The table describes the cost incurred for deployment of each model. Only the initials of the model is displayed in the table. As it can be seen from the table, for resnet and imagenet models, tensorflow framework is the most expensive framework to deploy. Pytorch framework is the like the median in terms of cost whereas caffe framework is the least expensive. For mobilenet model, the tensorflow framework is the least expensive whereas the pytorch framework is the most expensive.

- **RQ2. What is the response time of the function while using each of the frameworks** The response time of an AWS lambda function is calculated by taking the average of the response time of the function divided by the number of executions of that function.

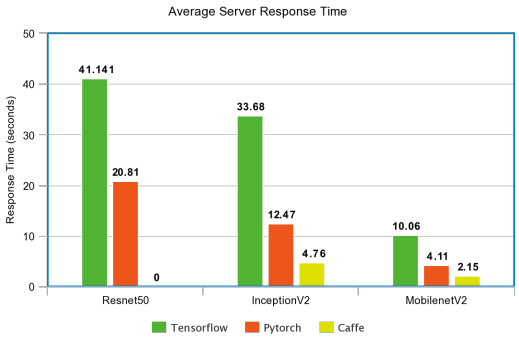

Fig. 6. AWS Lambda Response Time Graph

In figure 6, the average server response time is displayed along the y-axis and the machine learning models along

Response Time	Tensorflow			Pytorch			Caffe		
	R	I	M	R	I	M	R	I	M
	41.141	33.68	10.06	20.81	12.47	4.11	0	4.76	2.15

TABLE II
AWS LAMBDA RESPONSE TIME

Model Execution Time	Tensorflow			Pytorch			Caffe		
	R	I	M	R	I	M	R	I	M
	20.33	16.53	7.93	15.25	7.42	3.71	0	2.91	1.42

TABLE III
ML MODEL EXECUTION TIME

the x-axis. The various frameworks are highlighted by the different colors in the graph. The individual response times are displayed on the bar of each ML model.
Columns in table II depict the response time of each of the model. The initials of the models are considered in the table. From the data, the tensorflow models consume the greatest amount of time to respond whereas the caffe models take the least time to respond. The pytorch models are in the middle.

- **RQ3. What is the average time taken to serve the model?** The averagr time taken by the machine learning model to load in the AWS lambda function is known as the time taken to serve the model. Equation 2 depicts the method of calculation of this parameter.
Table III depicts the average model execution time. Tensorflow models are quite heavier and take time to load whereas the pytorch models are little lighter. Caffe models are the lightest and take the least time to load.Only the initials of the models have been used in the table

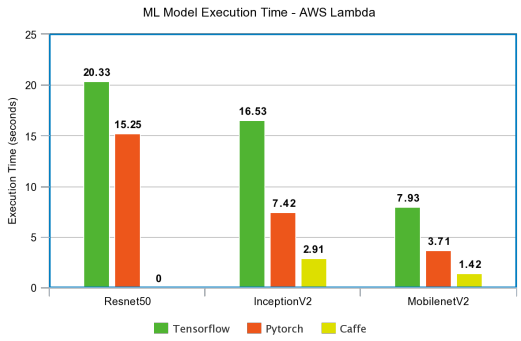

Fig. 7. ML Model Execution Time

As depicted in the above graph, the various vertical bars depict the model execution time. The x-axis depicts the grouping of the models and the frameworks whereas the y-axis depicts the execution time. In terms of loading the model into the AWS lambda function, the mobilenet models are the fastest to load whereas the resnet models being heavier are slower.

- **RQ3. RQ4. Which framework provide the best support for serverless deployment?** Data obtained from our simulation indicates the homogeneity across the performance measures of the evaluation candidates. For example, among all the among all the performance measures, tensorflow seems to shown the highest values for cost, response time and execution time and caffe seems to show the lowest values while pytorch lying somewhere in the middle. The homogeneity of data proves the uniqueness of the results generated through the simulation. This is due to the fact that if the execution time of a model is quite high, then for sure the response time of that specific model will be quite high. However, the cost also depends on many factors like dockerisation, model size and several other factors. We have considered the major factors into consideration in our experiment.
Moreover, the model Resnet-50 and MobileNet occupy huge memory and are quite expensive in terms of expenses and execution time. Therefore the cost is high for that specific model. **Overall, caffe framework is the best according the data obatined from the simulation taking all the factors into consideration**. The main factor playing with the executions and the cost is the dockerisation of models and libraries associated with AWS. It has a huge impact in terms of cost. Overall smaller models are least expensive to deploy in AWS lambda.

III. DISCUSSION

In this section, the discussion revolves around the results obtained from the experiment as described in section II-D. We will also illustrate the factors influencing the deployment of the ML models and the ML frameworks.

In the table 4 the factors affecting the deployment of the ML frameworks in AWS lambda are listed. Similarly in table 5, the factors affecting the machine learning models are illustrated. Let us now focus a bit on the results of the experiment.

Firstly, let us discuss the best framework for implementation in AWS lambda functions. Tensoflow, being a complex framework was taking the highest time for execution and model loading in our experiment. Pytorch, being the lighter framework than tensorflow was performing better according to the graphs and tables obatined in the simulation. Caffe,

Factors influencing serverless deployment (AWS lambda)	Tensorflow	Pytorch	Caffe
Complexity	Complex	Less Complex	Least Complex
Implementation	C++	Python	C++
Additional libraries required for deployment in AWS Lambda	Yes (Json, pillow, PIL)	Yes (Json, pillow, PIL)	Yes (Json, cv2)
Cost of execution	Highest (except in case of MobileNet)	Medium	Low (except in case of MobileNet)
Response Time (In AWS lambda simulation)	Highest	Medium	Low

TABLE IV
FACTORS AFFECTING MACHINE LEARNING FRAMEWORKS

Factors influencing serverless deployment (AWS Lambda)	ResNet-50	InceptionV2	MobileNetV2
Complexity	Very complex	Less complex	Least complex
Model Size	Huge	Huge	Small
Type of Architecture	Pyramidal cell architecture	Convolution layer	Residual Structure
Response Time (In AWS lambda simulation)	Highest	Medium	Low
Cost of execution In AWS lambda simulation)	Highest	Medium	Low
Type of neural network	Convolution Neural Network	Convolution Neural Network	Convolution Neural Network
Major usage	Image Classification	Feature Extraction	Computer Vision

TABLE V
FACTORS AFFECTING MACHINE LEARNING MODELS

on the other hand being the least complex framework, was the best in terms of execution speed and the response time. **Coming to the cost factors, there were some exception in same of caffe and tensorflow as models are dockerised for deployment in AWS Lambda**. This was a crucial factor in the cost determination.

Secondly, let us dicuss the best machine learning model to be deployed in the simulation. ResNet-50 being the most complex model was quite expensive and took a large duration of time for inference. Hence the cost of the deployment was expensive. Coming to InceptionV2; it was also a complex model but took lesser time for execution and loading. It was comparatively better than ResNet50. MobileNetV2 being the least complex model was the best in term of cost, execution time and loading time.

Overall, the results obtained were homogeneous and the study was successful in establishing the factors influencing deployment of machine learning models using various frameworks on AWS lambda.

IV. Conclusion

In this study, we performed an in depth comparison of running various machine learning models in various machine learning frameworks on AWS lambda environment. The study is whether the deployment of machine learning models is suitable on serverless frameworks or not.

Firstly, the study follows an experimental setup which consists of a serverless architecture and deployment of 8 lambda functions. We have used the 3 frameworks and the 3 machine learning models as listed in the abpve sections. The images are uploaded to an S3 bucket to trigger the lambda function for inference. The image upload follows the 1998 FIFA world cup logs. The results are then stored in cloudwatch and then pushed to a S3 bucket. The analysis of the results is done using CSV files downloaded from S3 using Microsoft Excel. The graphs were generated using a python code. The machine learning frameworks that we have used in our study are Tensorflow, Pytorch and Caffe. On the other hand, the ML models are Resnet-50, InceptionV2 and MobilenetV2.There are three different performance measures that we have employed in our study, i.e. Function Response Time, Model Execution Time and Function Cost.

Secondly, the performance measures are calculated using the equations that are documented in the AWS documentation. They are listed above as equation 1,2,3,4 and 5. At the end, we have listed the factors that had influence on the deployment and execution. We have also discussed the overall performance and felt that models which are lighter are the best to deploy in serverless environments.

Overall, MobileNetV2 was the best machine learning model deployed and Caffe was the best framework in the simulation. AWS lambda has several disadvantages with these kinds of deployments. The role of IAM, docker and other limitations of lambda function in terms of layers are overhead for easier deployment.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[3] Andrew Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 04 2017.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding, 2014.

[5] T. Jin M. Arlitt. 1998 world cup web site access logs, August 1998.

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[7] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge, 2015.

[8] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[10] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.