Serverless Platform Performance Analysis

Shivesh Ganju
New York University
sg6148@nyu.edu

Ravi Shankar
New York University
rs6980@nyu.edu

***Abstract* - Many enterprises and software solutions nowadays incorporate deep learning models and use them for inferencing so as to provide the customers the best experience possible. Since modern day software is now moving to microservice architecture, large deep learning models need to be suitable for serverless platform computing. In our experiments, we analyse the performance of 3 popular deep learning models namely Resnet50, InceptionV3 and Mobilenet and evaluate their inference latency on AWS Lambda using Tensorflow. Our experiments show that cold starts cause a significant impact in API and might be harmful in applications which are sensitive to time. Moreover our experiments show how a developer has to make a tradeoff between cost and performance in the form of choosing the right set of configuration parameters and the right deep learning model**

1. Introduction

As the computer science field is ever evolving, in the coming decade serverless cloud computing will be a dominating force and has already been adopted by industry as well as academia experts [1]. A compelling reason to try out serverless computing is the on-demand scaling feature which can automatically scale depending on the incoming traffic while another key aspect is the cost analysis where we are only charged for resources used and there are no provision costs.

Recently there has been a huge surge in leveraging cloud services for AI tasks by companies. In our work we try to realise the effect of Function as a Service (Serverless Computing using AWS Lambda) for running deep learning inference tasks. In particular we also see the effect of cold starts on skewing the latency distribution and thus not being able to meet the stringent requirements set by SLAs.

2. Related Works

Most of the work in cloud computing till now have been done using Virtual Machines, however use of serverless computing for inference tasks have been proposed of late. [2]

An attempt was made to study the latencies by Mallaya et al [4] but in their proposed solution appropriate latencies for fetching models from S3 buckets weren't factored into the overall latency visualization. Other aspects such as cold starts, effect of memory models on inference times weren't considered as well.

Ishakian et al. [3] experimented using deep learning models like Squeezenet, ResNet-18 and ResNeXt-50 and deployed them on AWS Lambda. The authors used Amazon's MXNet deep learning framework for their study. The authors also conducted load testing using Java's Jmeter framework.

3. Approach

In the following section we discuss how the different experiments were conducted for Resnet50, InceptionV3 and MobileNet.

AWS Lambda also provides the users an option to configure the memory of the Lambda function. In order to see the impact of the memory on the application performance, Memory was increased exponentially from 512MB to 3000MB for all the experiments.

A. Inference

The main application deployed on the Lambda function is responsible for making an inference about a random image chosen from the ImageNet dataset. The deep learning models chosen in our experiments were ResNet50, InceptionV3 and MobileNet. The development

environment chosen was Tensorflow since it is one of the most widely used frameworks with a large support forum. The above models were pretrained on the ImageNet dataset. Table 1 shows the comparison of the models on factors such as size and their performance.

Model	Size (MB)	Top 1 Accuracy ImageNet	Top 5 Accuracy ImageNet	FLOPS
ResNet50	98	0.749	0.921	4 GFLOPS
MobileNet	16	0.704	0.895	579 MFLOP
Inception V3	92	0.779	0.937	6 GFLOPS

Table 1: Model Comparison on Size and Accuracy

From the table we can see that ResNet50 is the largest model whereas MobileNet is the smallest model in terms of size. However we also see that since MobileNet has a smaller size and relatively smaller number of parameters to train, it also has lower performance than ResNet.

In the application, model weights were separately uploaded as part of the functional dependencies of the Lambda function. This was done to avoid the additional overhead of downloading the weights which would not give an accurate picture of the latency. The inference latency was calculated as the time taken to make a forward pass through the image.

B. Cold start experiments

Cold start is described as the phenomenon when the Lambda function has to initialize a container and be in a running state. This happens whenever a Lambda function gets triggered after a long time and happens because the Lambda function gives up its resources after being idle for a long duration of time.

In order to perform a cold start experiment AWS Cloudwatch was used to generate periodical events so as to trigger the Lambda function. Lambda is an event based function which basically means that it gets triggered and starts executing the code. Cloudwatch was used to generate events after every 10 minutes for a period of 2 hours in order to collect the latency metrics which were published on the CloudWatch Dashboard. Latency has been described as the time taken by the Lambda function to finish execution.

C. Warm Start Experiments

After a cold start phenomenon occurs with the Lambda function, the function retains its resources if it gets triggered in short intervals of time. As a result, whenever a Lambda function is called shortly after a cold start, it does not have to initiate a new container and reuses the container which was allocated in the previous cold call. This is known as a warm start and it does not cause an additional overhead of container bootstrapping.

Warm start experiments were performed using REST APIs. AWS APIGateway was used to generate an endpoint which could be hit using REST. Whenever a request was made to this endpoint the Lambda function was triggered and started executing. API Latency was calculated as the time taken to receive a response from the application after sending a GET request to the endpoint.

D. Load testing Experiments

In order to test the scalability of the application and see if Lambda is the correct environment for serving a high request rate application for any deep learning application, load testing experiments were performed.

Locust was used for performing the load testing of the application. Locust is a python's framework which swarms the user's application with millions of simultaneous users. Figure 1 shows how we increase the request rate per second for our application. We increase our

request rate linearly from 0 to 100 requests/second.

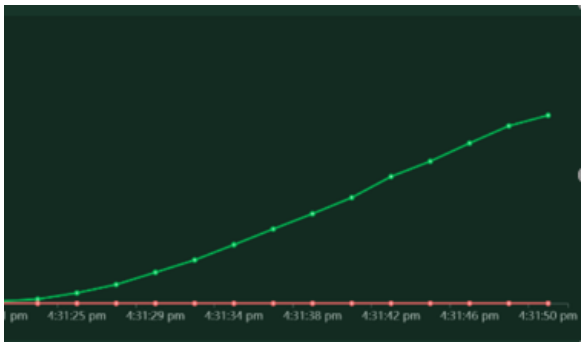

Figure 1 : Request rate for load testing experiments

A single request can be defined as a GET request similar to that which was described in section 3.C. The latency metrics as well as the inference time of the application was collected from the locust dashboard as well as the API responses which returned the inference times as well.

4. Implementation Details

A. AWS Lambda function

AWS Lambda is a serverless compute platform provided by Amazon. The maximum deployment package size of any Lambda function is 50MB. Tensorflow is not supported by Lambda in its python runtime. Therefore additional Tensorflow libraries had to be added in the functional dependencies of the Lambda package. Moreover in order to get the accurate API Latency, model weights had to be included locally along with the image so as to avoid the time it takes to get the image from S3. All the libraries and files caused the size of the project to be beyond 1GB due to which AWS Console could not be used for creating the Lambda function.

The alternate approach was to create a docker image of the project which had the development environment built through the Dockerfile and upload it to AWS ECR which is Amazon's Docker container registry. Lambda provides an option to be created through a docker image which is stored in ECR.

Therefore in order to build our project, we uploaded the image to ECR using AWS CLI. After that a Lambda function was created for every model using the docker images stored in ECR.

B. Warm Start Experiments

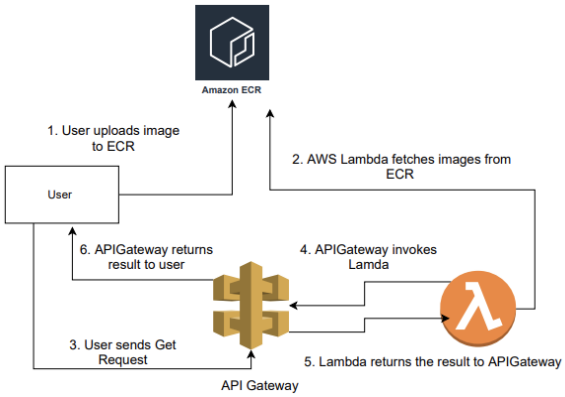

Figure 2 : Warm start experimental setup

Figure 2 describes the architecture adopted to conduct the warm start experiments. First step is for the user to create a docker image of the project and upload it to ECR as mentioned in section 4A. After that the user creates the Lambda function using the AWS Console using the ECR image which was created in the previous step.

AWS APIGateway was used to create the REST endpoint for the user and to trigger the Lambda function. The GET request was made using the requests library provided by python. The workflow for warm start experiments have been discussed in section 3C.

Load testing architecture also adopted the same experimental setup by integrating Locust with REST endpoints.

C. Cold Start Experiments

Figure 3 shows the experimental setup for performing cold start experiments. Cold start experiments could not use the same architecture that was used for the warm start experiments.

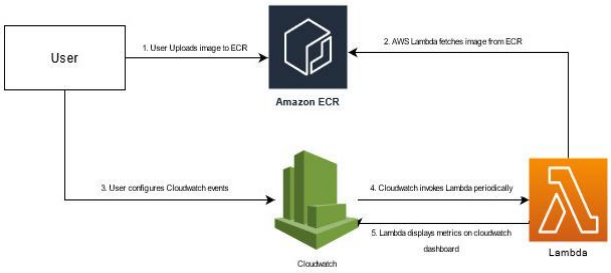

Figure 3 : Cold start Experimental Setup

This was because APIGateways provide API endpoints which get timed out after 29 seconds. Since cold start experiments involve a significant overhead of bootstrapping a container and in our case an additional overhead of fetching the image from ECR and deploying it on the container, this process took more than 29 seconds (Exact figures can be seen in the experimental results section).

In order to avoid this bottleneck, AWSCloudwatch was used to trigger the Lambda. AWSCloudWatch provides a mechanism to generate periodical events through the Events functionality which can be connected to Lambda. AWSCloudWatch was then configured to generate events after every 10 minutes for performing the cold start experiments. The results were also published on Cloudwatch and could be seen on the CloudWatch dashboard. The approach has already been discussed in section 3B.

5. Experimental Results

A. Warm Start Experiment Results

The following section describes the results obtained from the experiments performed in the above sections. The results have been divided into 4 sections, Warm start results, Cold start results and Load testing results and Summary where we compare all the models together along with some cost analysis.

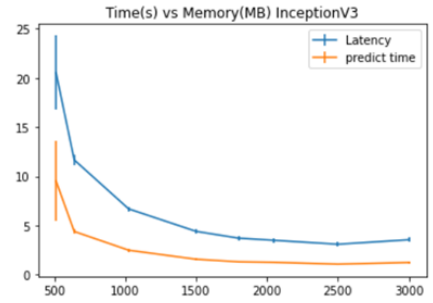

Figure 4: Warm Start Experiments : Time vs Memory for InceptionV3

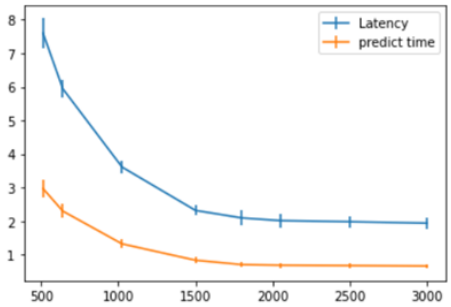

Figure 5: Warm Start Experiments : Time vs Memory for MobileNet

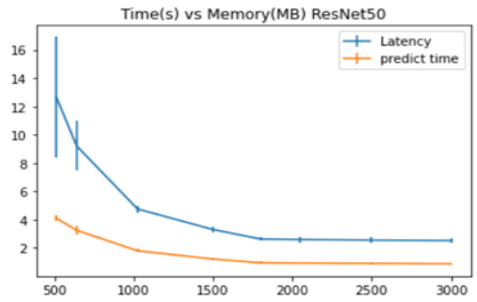

Figure 6: Warm Start Experiments : Time vs Memory for Resnet50

From figures 4,5 and 6 we can see that as we increase the memory, the latency as well as the inference time decreases for all the models. One possible reason is that since the size of the tensorflow model and the model weights exceed 500 MBs, the entire program instructions cant be loaded into the memory. As a result many memory swap ins and swap outs have to be performed by the OS with the hard disk which causes a significant performance impact. As we

increase the lambda's memory, more in-memory processing could be done which thus decreases the number of swaps. As a result the time decreased.

After a certain point increasing the memory does not cause a bigger decrease in latency since the entire model and the libraries could have been loaded into the memory and the only additional resources assigned would have been I/O resources which does not cause a significant impact in latency We also see as the size of the model decreases from Resnet to MobileNet, the latency decreases. This comparison has been done separately with explanations for the same.

B. Cold Start Experiment Results

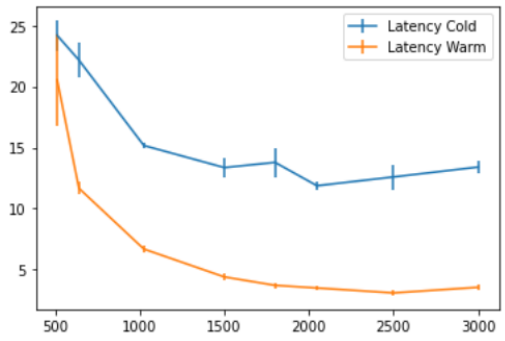

Figure 7: Cold Start Experiments : Time vs Memory for InceptionV3

Here we visualise the effect of cold start experiments. Figures 7,8 and 9 show a comparison of Cold start and Warm start experiments. From the graphs it is clear that cold starts have a significant impact on API latency.

The reason is because during a cold start the Lambda first initiates a container and then initialises it. After that it also has to deploy the docker image on the container. After all this is done, the code executes. This additional overhead of bootstrapping the container and deploying an image on it is significant as can be seen from the graphs.

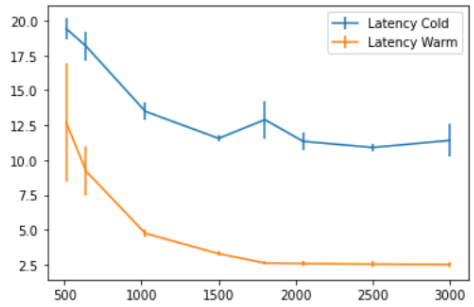

Figure 8: Cold Start Experiments : Time vs Memory for ResNet50

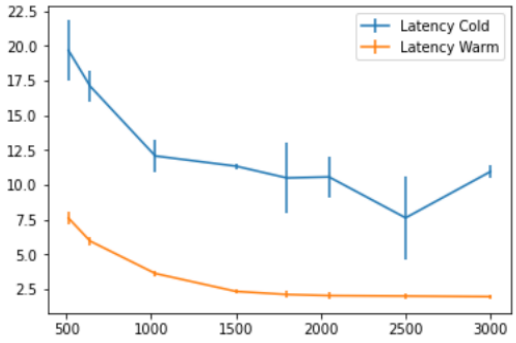

Figure 9: Cold Start Experiments : Time vs Memory for MobileNet

For warm start, the container that had been previously initialised in the cold start is reused so there is no additional overhead of bootstrapping and deploying and hence the latency is low. The trend on increasing the memory is not as evident as it was in the warm start experiments since the cold start overhead has no relation to the memory of the lambda function. The memory of the lambda function is important only for executing the code.

C. Load Testing Experiments

From figures 10,11 and 12 We see that the load testing results show similar results as the warm start experiments with almost similar trends for inference time as well as memory.

We also see that Lambda is a highly concurrent process and shows promising scalability results.

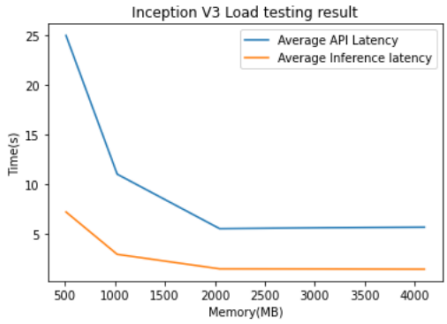

Figure 10: Load Testing Experiments : Time vs Memory for InceptionV3

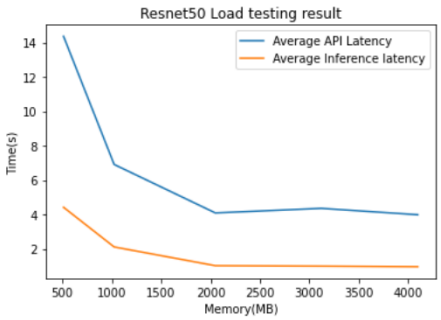

Figure 11: Load Testing Experiments : Time vs Memory for ResNet50

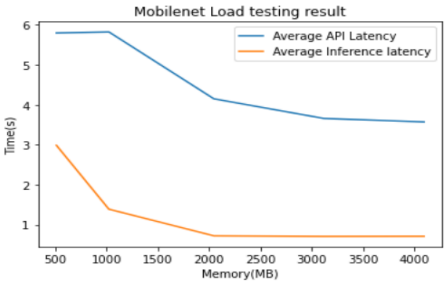

Figure 12: Load Testing Experiments MobileNet Time vs Memory

D. Compare and contrast models

In this section we contrast and compare cold start, warm start and inference time on Lambda.

Fig 14 shows the API Latency in Warm starts in all the models. The table 1 (Page 1) compares the models on the basis of their size, accuracy and FLOPS For 14,15,16 we see that mobilenet has the least latency followed by Resnet and inception. This is applicable for both inference times and API latencies. This correlates to the comparison on the Flops shows in table 1 (Page 1) where we see MobileNet having the least amount of Flops followed by Resnet and then inception.

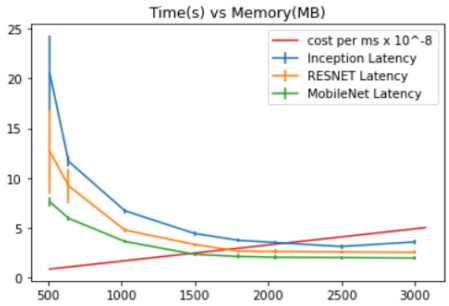

Figure 14: API Latencies in Warm Start

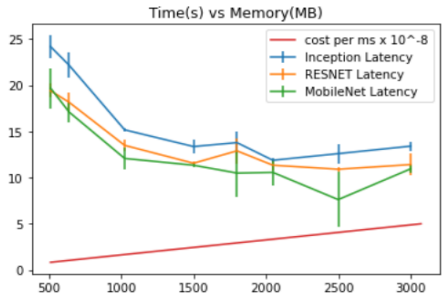

Figure 15: API Latencies in Cold Start

We also see that in low memory configuration, mobilenet performs the best since it has the smallest size as compared to all the models. We also see all the models showing a decrease in latency as the memory increases.

Another point worth noting is how the cost increases as we increase the memory. However it is evident from figures 14,15,16 that after increasing the memory beyond 1500 Mbs for MobileNet and 1750 MB for Resnet and Inception, there is not a significant decrease in latency but there is a significant increase in price as we move from 1500 to 2000 MB. At the first sight cost for lambdas might not seem significant, however considering real world applications where millions of requests are landed per second (considering each request is being handled by an individual lambda) on a real world application costs come out to be a substantial number.

Lastly we see that MobileNet would be the cheapest model since it takes the minimum time followed by Resnet and Inception for any memory configuration although it gives lower accuracies than them.

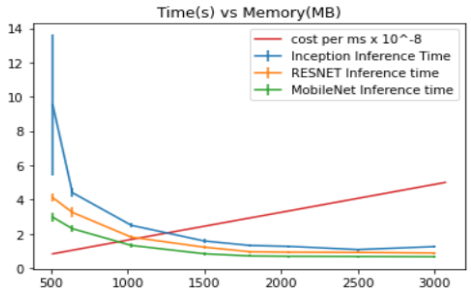

Figure 16: Inference time for all models

6. Conclusions

Increasing memory of the Lambda functions decreases the latency.

From the experiments we see that by increasing the memory of the lambda function, the inference time and latency improves. This is because with more RAM, more instructions per second can be executed in the memory and there is a lesser need for memory swapping which decreases the latency.

Delays occurred as a result of cold starts can skew the latency distribution and hence risk violating more stringent SLAs.

We see that the bootstrapping of a container in Cold Start causes a significant overhead as compared to warm starts. Therefore before starting any Lambda application, a dummy call should be made for allocating a container as for a real world application serving results from cold start could be a bad user experience.

Lambda is highly scalable and can serve highly concurrent applications.

We also see that Lambda function is quite suitable for serving deep learning models since under a huge load, there is not a significant impact to the latency, thus demonstrating the concurrency power of Lambda functions.

High latency sensitive applications can choose lower accurate models. High accuracy models have to bear more latency.

While selecting a particular deep learning model, we see that MobileNet has the least latency among all the above models. One major factor is due to lesser number of parameters to train as well as a small model size which causes the forward pass time to be less. However, the user has to choose between API Latency time vs Accuracy of the model while developing the application as we see MobileNet has the least accuracy in the 3 models. If our application wants higher accuracy we'll have to bear the latency and vice versa.

While developing any application, a correct set of memory configuration has to be kept in mind while keeping the project under a budget. Increased cost is not always correlated with better performance.

For choosing how much resources to allocate to the Lambda function, we see that after increasing the memory beyond a particular threshold, there is not much significant improvement in the latency but there is a significant rise in cost. For developing any solution, a decision must be made between the cost and the latency as per the business requirements. If the application requires very low latency, high memory configuration can be chosen increasing the price of the project. Hence we see that Increased cost is not always correlated with better performance.

7. References

[1] Jonas *et al* Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]

[2] S. T. predictions with Python and A. Lambda,"https://medium.com/tooso/serving-tensorflow-predictions-with-pythonand-aws-lambda-facb4ab87ddd," 2017.

[3]Ishakian *et al.*, "Serving deep learning models in a serverless platform," arXiv preprint arXiv:1710.08460v2, 2018

[4] S. Mallya, "Seamlessly scale predictions with aws lambda and mxnet," 2017, https://aws.amazon.com/blogs/compute/seamlessly-scalepredictions-with-aws-lambda-and-mxnet/