

Deploying Deep Learning Models Using Serverless Computing for Diabetic Retinopathy Detection

Matheus W. Camargo(✉), Cristiano Alex Künas, and Philippe O. A. Navaux

Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
{mwcamargo,cakunas,navaux}@inf.ufrgs.br

Abstract. The incidence of diabetes is increasing at an alarming rate across the world. As a result, cases of diabetic retinopathy (DR) are on the rise, a complication of diabetes that in its most severe form can lead to blindness. The lack of specialized labor for diagnosis, essential for the successful treatment of the disease, brings the need to study alternatives for diagnosis via computational means. Recent research on the use of Deep Learning for the detection of DR proves to be an important alternative to improve the use of specialized labor, by prioritizing the most serious cases. From this context, the work objective is to evaluate the performance and financial cost of alternatives based on serverless computing for the deployment of Deep Learning models for DR classification. Using the Amazon Sagemaker serverless inference service, optimizations and different configuration alternatives were considered, obtaining up to 9.4% of financial cost reduction and up to 2.35× performance boost. Finally, concepts such as containerization and infrastructure as code were used during the solution implementation, to allow the reproduction of deployment and experiments performed.

Keywords: Serverless computing · Cloud computing · Deep Learning · Diabetic Retinopathy · Deployment

1 Introduction

Diabetes has been growing globally at alarming rates. In 2021, it was estimated that around 537 million people suffered from this disease worldwide (Fig. 1). It is expected that 1/3 of these people will develop diabetic retinopathy (DR). DR is a complication of diabetes in which a high level of blood sugar causes damage to the retina, impairing vision and even leading to blindness at its most severe level.

Within this context, diagnosing the disease in its early stages is still the best form of prevention since, after diagnosis, it is possible to prevent DR stage progression by controlling blood sugar levels and blood pressure. An optical coherence tomography makes the diagnosis (OCT) examination, which obtains images of the patient's retina, which a trained ophthalmologist subsequently evaluates.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023 Workshops, LNCS 14109, pp. 274–289, 2023.
https://doi.org/10.1007/978-3-031-37120-2_18

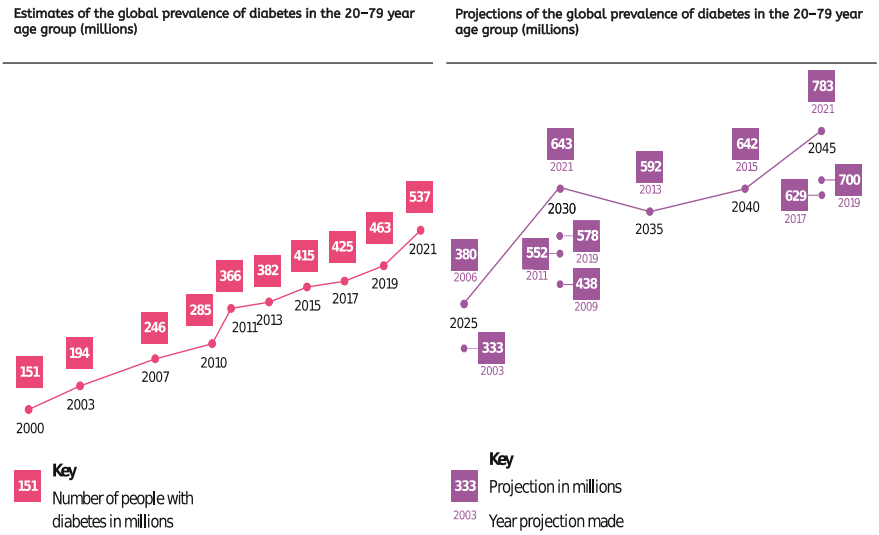

Fig. 1. Estimation and projection of the prevalence of diabetes globally. The left Figure shows the evolution of diabetes prevalence, which was most recently estimated at 537 million people worldwide. The right Figure shows a projection of diabetes until 2045, in which approximately 783 million people will suffer from diabetes [5].

An example of the effectiveness of public policies for diagnosing the disease is the reduction in the number of cases of blindness caused by DR in the United Kingdom. Following the introduction of a comprehensive policy aimed at promoting DR testing on a large scale, the disease lost its position as the leading cause of blindness in adults [13].

Although preventive exams are recognized worldwide as the primary way to combat the disease, this practice is still not widely disseminated due to the shortage of specialized professionals capable of evaluating eye fundus images. In China, the ratio between patients with diabetes and ophthalmologists reaches 1:3000 [4], making applying policies to perform tests at the national level impracticable.

Therefore, it is necessary to apply computational methods capable of evaluating eye fundus images and detecting the presence of diabetic retinopathy in its different degrees of complexity. A very promising approach is the use of *Deep Learning*, in which neural networks are trained in a supervised way to receive eye fundus images and classify the image, interpreting it according to different stages of DR. Such technology, if made available to health professionals, could be used to prioritize the most severe cases, thus providing better use of the limited workforce capable of confirming diagnoses.

Given the current context, this study assesses an alternative approach to deploying two neural network models trained to detect Diabetic Retinopathy on a serverless platform. To achieve this, the Amazon Sagemaker cloud computing

service will be utilized, with a focus on evaluating both the financial cost and performance of the proposed solution.

The rest of this paper is organized as follows. Section 2 covers existing work on deep learning deployment on serverless platforms and deep learning for DR detection. In Sect. 3, we describe implementation details, outline the hardware and software setup used, and the experiments' methodology. The results of optimizations' performance evaluations and resources for research reproduction are presented in Sect. 4. Finally, Sect. 5 concludes the paper and outlines future work.

2 Related Work

In this Section, works related to the *deployment* of Deep Learning models are discussed, a fundamental step for making the algorithms available to the end user. As presented in Sect. 2.1, serverless computing presents itself as a compelling alternative due to reduced financial cost and ease of scaling. Finally, in Sect. 2.2, we present works that focus on using Deep Learning techniques to detect diabetic retinopathy. As shown, such techniques have proven to be a promising alternative to the shortage of specialized labor capable of diagnosing DR.

2.1 Serverless Computing for Deep Learning Models Deployment

The usage and popularity of the serverless paradigm are emerging rapidly. This paradigm presents compelling advantages, as pointed out in [1]. For instance, it can significantly reduce financial costs due to the ability to scale compute resources down to zero. Another critical advantage the paradigm offers is the ease of configuration and deployment, resulting in the abstraction of operations and infrastructure configurations.

However, this paradigm also presents significant drawbacks and open problems. For example, resource limitations can exist, such as constraints on maximum memory size and CPU usage. Another critical problem is the cold start latency, a consequence of scaling compute resources down to zero and can significantly hinder application latency.

The use of serverless computing for machine learning and deep learning poses challenges due to these applications' high computational and memory demands, as highlighted in [18]. Despite that, numerous studies have been conducted to overcome these challenges and enable deploying deep learning models in serverless platforms, enabling deploying such models without configuring infrastructure.

In this context, [7] addressed using serverless computing to deploy deep learning models. The authors used Amazon's serverless cloud computing service, AWS Lambda. Although Lambda functions simplify the process of managing the infrastructure on a large scale and even reduces operating costs, their use penalizes the application's performance in the form of cold-starts and warm-starts, as the authors showed.

Furthermore, as shown by the authors of [7], the latency for both cold-start and warm-start cases tends to decrease as the memory increases. However, this decrease is limited, as there are cases where increasing the available memory raises the cost per activation without significant performance gains. Additionally, the size of the model used directly impacts both latencies, making it important to consider models of different sizes when evaluating optimizations and different configurations. These findings inspired our own selection of models to assess how various optimizations impacted the application's overall performance.

Similarly, in [19], the serverless computing paradigm was explored to deploy neural networks for NLP (Natural Language Processing). The models were deployed in a pay-per-request model, with zero ongoing cost for maintaining the service active. Interestingly, as stated by the authors, the system was scalable without any extra engineering effort, restating one advantage of the serverless paradigm.

2.2 Deep Learning for DR Grading

Machine learning has gained prominence in recent decades, showing significant advancements in diverse fields, such as object recognition or disease diagnosis. In [4], the authors presented the DeepDR system, which, based on a residual neural network, showed that it is possible to detect the presence of different degrees of diabetes with an area under the receiver's operational characteristic curve from 0.916 to 0.970 for each degree of DR.

Other neural network architectures can also be used for DR detection. The work of [14] is based on the Inception-V3 architecture. A significant conclusion of the work concerns the impact of image resolution on model training. After an increase in image resolution, it was possible to increase the accuracy of detections by 6%.

Interestingly, even smaller models can successfully detect diabetic retinopathy. In [16], the MobileNet architecture was trained using the APTOS dataset, obtaining a model with accuracy and precision of 95%. As mentioned in the 3.1 section, to analyze alternatives for the deployment of models considering different sizes of neural network architecture, one of the architectures used was MobileNet, due to its reduced size.

The reduced size of the MobileNet architecture has been exploited for DR detection even in mobile applications. In the [3] work, the MobileNet architecture was used so that the models' inference could be performed on cell phones, allowing a complementary diagnosis to be carried out for users who do not have a stable internet connection, such as users who reside in the rural area.

3 Methodology

This section covers the development and implementation details of the solution for the *deployment* of DR classification models using cloud computing. Moreover,

we will also address the methodology followed in carrying out the experiments and evaluating the implemented optimizations.

The 3.1 section addresses the development and implementation of the solution, such as the model training method and dataset used, the cloud serverless service used and required interfaces, data extraction, and analysis, as well as the process, followed to deploy endpoint inferences. Finally, the 3.2 section describes the methodology used to carry out the experiments.

3.1 Solution Development

First, we trained and selected models for the detection of DR. For this purpose, a notebook from the Kaggle platform was used, where 27 pre-trained neural network architectures were trained. The dataset used for training is derived from the APTOS 2019 competition dataset, available at [10]. The class distribution of the derived dataset is identical to the competition's original dataset, as shown in Fig. 2.

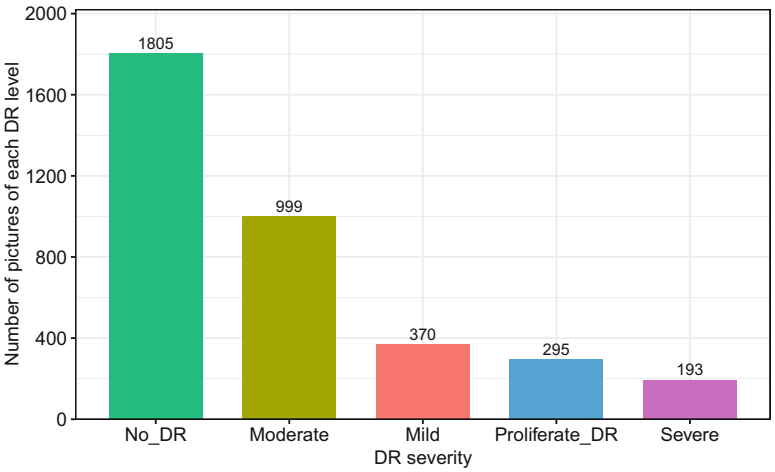

Fig. 2. Distribution of dataset classes used for model training.

All images were resized to 224×224 resolution for different pre-trained deep-learning models. In addition, a Gaussian filter was applied to the images to reduce noise. As shown in Fig. 3, the samples are labeled according to 5 degrees of DR: no DR, mild, moderate, severe, and proliferative.

As shown in [7], model size has a significant impact on both warm-up and cold-start latency. Therefore, to verify the behavior of models of different sizes against different optimizations, the VGG19 and MobileNet models were chosen. The relationship between models, artifact size on disk, and the number of parameters is presented in the Table 1.

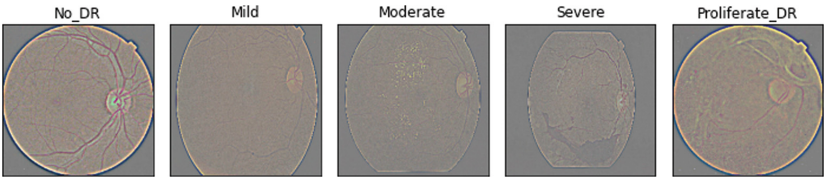

Fig. 3. Images sample from the dataset used for model training. All images are labeled according to 5 degrees of DR: no DR, mild, moderate, severe, and proliferative

After training the models, they were saved in the SavedModel [8] format. So that the models could then be used in the Sagemaker [12] serverless inference service, the model was compacted following the tar.gz format and uploaded to an S3 bucket.

Having the model artifacts hosted in an S3 bucket, it was then necessary to develop a code that loads the model and predicts the classification of diabetic retinopathy from an image. However, given the time taken to load the models from the file system, a *Singleton* class was defined for loading the model and making inference predictions so that the model was kept in memory and reused between different inference requests.

Table 1. Relation between model, number of parameters, and artifact size in the file system. We can notice that the VGG19 is about six times bigger than the MobileNet model.

Model	Number of parameters	Artifact size (SavedModel)
MobileNet	3.377.221	15,7 MB
VGG19	20.107.205	77,7 MB

To satisfy the container interface expected by the Sagemaker serverless inference service, it was necessary to implement an HTTP API using the FastAPI [11] library. Finally, the container's entry point was defined in a file called `serve`, which aims to start sub-processes for the NGINX and Gunicorn services. The NGINX [15] service acts as a proxy to the HTTP API workers, redirecting API calls to Gunicorn [17]. Gunicorn is a pre-fork model Web Server Gateway Interface [6] server that can run and manage multiple workers of an HTTP server.

After starting the processes that will serve the Sagemaker inference requests, the `serve` process continues to wait until the completion of its child processes, also having the responsibility of killing the NGINX processes and the Gunicorn process if it receives a SIGTERM signal indicating that the container will be destroyed. A diagram showing sub-process creation during container initialization is shown in Fig. 4.

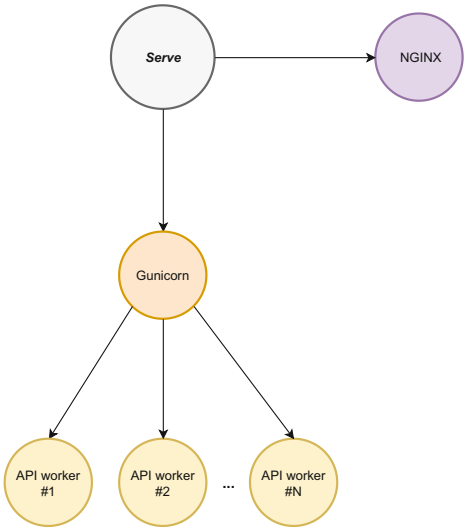

Fig. 4. Diagram of sub-process creation during container initialization. The `serve` process is the main process responsible for creating the NGINX sub-process and Gunicorn sub-process. Then, the Gunicorn sub-process creates N workers, depending on the number of vCPUs of the instance used to run the containers.

3.2 Experiments Methodology

Given that the Sagemaker serverless inference service scales the compute resources down to zero when the application is idle, there are two different latency cases: warm-up and cold-start. The former happens when the compute resources are already initialized, not increasing the application's latency. The latter occurs after a period of application idleness, usually after 10 to 15 min, when the compute resources have been destroyed and need to be re-initialized to serve new requests.

Based on these two types of latency, we then defined two experiments scenarios, each one dedicated to measuring a specific case of latency:

- **Warm-up scenario:** First, we make a warm-up request to initialize the container, and the other requests do not suffer from *cold-start* latency. Then, 25 requests are made in series, with intervals of 5 s between each request.
- **Cold-start scenario:** 10 sequential requests are made, with intervals of 15 min between each request, so that the container is destroyed between executions, resulting in cold-start latencies that we want to measure.

For both scenarios, two metrics related to the application execution time were extracted: total latency and prediction time. Total latency is the time taken between sending the request and the server's response, encompassing both the time for sending the image, model prediction, and server response. The prediction time measures only the time taken to perform the inference of an image.

Using both scenarios, two optimizations were evaluated: docker image optimization and endpoint memory size optimization. The optimizations are further elaborated upon in the subsequent subsections.

Docker Image Size Optimization. To verify the impact of docker image size on the cold-start latency of the application, two optimizations were applied. The first optimization applied was refactoring the docker file to use the concept of multi-stage build [2]. Next, dependencies were reduced, eliminating non-essential dependencies for the application's operation and optimizing the size of existing dependencies, such as replacing the tensorflow-mkl dependency with the Tensorflow dependency. The optimizations and their impact on the size of the resulting Docker image are shown in Table 2.

Table 2. Table of different optimizations applied to the base image and the respective disk image size (without being compressed) and compressed image size (size occupied by the image in image registries, like ECR).

Optimization	Compressed size	Disk size
None	1798.27 MB	4.97 GB
Multi-stage build	923.99 MB	3.32 GB
Dependencies reduction	517.23 MB	2.06 GB

Since the application's container is already running, its initialization time does not influence the application's latency, since the container is only destroyed after some idle time. For this reason, only the cold-start scenario was considered when evaluating the impact of Docker image size optimizations on latency. A total of six experiments were run by combining each optimization and model.

Endpoint Memory Configuration. To verify the impact of the endpoint memory size configuration on the prediction time and the application's total latency, the following memory values were used: `1024MB`, `2048MB` and `3072MB`.

Increasing the memory configuration of a lambda function reflects both the cold-start and the warm-up case, as shown in [7]. For this reason, both the cold-start and warm-up scenarios were evaluated to verify if the same behavior was maintained. On total, twelve experiments were run by combining three memory configurations, two models and two scenarios (cold-start and warm-up).

4 Results

This Section presents the results obtained using the serverless inference service based on the experiments mentioned in Sect. 3.2. We analyzed two optimizations' performance and financial cost: Docker image size and endpoints memory configuration.

The first optimization applied while using the serverless inference service was the Docker image size optimization. Reducing the image size was essential for testing other optimizations and versions of algorithms used in this work since a smaller image leads to lower upload latency for image records and shorter build time. Interestingly, the Docker image optimization also reduced the cold-start latency for both analyzed models, as detailed in the Sect. 4.1.

Optimizing the endpoints' memory size brought significant gains in performance and cost for the execution of the analyzed models, as will be detailed in the Sect. 4.2.

Finally, in the Sect. 4.4 we provide supplementary material for research reproduction.

4.1 Docker Image Optimization

As mentioned in Sect. 3.2, the cold-start latency does not contribute to increasing the prediction time of the models. Therefore, the only fundamental metric to be analyzed is the total latency time of a request.

Analyzing Fig. 5, it can be seen that optimizing the Docker image size increased performance for both models. Comparing the total latency time of the largest Docker image (no optimization - 1798.27 MB) with the smallest Docker image (optimized dependencies - 517.23 MB), there was a speed-up of 1.2864× for the MobileNet model and of 1.3272× for the VGG19 model.

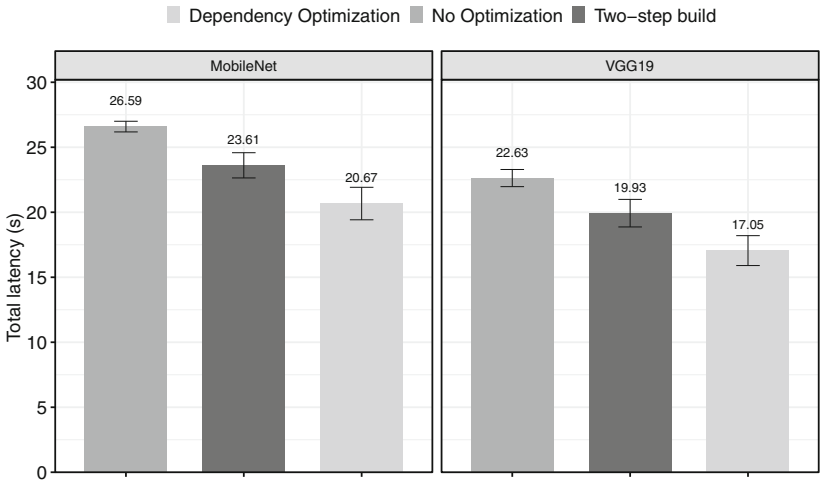

Fig. 5. Total latency time for the MobileNet and VGG19 models for different Docker images. The total latency time is reduced as optimizations are applied to the docker image.

Not limited to the issue of `cold-start` latency time, optimizing the size of the Docker image is essential for reducing the time to deploy, build, and reduce the cost of storing Docker image registries as the ECR service used in this work.

4.2 Endpoint Memory Size

As mentioned in [7], as the memory size configuration of the endpoints increases, there is a decrease in both the total latency and the prediction time of the models.

Next, we analyze the impact of increasing memory for both the `warm-up` and `cold-start` scenarios, described in Sect. 3.2.

Warm-Up Scenario. Analyzing Fig. 6, we noticed a significant reduction in the prediction time and total latency for the VGG19 model. In particular, when comparing the 2048 MB memory configuration with the 1024 MB memory configuration, there was a speed-up of 2.74× analyzing the prediction time.

An exciting case is noticed when evaluating the performance and expense of the different memory configurations for the VGG19 model. As per Fig. 7, the lowest endpoint cost is achieved with 2 GB of memory, even though the cost per second of that endpoint is double the 1 GB endpoint configuration. As the 2 GB endpoint is 2.74× faster than the 1 GB endpoint, we arrive at a case where there is both cost reduction and performance improvement, as the execution cost is directly proportional to the execution time. Therefore, we conclude that an increase in memory configuration does not always imply an increase in the cost per endpoint inference.

By selecting the 2 GB memory configuration for the VGG19 model, we achieved a 9.4% cost reduction, combined with the performance increase mentioned previously.

Analyzing Fig. 8, we noticed a difference in behavior between the VGG19 and MobileNet models. Considering the prediction time of the MobileNet model, there was a considerable performance increase between the 1 GB and 2 GB endpoints, totaling a speed-up of 2.35×. However, when we look at the endpoint's total latency, the speed-up drops to 1.38×.

Since the MobileNet model's prediction time is short compared to the total latency, representing only 37% of the total latency, the performance gains from the memory increase are smaller than larger models such as the VGG19. Due to this fact, for the MobileNet model, every increase in memory meant an increase in cost, as shown in Fig. 7.

Cold-Start Scenario. Similar to the results of the `warm-up` scenario, there were also performance gains in total endpoint latency in the `cold-start` scenario as the endpoint memory used increased.

As shown in Fig. 9, for both the VGG19 and MobileNet models, there was a significant reduction in `cold-start` latency when comparing the 1 GB and 2 GB memory configurations. For the MobileNet model, there was a speed-up of

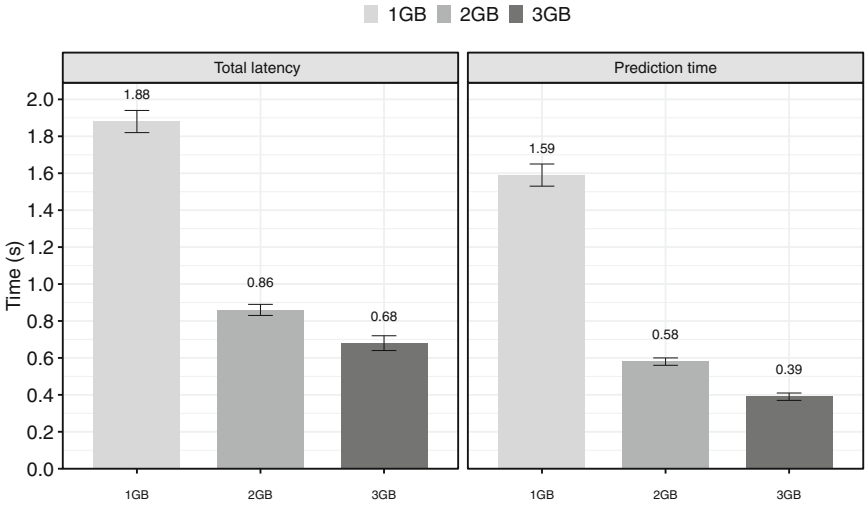

Fig. 6. Prediction time and total latency for the VGG19 model according to memory size. Note that as memory increases, time decreases.

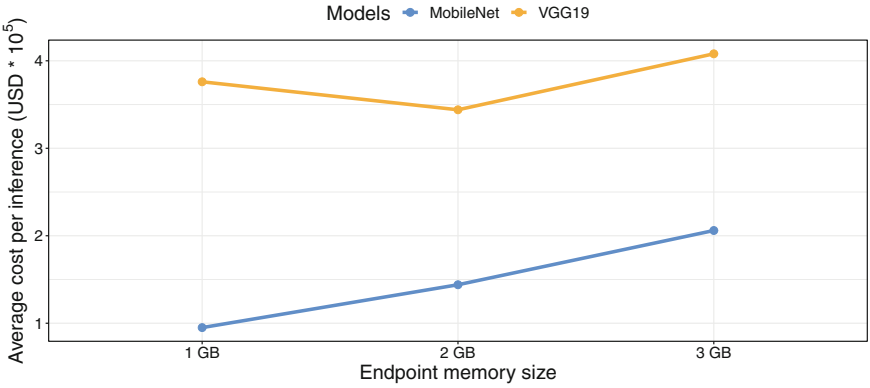

Fig. 7. Average cost per inference by endpoint memory size for MobileNet and VGG19 models.

1.65× between the 1 GB and 2 GB configurations, and for the VGG19 model, the speed-up was 1.54×.

Unlike the `warm-up` scenario, where the total latency reduction was more significant for the VGG19 model, both models showed similar performance gains in the `cold-start` scenario.

As previously mentioned, increasing memory does not translate into improved performance for models whose prediction time is a small fraction of the total latency time. However, the relationship between prediction time and total latency does not impact performance gains for the `cold-start` latency case.

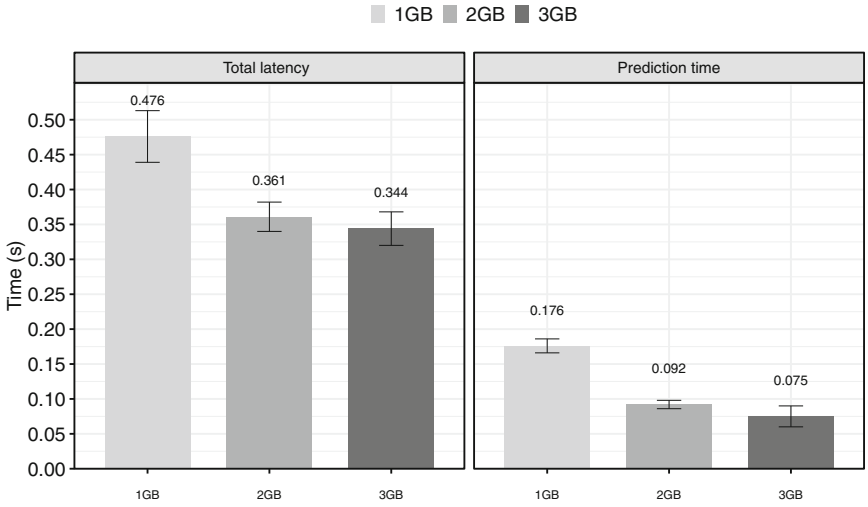

Fig. 8. Prediction time and total latency for the MobileNet model according to memory size. Note that as memory increases, time decreases.

To explain this phenomenon, we can analyze the *ModelSetupTime* metric, which refers to the time required to initialize the computational resources for the execution of the endpoints. According to Tables 3 and 4, we can see that as the memory grows, the *ModelSetupTime* decreases. Since *ModelSetupTime* is a critical additional component for the total latency of a request, this explains the reduction of `cold-start` latency for both models.

Table 3. Average ModelSetupTime of the VGG19 model for each used memory configuration. We can see that the time taken to initialize endpoint computational resources decreases as the memory size increases.

Endpoint memory size	*ModelSetupTime*
1 GB	10.7879 s
2 GB	6.8852 s
3 GB	6.5223 s

Interestingly, there was no considerable difference between the 2 GB and 3 GB memory configurations for both models, as seen in Fig. 9. This observation aligns with what was shown in [7], where the memory configuration of the lambda endpoints affected the `cold-start` latency. However, after a particular memory size, the increase in memory did not increase performance.

Table 4. Average ModelSetupTime of the MobileNet model for each used memory configuration. We can see that the time taken to initialize endpoint computational resources decreases as the memory size increases.

Endpoint memory size	*ModelSetupTime*
1 GB	10.6071 s
2 GB	5.6728 s
3 GB	5.6074 s

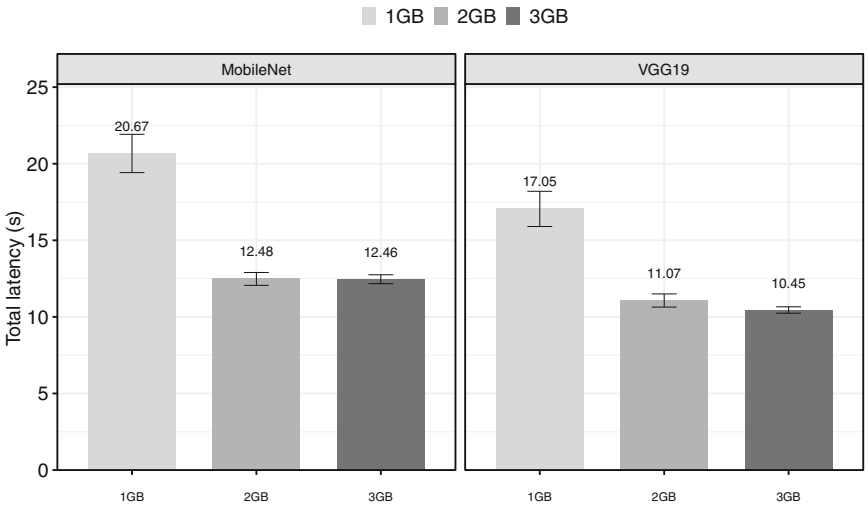

Fig. 9. Total `cold-start` latency for the MobileNet and VGG19 model in the serverless inference service for each used memory size.

4.3 Cold-Start and Warm-Up Latency Comparison

As previously mentioned, using serverless computing for deploying deep learning models presents higher latencies during the initialization of the computational resources dedicated to the execution of the inference, known as `cold-start` latency.

From the combination of Docker image size optimizations and endpoint memory configuration, it was possible to reduce the `cold-start` latency by about 2× for both models used, as shown in Table 5. There was a performance increase of 2.04× for the VGG19 model and 2.13× for the MobileNet model.

Even after applying the optimizations in combination, we noticed a big difference between the latency of the `warm-up` case and the `cold-start` cases' latency. Comparing the `warm-up` latency with the `cold-start` latency of endpoints with a 2 GB configuration, the cold-start latency is 57.42× higher for the MobileNet model and 19.82× higher for the VGG19 model.

Table 5. `Cold-start` latency by the model used before and after optimizations. Note that there was a performance increase of about 2×.

Model	Latency without optimization	Latency with optimization
VGG19	22.63 s	11.07 s
MobileNet	26.59 s	12.48 s

This large difference can be an impediment for solutions that have restrictions on performance. In these cases, there are alternatives to be explored to minimize the impact of cold-start latency, such as the use of requests to warm-up containers during periods of application use. More advanced techniques can be used, such as the use of time series forecasting to determine the periods of use of an application and the appropriate pre-allocation of resources, as studied in [9].

4.4 Resources for Research Reproduction

This work uses tools such as Docker and Terraform to improve research reproducibility. Nevertheless, the following resources are provided as supplementary material to the results:

- Public repository with source code on GitHub: The repository[1] contains code for provisioning infrastructure, building images, and also performing experiments and analyzing them. Additionally, all experiment results are present in the repository as CSV files.
- Public Docker image repository: All images used for the experiments are present in a public repository[2].
- Documentation: In the public GitHub repository, there is a file called `README.md`, containing instructions on how to use tools such as Docker and Terraform for generating images and provisioning the infrastructures used in this work.

5 Conclusion and Future Work

In this work, a solution for deploying deep learning models for detecting diabetic retinopathy in the cloud was implemented. For this, the Amazon Sagemaker serverless inference service was used.

Based on serverless computing, it was possible to deploy the models in a low-cost, on-demand solution without the need to provision and configure instances for the execution of the algorithms. As demonstrated in this work, this alternative suffers from high latencies in cases of cold-start, which can be up to 57× higher than the latency in the warm-up case.

[1] Link to GitHub: https://github.com/MatheusWoeffel/dr-detection-deploy.
[2] Link to ECR: https://gallery.ecr.aws/g1e7s9u4/dr-detection-deploy.

To mitigate this problem, it was shown that it is possible to reduce cold-start latency by reducing Docker image size and selecting the correct memory configuration, increasing performance up to 2.13×. Moreover, optimizing the memory configuration of the endpoints also brought performance gains for the normal latency case, improving performance by up to 2.35× and reducing the financial cost by 9.34%.

As demonstrated, the optimizations and alternatives explored behaved differently depending on the models' size. In this sense, a topic for future study is the use of neural network quantization techniques and the analysis of the consequences in terms of performance and financial cost of the solutions studied in this work, evaluated in the optimized models. Given the medical context of the application, it will also be necessary to carefully analyze the impact of quantization techniques on the accuracy and sensitivity of the models.

Another improvement topic is using different architectures to perform model inference. In addition to using CPUs, as researched in this work, it would be interesting to use instances with specific purpose processors destined to optimize the execution of inferences. For example, instances of the `inf1` type stand out, which use *systolic arrays* to optimize the computation of matrix multiplication, an important operation present in the execution of inference models.

Acknowledgment. This study was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, by Petrobras grant n.° 2020/00182-5, by CNPq/MCTI/FNDCT - Universal 18/2021 under grants 406182/2021-3, by CIARS RITEs/FAPERGS project and by CI-IA FAPESP-MCTIC-CGI-BR project.

References

1. Baldini, I., et al.: Serverless computing: current trends and open problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds.) Research Advances in Cloud Computing, pp. 1–20. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-5026-8_1
2. Bhat, S.: Understanding the dockerfile. In: Bhat, S. (ed.) Practical Docker with Python: Build, Release, and Distribute Your Python App with Docker, pp. 61–103. Springer, Cham (2022). https://doi.org/10.1007/978-1-4842-7815-4_4
3. Bidari, I., Chickerur, S., Kulkarni, A., Mahajan, A., Nikkam, A., Abhishek, T.: Deploying machine learning inference on diabetic retinopathy in binary and multi-class classification. In: 2021 International Conference on Industrial Electronics Research and Applications (ICIERA), pp. 1–6. IEEE (2021)
4. Dai, L., et al.: A deep learning system for detecting diabetic retinopathy across the disease spectrum. Nat. Commun. **12**(1), 1–11 (2021)
5. International Diabetes Federation: IDF Diabetes Atlas. International Diabetes Federation, Brussels, Belgium (2021)
6. Gardner, J.: The web server gateway interface (WSGI). In: The Definitive Guide to Pylons, pp. 369–388 (2009)
7. Ishakian, V., Muthusamy, V., Slominski, A.: Serving deep learning models in a serverless platform. In: 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 257–262. IEEE (2018)

8. Janardhanan, P.: Project repositories for machine learning with TensorFlow. Procedia Comput. Sci. **171**, 188–196 (2020)
9. Jegannathan, A.P., Saha, R., Addya, S.K.: A time series forecasting approach to minimize cold start time in cloud-serverless platform. In: 2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom), pp. 325–330. IEEE (2022)
10. Karthik, Maggie, Sohier Dane: Aptos 2019 blindness detection (2019). https://kaggle.com/competitions/aptos2019-blindness-detection
11. Lathkar, M.: Getting started with FastAPI. In: Lathkar, M. (ed.) High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python, pp. 29–64. Springer, Cham (2023). https://doi.org/10.1007/978-1-4842-9178-8_2
12. Liberty, E., et al.: Elastic machine learning algorithms in amazon SageMaker. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 731–737 (2020)
13. Liew, G., Michaelides, M., Bunce, C.: A comparison of the causes of blindness certifications in England and wales in working age adults (16–64 years), 1999–2000 with 2009–2010. BMJ Open **4**(2), e004015 (2014)
14. Moreira, F., Schaan, B., Schneiders, J., Reis, M., Serpa, M., Navaux, P.: Impacto da resolução na detecção de retinopatia diabética com uso de deep learning. In: Anais do XX Simpósio Brasileiro de Computação Aplicada à Saúde, Porto Alegre, RS, Brasil, pp. 494–499. SBC (2020). https://doi.org/10.5753/sbcas.2020.11546. https://sol.sbc.org.br/index.php/sbcas/article/view/11546
15. Nedelcu, C.: Nginx HTTP Server. Packt Publishing (2013)
16. Pavate, A., Mistry, J., Palve, R., Gami, N.: Diabetic retinopathy detection-MobileNet binary classifier. Acta Sci. Med. Sci. **4**(12), 86–91 (2020)
17. Pretty, D., Blackwell, B., et al.: H1DS: a new web-based data access system. Fusion Eng. Des. **89**(5), 731–735 (2014)
18. Shafiei, H., Khonsari, A., Mousavi, P.: Serverless computing: a survey of opportunities, challenges, and applications. ACM Comput. Surv. **54**(11s), 1–32 (2022)
19. Tu, Z., Li, M., Lin, J.: Pay-per-request deployment of neural network models using serverless architectures. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations, pp. 6–10 (2018)