Deployment and Optimization of Machine Learning Models in a Serverless Architecture

By Ngai Lai Yeung and Zhang Zi Di

1. Introduction

This project focuses on deploying a CNN-based image classification model using OpenFaaS.[1] OpenFaaS is an open-source framework that allows developers to easily deploy and manage serverless functions in a variety of environments.

It provides a powerful and flexible environment for building serverless applications, enabling developers to easily create and manage functions while focusing on their core business logic. Its open-source nature and active community contribute to its growing popularity in the serverless ecosystem.

2. System Design and Implementation

To compare the performance of serverless and traditional server architectures, we implemented two similar systems that run the same CNN-based image classification model.

2.1 The Machine Learning Model:

We deployed a Convolutional Neural Network (CNN) model for image classification using the CIFAR-10 dataset.[2] The model architecture is specifically designed for this task and includes the following layers:

Convolutional layer with 32 filters → ReLU activation → Dropout layer → Convolutional layer with 32 filters → MaxPooling layer (2x2) → Convolutional layer with 64 filters → ReLU activation → Dropout layer → Convolutional layer with 64 filters → MaxPooling layer (2x2) → Fully connected layer.

To mitigate overfitting, dropout and 2x2 max pooling layers are incorporated between each convolutional layer. The optimizer used is Nesterov Momentum with a learning rate of 0.01.

2.2 Serverless Architecture Design

The serverless architecture utilizes OpenFaaS, a platform that facilitates the deployment of event-driven functions and microservices. OpenFaaS is deployed on a local Kubernetes cluster simulated via KinD (Kubernetes-in-Docker).[3]

The OpenFaaS function is designed to respond with the model's accuracy after deployment. The function is packaged using the `python-http-debian` template, which provides a Python runtime environment for HTTP-based functions. PyTorch[4] is used as the deep learning framework. Conda was used within the Docker container to address conflicts caused by PyTorch installation with `pip`.

2.3 Traditional Server Architecture Design

To provide a comparison with the serverless architecture, the CNN model was also deployed using a traditional server-based approach. A Flask[5] application was developed to expose the model functionality via a REST API. The application is also designed to respond with the model's accuracy after deployment. The Flask application is hosted on an AWS EC2 `t3.medium` instance to simulate a traditional server environment.

3. Experiments and Results

To evaluate the performance of both serverless and traditional server-based architectures, several experiments were conducted. The focus was on key metrics that provide insights into the efficiency and resource utilization of both approaches.

3.1 Metrics:

The experiments were designed to measure the following key performance indicators:

Startup Time: Time taken to start the service from a cold state. This involves training the machine learning model from scratch and takes a significant amount of time.
Response Time: Average time to process a single request.
Resource Utilization: CPU and memory usage during idle and busy states.

3.2 Test Environment:

To ensure a fair comparison, both architectures were tested under controlled environments with similar resource configurations:

- **Serverless (OpenFaaS)**:
  - Tested locally on Windows Docker Desktop with Kubernetes-in-Docker (KinD).
  - The hardware environment was **32 vCPUs** and **64 GM RAM**, which is significantly greater than the AWS EC2 instance described below.
  - To establish a meaningful comparison, we apply a scaling factor of 16 to normalize the benchmark results, as our local environment has exactly 16 times the computing resources (32 vCPUs vs 2 vCPUs) and memory capacity (64 GB vs 4 GB) compared to the AWS instance. While this linear scaling approach has its limitations, it provides a reasonable basis for approximate performance comparison.
- **Traditional Server (Flask)**:
  - Deployed on an AWS EC2 `t3.medium` instance running Flask as the WSGI server.
  - The instance was configured with **2 vCPUs** and **4 GB RAM**.
- **Benchmark Tools**:
  - Python's `time` module was used for measuring startup time.
  - `ab` (Apache Benchmark)[6] generated concurrent HTTP requests to evaluate response times.
    The command we used is
    ```
    ab -n 10 -c 2 -s 300 <URL of the REST API>
    ```
    which specifies a total of 10 requests, with 2 requests sent concurrently in each batch, and a timeout set to 300 seconds for each request.
  - Resource usage was monitored using `docker stats` for OpenFaaS and `htop` for Flask.

3.3 Results:

Metric	Serverless (OpenFaaS)	Traditional Server (Flask)
Startup Time	Actual: 78.30s Normalized: 1252.8s	844.46s
Avg. Response	Actual: 4.6s	28.80s

	Normalized: 73.60s	
Resource Usage (Idle)	CPU (actual): 0.16% CPU (normalized): 2.56% Mem (actual): 0.34% Mem (normalized): 5.44%	CPU 4.9% Mem 26.6%
Resource Usage (Busy)	CPU (actual): 15.41% CPU (normalized): 246.56% Mem (actual): 8.06% Mem (normalized): 128.96%	CPU 99.9% Mem 26.49%

3.4 Analysis:
The experiments reveal distinct trade-offs between serverless and traditional server-based architectures, highlighting their strengths and weaknesses in different scenarios:

Startup Time: Serverless architectures, such as OpenFaaS, exhibit longer startup times. This is probably due to the **cold start issue**,[7] where functions need to be initialized on demand. In contrast, traditional servers like Flask are always running, providing faster and more predictable startup times.

Average Response Time: Serverless showed a slower average response time compared to Flask. This is likely due to the added overhead of managing serverless infrastructure, such as container orchestration and request routing.

Resource Usage During Idle Time: Serverless significantly outperforms traditional servers in idle resource efficiency. By dynamically scaling down to near-zero resource usage when no requests are being handled, OpenFaaS demonstrated only **2.56% CPU** and **5.44% memory** usage in idle states. In contrast, Flask maintained **4.9% CPU** and **26.6% GB memory**, reflecting the inefficiency of always-on traditional servers during periods of inactivity. If we consider the distortion of the linear normalization method, the idle resource usage for OpenFaaS should be even better.

4. Challenges and Solutions

4.1 Dependency Issues:

- **Challenge**: PyTorch installation caused conflicts when using `pip`.
- **Solution**: Dependencies were managed using Miniconda3, which resolved version conflicts.

4.2 Limited Backend Development Experience:

- **Challenge**: Lack of familiarity with HTTP request handling in Flask and OpenFaaS.
- **Solution**: Focused on implementing a basic GET method to achieve the core functionality that trains the model and reports its accuracy over the test dataset.

4.3 Hardware Resource Disparity:

- **Challenge**: Local testing environment (32 vCPUs, 64GB RAM) significantly exceeded the production server specifications (2 vCPUs, 4GB RAM), making direct performance comparison impractical.
- **Solution 1**: Applied a scaling factor of 16 to normalize benchmark results, based on the exact ratio of computing resources between environments. While this approach enables approximate comparison, we acknowledge its limitations as performance scaling may not be strictly linear.
- **Solution 2:** Attempted to restrict the local hardware resources through Docker Desktop's WSL 2 configuration (`.wslconfig`) to match the production environment (2 vCPUs, 4GB RAM). However, this approach led to unexpected timeout issues during model training, suggesting that the resource constraints might be too aggressive for the current implementation.

5. Conclusion

In conclusion, serverless architectures excel in scalability and idle resource efficiency, making them well-suited for dynamic, event-driven workloads. Traditional servers, however, remain advantageous for predictable workloads, offering faster startup times and consistent performance. The choice between the two depends on the specific application requirements and traffic patterns.

6. References

[1] O. Ltd, “Home,” *OpenFaaS - Serverless Functions Made Simple*. https://www.openfaas.com/

[2] A. Krizhevsky and G. Hinton, “*Learning multiple layers of features from tiny images,*” Master's thesis, Dept. of Computer Science, Univ. of Toronto, 2009. https://www.cs.toronto.edu/~kriz/cifar.html

[3] “kind,” *kind.sigs.k8s.io*. https://kind.sigs.k8s.io/

[4] PyTorch, “PyTorch,” *Pytorch.org*, 2023. https://pytorch.org/

[5] Flask, “Welcome to Flask — Flask Documentation (3.0.x),” *Palletsprojects.com*, 2024. https://flask.palletsprojects.com/en/stable/

[6] “ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4,” *Apache.org*, 2019. https://httpd.apache.org/docs/2.4/programs/ab.html

[7] E. Jonas et al., ‘Cloud programming simplified: A berkeley view on serverless computing’, arXiv preprint arXiv:1902. 03383, 2019.

[8] “Locust - A modern load testing framework,” *locust.io*. https://locust.io/