Containerized or Serverless Learning: Outcomes

Diya Ramasamy, Iris Ma, Krithika Balasubramanyam, Raj Shreyas Penukonda, Yuyu Lai
{dlr6, jma277, kba111, rsp8, yuyul}@sfu.ca
GitHub: https://github.com/irisjiayuema/containerized-vs-serverless

Project Title

We compared the latency of deploying machine learning (ML) workloads on containerized and serverless environments. All code and additional results are on GitHub.

System Design

Models

We deployed three ML models:

- Classical Machine Learning
- Natural Language Processing (NLP)
- Computer Vision (CV)

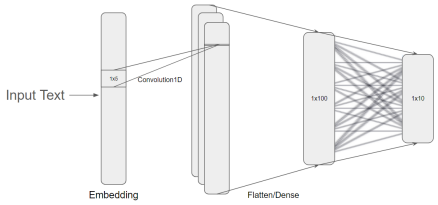
We chose these three types because each model takes different input types and are all popular in modern ML.

Classical Machine Learning: Logistic Regression on the Iris Dataset We fit a Logistic Regression on the tabular iris dataset. We retrieved the Logistic Regression model and iris dataset from Python's scikit-learn library; the data consists of three species of irises (Setosa, Versicolour, and Virginica) and their numerical characteristics. The model predicts the species of iris from these numerical features. After training, the model is saved as a pickle file.

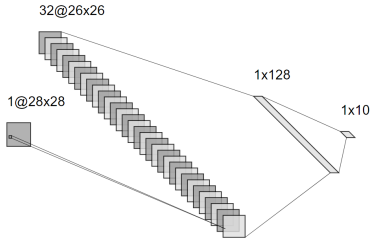
sepal length	sepal width	petal length	petal width	species
5.1	3.5	1.4	0.2	0
4.9	3.0	1.4	0.2	0
⋮	⋮	⋮	⋮	⋮

Table 1: Iris dataset overview.

Natural Language Processing: Predicting Wine by Reviews We trained a text classification system containing an Embedder and a 3-layer Convolutional Neural Network (CNN). This model is designed to identify wine types based on their reviews. We trained on the Wine Review dataset using Keras and TensorFlow. For model deployment, we stored the model architecture in a .pb file, while its weights and hyperparameters were automatically saved in separate files. Here is the architecture:



Computer Vision: Image Classification on Fashion MNIST Data We opted for a 3-layer CNN model to classify grayscale images into 10 distinct fashion categories. We trained on the Fashion MNIST dataset using Keras with the Tensorflow 2.13 framework. We saved the resulting .pb file with the associated weights stored in separate files. Below is the architecture:



Testing

We called the endpoints of each model from Postman with batch sizes of 1, 2, 5, 10, 50, 100. Each of these test inputs is an instance of the data being predicted. For example, to test batch sizes of 1 and 2 of the Iris dataset, we would submit:

Batch Size 1:

```
{
  "instances": [
    [6.7, 3.1, 4.7, 1.5]
  ]
}
```

Batch Size 2:

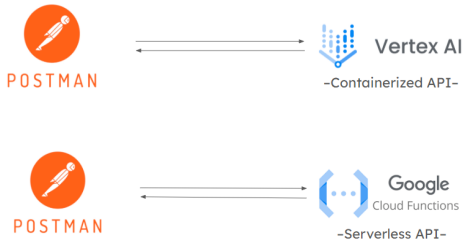
```
{
  "instances": [
    [6.7, 3.1, 4.7, 1.5],
    [5.1, 3.2, 2.1, 1.0]
  ]
}
```

This allowed us to capture valuable insights on how Vertex AI and Cloud Functions compare on scalability and latency.

Implementation

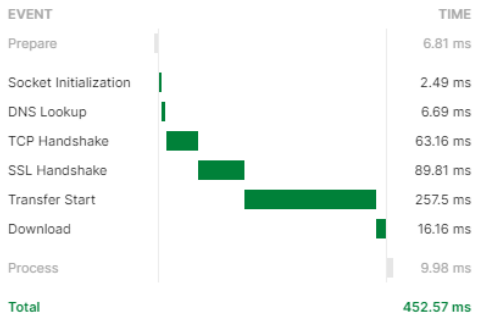
We used Google Cloud Platform for both deployment environments to further reduce other factors that could affect the comparison such as different companies operating from different data centers. All models are loaded to Google Cloud Bucket, where they are accessed by both deployments.

We chose Google's Vertex AI for the containerized environment and Google Cloud Functions for the serverless deployment. We made all calls to both APIs from Postman.

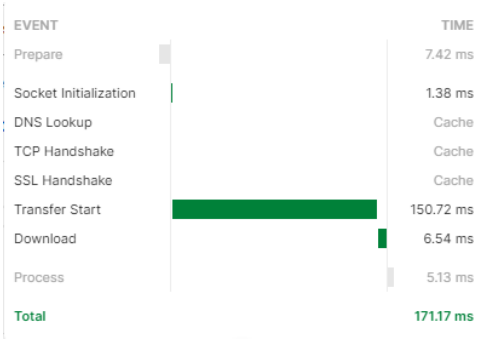


Metrics

Postman reports several different values as shown in the following screenshot:



Upon experimentation with Postman we found that DNS Lookup, TCP Handshake and SSL Handshake are cached after the initial run:



Caching appears to have a significant effect on the total time. So, to avoid recording the overhead numbers, **we focus on the Transfer Start as our primary latency metric**. Additionally, as latency appears to vary between runs, we performed 3 trials, some with and others without cache.

Results

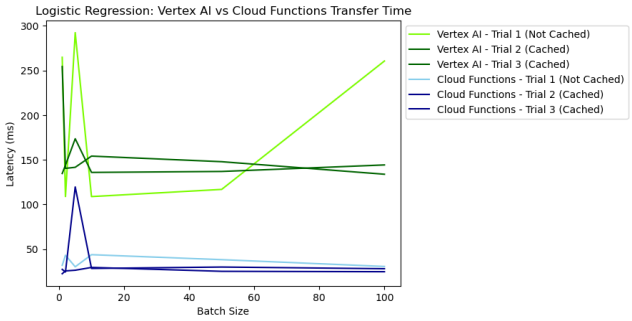
Following the logic above, we plot the Transfer times. We have recorded additional values from Postman, however it was clear that Transfer time was most reflective of the latency we experienced.

Note that while we do not expect cache to affect the transfer times, we did label whether or not a trial was cached as it impacted the total time (on GitHub).

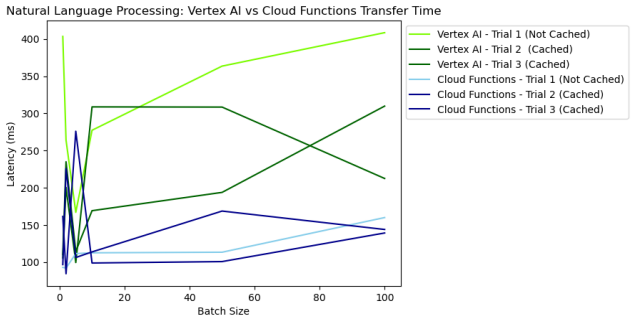
Primary Metric: Transfer Time

All plots in this section were generated from Postman's Transfer Start times. Plot code and tabular data is on GitHub.

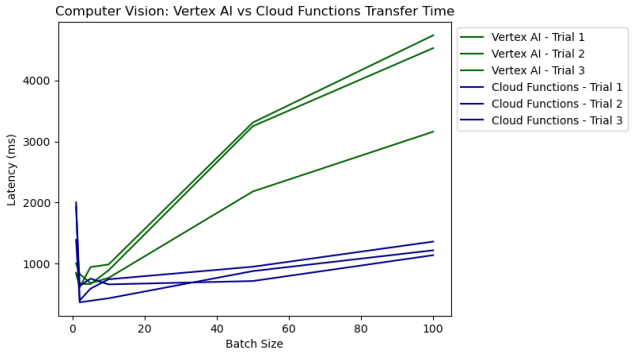
Logistic Regression



Natural Language Processing



Computer Vision

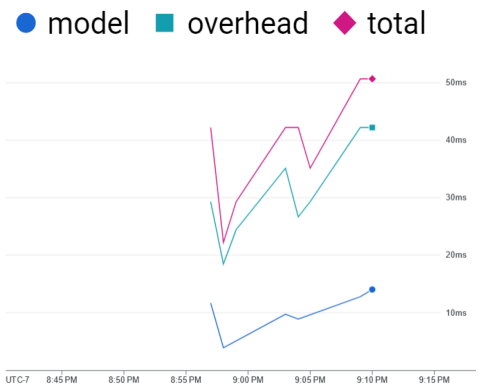


Note that during this final experiment, caching occurred at random and was out of our control. Almost all measurements are taken when there was no cache.

Additional Metrics

To implement the feedback for our progress report, we also looked for the latency measures provided by Google Cloud Platform's logs. However, while Google Cloud Functions provides an "execution time", Vertex AI provides a specific "prediction latency" that is broken down into model and overhead, which sum to a total. Due to these differences, we did not use these values as a basis for comparison, but they reflect a possible finding. We will show just the NLP model's plots here.

Vertex AI



Cloud Functions

Analysis

Primary Metric: Transfer Time

From the plots reported in the primary metric section of the results, Cloud Functions, the serverless implementation, has lower latency across all models. For the simplest logistic regression model, this is always true. For lower batch sizes for NLP and CV, it was less obvious that one environment was faster than the other. However, as the batch size increases, Cloud Functions always have lower latency.

We believe that the serverless deployment is more successful in our results due to its scalability and dynamic resource allocation. The flexibility to assign resources as needed allows the system to perform consistently, regardless of workload.

Additional Metrics

Though we were unable to directly compare these plots, they show an interesting result.

As time went on, we submitted bigger batch sizes. For the containerized implementation, it appears that the batch size caused a slight increase in latency for both the model and overhead. By contrast, in the serverless implementation, there was no explicit pattern during the calls made (the model was deployed at 9:20 PM and all calls were made before 9:35 PM). This suggests that batch size may affect the model's latency in containerized implementations but the latency of the serverless deployment does not have a direct relationship with the batch size. A similar scenario was observed for the CV model which had larger inputs, while no pattern was observed for the logistic regression which had the smallest inputs.

This result also aligns with our previous conclusion that the serverless implementation benefits from dynamic resource allocation. It appears the containerized deployment is affected by batch sizes as its resources are fixed. Alternatively, by increasing resources for higher workloads, the serverless deployment can consistently respond with low latency.

Cost Though unrelated to latency, we would like to make a final note of the costs of deploying our models. Overall, Vertex AI charged $35.48 while Cloud Functions charged $2.02 for our testing. We believe that this large disparity may also be due to the containerized deployment reserving additional unused resources while the serverless deployment is billed on demand. On latency, scalability and cost, serverless is preferable.

Image Sources

- Postman Logo
- Vertex AI Logo
- Google Cloud Functions Logo