Universitat Rovira i Virgili (URV) y Universitat Oberta de Catalunya (UOC)

Master in Computational and Mathematical Engineering

FINAL MASTER PROJECT

Area: Distributed Systems and Artificial Intelligence

Machine Learning on a Serverless Architecture

Author: Pablo Gimeno Sarroca

Tutor: Marc Sànchez Artigas

Tarragona, June 19, 2021

Dr. Marc Sànchez Artigas, certifies that the student Pablo Gimeno Sarroca has elaborated the work under his direction and he authorizes the presentation of this memory for its evaluation.

Director's signature:

Marc Sanchez Artigas - DNI 39902760Z (TCAT)

Digitally signed by Marc Sanchez Artigas - DNI 39902760Z (TCAT)
DN: c=ES, o=Universitat Rovira i Virgili, 2.5.4.97=VATES-Q9350003A, ou=Empleat públic de nivell mig, sn=Sanchez Artigas - DNI 39902760Z, givenName=Marc, serialNumber=IDCES-39902760Z, cn=Marc Sanchez Artigas - DNI 39902760Z (TCAT)
Date: 2021.06.19 18:11:25 +02'00'

Credits/Copyright

FINAL PROJECT SHEET

Title:	Machine Learning on a Serverless Architecture
Autor:	Pablo Gimeno Sarroca
Tutor:	Marc Sànchez Artigas
Date (mm/yyyy):	06/2021
Program:	Master in Computational and Mathematical Engineering
Area:	Distributed Systems and Artificial Intelligence
Language:	English
Key words	Serverless, Cloud, Machine learning

Abstract

The surge in cloud serverless technologies over the past few years has propitiated the appearance of new serverless-based use cases. Though it was originally intended to be used in internet of things (IoT) and web services, other general-purpose serverless use cases have been proposed, e.g. big data analytics. The goal of this FMP is to explore relatively recent serverless use case, machine learning (ML) training, and prove that, despite their limitations, serverless technologies can be a solid alternative to classic IaaS architectures for training machine learning models in the cloud. To do so, we provide various serverless, FaaS-based implementations of well-known ML algorithms, which we compare with other non-serverless implementations. The results of the comparison show that, with the proper optimisations, the proposed serverless implementation can be significantly faster than classic Python ML frameworks like Pytorch, proving that serverless technologies, and especially FaaS, are suitable for ML tasks such as training.

Keywords: Serverless, Machine learning, Cloud, FaaS

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few years, cloud computing has experienced an important surge in new technologies. From the very first cloud computing models, new cloud paradigms have been developed, some especially aimed at cutting costs and easing the deployment and scaling of applications. On this line we find **serverless computing**, which is a cloud computing model where the explicit management of resources such as servers or network is handled by the cloud provider. This model, as opposed to other classic cloud computing models, bases its billing on a *pay-as-you-go* model, where the user only pays for the resources they actually use. Hence, serverless provides flexibility, scalability and the optimisation of resource costs.

FaaS is a form of serverless runtime which allows running user-defined functions in small, independent stateless execution units. Some FaaS frameworks such as Lithops [12] allow running and managing the results of hundreds of parallel functions in the cloud. This high level of parallelism is an important aspect in machine learning (ML), and could be exploited to speed up the training of ML algorithms.

However, despite its benefits, serverless and FaaS present a series of **limitations** which need to be considered before building a serverless architecture for ML training. These limitations include limited resources, a lack of storage, unpredictable launch times and a lack of direct communication.

1.1 Aims of the work

The main goal of this FMP is to explore FaaS' properties and limitations to determine whether it is suitable for executing ML training tasks. In order to achieve this main goal, we investigate FaaS' properties and provide a serverless architecture designed for ML training that minimises FaaS' limitations and takes advantage of its strengths. Using this implementation, we provide a comparison with other state-of-the-arts ML and serverless big data processing frameworks to

prove the main point of this FMP.

To achieve the goals of the project. we divide the work in 3 main sections:

- **Background**. In the initial stage we provide the fundamentals of serverless computing, FaaS, ML algorithms, dataset preprocessing and related frameworks.
- **Implementation**. We provide a serverless implementation that takes into account FaaS' properties for ML. Additionally, we introduce other implementations to compare with the proposed serverless architecture.
- **Evaluation**. During this final stage we run a series of experiments to obtain performance data of the serverless implementation and compare it with a set of competing systems.

This FMP has been developed as part of a grant at Universitat Rovira i Virgili's CLOUDLAB, and directed by Dr. Marc Sànchez Artigas. Additionaly, an important part of the work presented in this FMP is part of a research paper which is, at the moment of writing this paragraph, undergoing a blind review process.

1.2 Planning

We have developed this project in 18 months, during which we have divided the work in 3 stages that have been performed incrementally: research, development and evaluation.

- **Research**. During this stage, we searched for research papers regarding the topics of the project, as well as related work. We also earned the technical and conceptual skills required to implement part of the components of the system, especially the ML algorithms and communication, and looked for ways to introduce optimisations to the architecture's weaknesses inherited from FaaS.
- **Development**. After researching and understanding the fundamental concepts for the implementation, we implemented part of the final solution. Firstly, we implemented the serverless architecture. After that, we focused on the communications and the ML algorithms. Finally, we introduced a series of optimisations to overcome part of the challenges that FaaS introduces. Once the serverless implementation was finished, we focused on developing PyTorch and PyWren implementations to carry out a comparison.
- **Evaluation**. In this final stage we perform multiple checks to ensure that the implementation does not contain any problems. These checks include sanity checks to make sure the models of all implementations are equivalent, as well as performance benchmarks for

some of the architecture's components. Additionally, we carry out a set of performance evaluations comparing our implementation with the other two competing systems.

The result of this process is a serverless IBM-Cloud-based implementation of multiple ML algorithms capable of overcoming FaaS' limitations for ML training. We also provide implementations for PyTorch and PyWren-IBM Cloud to carry out a comparison in terms of execution time and loss.

Chapter 2

Background

Along this chapter, we provide the fundamental concepts of serverless computing, and its applications. Then, introduce the FaaS cloud paradigm, its properties, challenges and how it can be used in ML workflows. Additionally, we explain in detail the implemented ML algorithms used in the evaluation, along with their properties. Finally, we provide a description of the datasets used during the evaluation, along with details of the preprocessing techniques that have been applied on them.

2.1 Serverless computing

Serverless computing is a cloud execution model in which the provisioning, deployment and scaling of resources are taken care of by the cloud provider. In serverless, users only pay for the actual usage of the resources in what is known as a *pay-as-you-go* billing model. Serverless services offered by cloud providers include multiple kinds of resources:

- **Storage**. Cloud providers offer multiple serverless storage services which include relational databases (Amazon Aurora), object oriented storage (Amazon S3, IBM Cloud Object Storage) or key-value databases (serverless Redis instances).

- **Runtimes**. Serverless allows to run code in the cloud without explicitly provisioning servers. The main form of serverless runtime is FaaS, or Function as a Service. FaaS allows running user-defined functions in small stateless execution units with limited resources. There are multiple examples, such as Amazon Lambda or IBM Cloud Functions.

The main component we used to build the implementation is FaaS. The following subsection provides a thorough explanation of FaaS' properties and limitations for ML training.

2.1.1 FaaS for ML

FaaS presents some properties that benefit the training of ML algorithms:

- **High parallelism**. Cloud providers allow to run up to thousands of functions simultaneously. Choosing the right memory size, full vCPUs can be obtained, effectively providing thousands of cores. ML training tasks benefit from this high parallelism.

- **On-demand scaling**. ML learning on classic IaaS services require the user to monitor the usage of resources and scale them if needed. Moreover, resources are static, in a sense that they cannot be added or suppressed during training without manually turning servers on or off. Serverless, and especially FaaS, is more convenient for the user in terms of scalability, since it is taken care of automatically, and allows the user to cut costs paying only for the resources they actually use.

- **Multi-language support**. FaaS supports a wide variety of programming languages commonly used in ML. Most cloud providers support any language that can be compiled or interpreted within a Docker container. These languages include Python, C/C++ or Java, among others. Using the right tools, languages can be combined, providing high-level syntax and low-level performance (see 3.2.1).

However, despite its benefits, FaaS for ML has some potential drawbacks that require attention [3]:

- **Limited memory and CPU resources**. The amount of RAM available is limited in FaaS services. Analogously, cloud providers limit the CPU resources for functions. Moreover, the allocation of vCPU cores is tied to the amount of RAM that is ordered. For instance, IBM Cloud limits the RAM to 2GB, which provides the equivalent of a full vCPU core. ML training usually requires considerable amounts of RAM to keep the models and dataset. Hence, the problem must be partitioned accordingly so these elements fit into memory.

- **Short-lived functions**. Serverless functions have a limited execution time and are killed when this time is over, losing all the data that were not saved in an external storage. Thus, training needs to be efficient so when the functions are killed, the model is trained, or some kind 're-launch' mechanism should be introduced, where the model is stored and read from an external storage.

- **Unpredictable cold start times**. When running serverless functions for the first time, they may take from a few seconds to up to a minute to start. This happens because

serverless functions are executed in containers that need to be instantiated. This is known as cold start. When the functions have been run once, the containers are not deleted immediately, and they are considered to be 'warm', reducing the cold start times.

- **Indirect communication**. serverless functions require some sort of communication component in order to communicate between themselves. Thus, the communication between functions is indirect, and must go through the communication component, making it impossible to introduce more optimal network topologies for ML such as a ring to perform collective communications, e.g. AllReduce [8].

- **Lack of hardware acceleration**. ML training benefits from the high parallelism that GPUs provide. However, generally, cloud providers do not provide support for hardware acceleration such as GPUs.

In order to obtain an optimal solution, these issues need to be addressed. In section 3 we provide an implementation which attempts to sort them out.

2.2 ML algorithms

This section introduces two ML algorithms often found in the literature: **Logistic Regression** (LR) and **Probabilistic Matrix Factorisation** (PMF). These are the algorithms that we will implement and use to evaluate the performance of the proposed serverless-based implementation. LR is a **supervised** ML algorithm, that is, the data is labeled and it classifies samples based on some predefined classes, while PMF is **unsupervised**, meaning that the data is not labeled.

We have chosen to use variations of the **Stochastic Gradient Descent** (SGD) optimisation method to train the models. SGD is an iterative optimisation algorithm which tries to find the parameters (model) that minimise an objective function, updating the model along the process. The updates, or steps, are obtained by calculating a **gradient** and applying it on the model. The gradient is the derivative of the objective function with respect to its parameters (the model's current values and the mini-batch). The idea is to make the parameters converge step by step towards the objective function's minimum using the gradients. Figure 2.1 depicts this iterative process.

Since it was important for the implementation to understand how every calculation of the algorithm is performed, the following subsections present the models and the optimising methods and objective functions used, along with the most relevant calculations.

2.2.1 Logistic Regression

Logistic Regression is a supervised ML algorithm which is usually found in classifier systems. This algorithm is used as a method to predict the binary label of a set of samples [24]. In LR, the model is composed of an array of weights, where each weight corresponds to a variable in the dataset. The LR is named after the **logistic function**, a type of **sigmoid** function which takes a real input and outputs values between 0 and 1. The logistic function is used, in this particular case, to estimate the predictions $\hat{y}$ as seen below:

$$\hat{y}_i = S(W \cdot X_i)$$

where W is the array of weights, X_i is the ith sample of a mini-batch, $\hat{y}_i$ is the predicted label of the ith sample and S is the logistic function:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Objective function. In order to evaluate the training loss in this implementation and update the model, we minimise the **Binary Cross-Entropy** (BCE) objective function, which is commonly found in classification algorithms. BCE is calculated using the following formula:

$$BCE = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)]$$

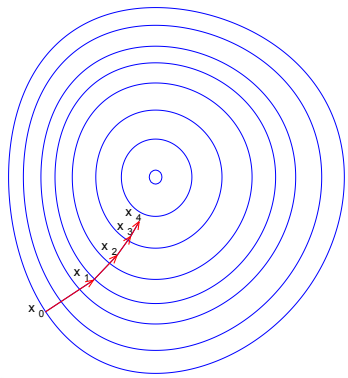


Figure 2.1: SGD stepping towards the optimal solution

which can also be written as:

$$BCE = -\frac{1}{n}\sum_{i=1}^{n}\begin{cases}\log(\hat{y}_i), & \text{if } y_i = 0\\ \log(1-\hat{y}_i), & \text{if } y_i = 1\end{cases}$$

where n is the mini-batch size and y_i is the real label of the ith sample of a mini-batch.

Optimiser. We used **Adam** as the optimiser in the LR implementation [11]. Adam does not apply gradients directly. It follows the same strategy as in SGD with momentum (see Optimiser in Probabilistic Matrix Factorisation), applying exponential moving averages of the past gradients to reduce gradient noise, but also computing the moving average of the past squared gradients as in RMSprop. These two elements are calculated are calculated as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\cdot\frac{1}{n}\sum_{i=1}^{n}\nabla f(W, X_i, y_i)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\cdot\frac{1}{n}\sum_{i=1}^{n}\nabla f(W, X_i, y_i)$$

where $\nabla f(W, X_i, y_i)$ is the gradient and ϵ is a small factor (usually 10^{-8}) which prevents a division by 0. This optimiser introduces two hyperparameters into the model: β_1 and β_2, which have default values of 0.9 and 0.999 respectively. These βs determine how many steps are taken into account to calculate the moving averages of the gradients. Before updating the model with the moving averages, a bias correction is applied:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

And the model is then updated, applying the learning rate or step size η:

$$W = W - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

As mentioned above, the objective function gradient is a measure of how much and towards where the model changes with regard to the change in the error. The gradient is used to update the model using a gradient-based optimiser. The gradient of a mini-batch is calculated as follows:

$$\nabla f(W, X_i, y_i) = (\hat{y}_i - y_i)\cdot X_i$$

Where W is the model and X_i and y_i are, respectively, the ith sample and label of a mini-batch.

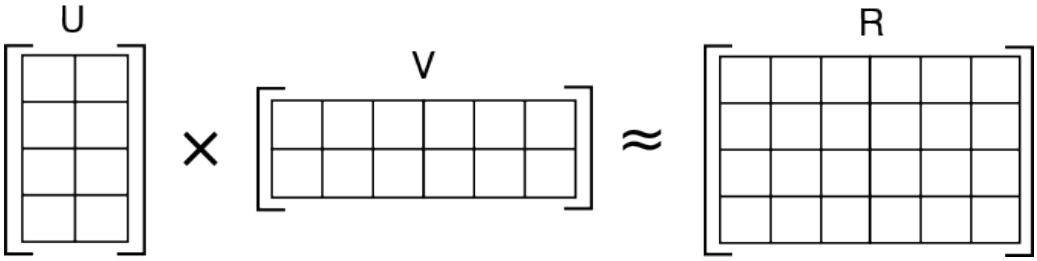


Figure 2.2: PMF model. U is the users matrix and V the items matrix. The product of these matrices results in an approximation of the original ratings matrix R.

2.2.2 Probabilistic Matrix Factorisation

Probabilistic Matrix Factorisation [13] is a collaborative filtering algorithm based on low-rank factor models, i.e., the decomposition of a higher rank matrix in two lower rank matrices, the product of which results in an approximation of the original matrix. PMF is often used to build recommender systems.

In PMF, the original ratings matrix of size $K \times M$, where K is the number of users and M the number of movies. The model decomposes the original matrix into two matrices that contain the trainable parameters: the users matrix U and the items matrix V. These matrices have $K \times d$ and $M \times d$ parameters each. The value of d is a hyperparameter of this model. Rows in the users matrix are known as user latent vectors, and rows in the items matrix are called item latent vectors.

Figure 2.2 shows a graphical representation of the PMF model.

Objective function. The objective function we minimise in PMF is the root mean squared error (**RMSE**). RMSE is the mean of the sum of the squared errors of the predictions. The error is calculated every step, for every mini-batch, using the following formula:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

Optimiser. To optimise the objective function, we use a mini-batch-based **Stochastic Gradient Descent (SGD) with Nesterov momentum** [21]. As mentioned before, SGD is an iterative optimisation algorithm for finding a local minimum in a function. In mini-batch SGD, the model is updated every iteration as follows:

$$W = W - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla f(W, X_i, y_i)$$

With mini-batch SGD, gradients are calculated from a small part of the dataset every step. This can lead to some noisy gradients that may generate updates that move the model in the wrong direction. In order to mitigate this problem and accelerate convergence, an optimisation is introduced: the **Nesterov momentum**. The Nesterov momentum updates the model using exponential moving averages of the gradients, instead of calculating and directly applying a gradient. Using this technique, the noise of the gradient is reduced, accelerating the convergence of the algorithm. In this case, the model update can be written as:

$$V_t = \beta V_{t-1} + \eta \frac{1}{n} \sum_{i=1}^{n} \nabla f(W - \beta V_{t-1}, X_i, y_i)$$

$$W = W - V_t$$

where V_t is the sequence that holds the exponential averages of the gradients at step t. Note that a hyperparameter is introduced: β, which defaults to 0.8 and determines how many points are used to calculate the moving average. The smaller the β, the more the gradient fluctuates. It is also worth mentioning that the Nesterov momentum does not directly apply the gradient's moving averages of the current step, which would be the classical momentum approach. Instead, it approximates the future position of the model by applying the accumulated gradient, and then calculating and applying a correction, or $\nabla f(W - \beta V_{t-1}, X_i, y_i)$ instead of $\nabla f(W, X_i, y_i)$.

The gradient of a mini-batch is calculated with the following formulae:

$$\nabla f(U, X_i, y_i) = 2(\hat{y}_i - y_i) \cdot U_k$$

$$\nabla f(V, X_i, y_i) = 2(\hat{y}_i - y_i) \cdot V_m$$

Where k is X_i's row and m is X_i's column. The prediction $\hat{y}_i$ is calculated as the scalar or dot product of the ith sample's row and column of U^T and V respectively. Thus, the prediction for a certain mini-batch can be written as:

$$\hat{y}_i = U_k^T V_m$$

2.3 Datasets and preprocessing

We have used multiple datasets during the evaluation of the proposed implementation. This section presents the most relevant details about them and provides the preprocessing techniques that are applied on them in order to prepare them for ML training.

2.3.1 Criteo

For testing both LR implementations, the **Criteo** display advertising challenge dataset [4] is used. The Criteo dataset contains 47 million samples, and each of them is made up of 13 numerical features, 26 categorical features and a binary class label. Before using Criteo for ML training purposes, some issues require attention:

Missing values. The Criteo dataset contains rows with missing values within some of its features. There are multiple techniques that can be applied to handle missing values [20]. For instance, missing values in numerical features can be replaced with a certain value, such as the feature's mean, median, etc. In categorical and numerical features, the rows containing missing features can be deleted, causing a considerable information loss, or predicted, which is costly but preserves information. Since the dense implementation of LR cannot handle missing values, they are are filled with zeros during the preprocessing. This technique works well with numerical features, ensures a low information loss and is simple to implement and run on large datasets such as this. The sparse implementation of LR, which also uses categorical features, can handle missing values, and no further transformations are required to be applied on the dataset.

Categorical features. As mentioned above, the Criteo dataset contains 26 categorical features per sample which need to be transformed to be used. To do so, we perform **feature hashing** or the "hashing trick" during preprocessing [22]. Feature hashing transforms categorical features into sparse vectors of numerical features. To do so, a hash function is applied to each categorical feature in every sample, Now, instead of having 26 categorical features, the Criteo dataset will have a certain number number of numerical features representing the categorical features. This number is a parameter, and in this particular case, 10^6 categorical features have been used, resulting in a dataset with $13 + 10^6$ numerical features.

Standardisation. Criteo dataset's features have uneven ranges, which could make the larger features shadow the effect of smaller features on the gradient calculation. If the algorithm was trained with the raw dataset, this could lead to a bad quality model. In order to avoid this potential problem, we standardise the dataset [23]. Standardisation is a form of feature scaling which centers the features around zero (zero-mean) and makes them have unit-variance. The calculation consists in obtaining every feature's mean ($\overline{x}$) and standard deviation (σ), and apply the following formula on every feature's values:

$$x' = \frac{x - \overline{x}}{\sigma}$$

where x' is the normalised value and x is the original value.

We have used this technique on the Criteo dataset including only the 13 numerical features. For the sparse implementation, a different scaling technique is applied.

Min-max scaling. Standardisation is useful when numerical data is expected to follow a Gaussian distribution. However, it is not likely that the numerical features obtained from applying the hashing trick on the categorical features will follow that distribution. Thus, we apply min-max scaling on the dataset, first calculating the minimum and maximum value of each variable and applying the following formula:

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

This form of scaling works well for any distribution, and also helps to reduce the differences in the ranges of the variables, scaling them between $[0, 1]$.

2.3.2 MovieLens

Finally, in order to test the PMF algorithm, we test two similar datasets: **MovieLens-10M** and **MovieLens-20M** [14]. Both datasets are made up of movie reviews from multiple users. The former contains 10 million reviews from 10,681 users on 71,567 movies, while the latter contains 20 million reviews from 27,278 users on 138,493 movies. Samples from these datasets consist of tuples of the form $\{user, movie, rating, timestamp\}$. We do not use the timestamps in the samples during training; hence, they are dropped during the dataset preprocessing.

Global mean and data centering. During preprocessing, we calculate the dataset's ratings' global mean, which is subtracted from each sample. This ensures that the ratings are centered around 0, which is required in PMF.

2.3.3 Dataset partitioning

There are multiple strategies when it comes to dataset partitioning. For instance, datasets can be loaded and partitioned in memory, which is common in classic ML workflows. However, this approach is not valid when it comes to ML on FaaS, due to memory constrains in serverless functions. Other approaches include partitioning the dataset in as many parts as workers, and every worker updates its local model using its assigned part. This approach also raises an issue: workers will train their local models differently, because they will never have access to all samples in the dataset, making it harder to converge.

To solve these problems, we shuffle the datasets, partition them in mini-batches locally, and upload them to a serverless cloud storage. Shuffling the dataset grants that the samples will be evenly distributed among the mini-batches. The training process follows a mini-batch-

based approach, where loss and gradients are computed and applied from a small sub-set of data every step. Moreover, All workers have an identical replica of a list containing the mini-batches' identifiers. When an epoch is over, that is when workers have finished processing all listed mini-batches collaboratively, the list is shuffled. This ensures that all workers at some point of the execution will have processed all mini-batches, reducing the error introduced by partitioning the dataset, and making convergence possible.

2.4 Competing systems

In Chapter 4 we carry out an evaluation of the proposed implementation. Part of this evaluation consists in a comparison with other systems in terms of execution time and convergence. In order to establish a fair comparison and study the evolution of loss during the training of the selected algorithms, we have chosen two frameworks with similar specifications to the proposed implementation's. These are a specialised deep learning framework, **PyTorch**, and a serverless framework for big data analytics, **PyWren-IBM Cloud**. We provide general specifications about these two frameworks in the following subsections, but specific implementation details are provided during the evaluation.

2.4.1 PyTorch

PyTorch is a widely-used open-source Python deep learning framework [16]. While it provides a Python frontend, its backend is fundamentally implemented in C/C++ to provide C performance while also providing a simple syntax. PyTorch supports both running ML training on both CPU and GPU. It also includes multiple implementations of well-known ML algorithms, objective functions and optimisers, and allows defining custom ones.

PyTorch's operation is based on a specialised data structure: **tensors**. Tensors are data structures similar to arrays which encode the model's inputs, outputs and parameters. Tensors contain elements of a single data type and support hundreds of simple operations such as products or sums, or even more complex operations such as applying trigonometrical functions, clipping, calculating the dot product, etc. Since PyTorch can run the ML training process in CPU or GPU environments, it provides tensor implementations for both.

Tensors are meant to be the data structure that sustain neural networks in PyTorch. Neural networks (NNs) are a collection of nested functions (layers) which are run on some input data [15]. NNs are composed by a set of trainable parameters that are part of the model: weights and biases.

The training process of NNs is divided in two steps:

- **Forward propagation**. The NN calculates predictions of the correct output using the given input data and its parameters. In this case, the desired ML model is used to obtain the predictions (in our case LR or PMF).

- **Backward propagation**. The predictions of the previous step are used to calculate the loss and, with the loss value, adjust (optimise) the NN's parameters, calculating a gradient and applying it. PyTorch provides a method to automate this calculation by setting the `autograd` parameter to `True` on the tensors that compose the NN.

PyTorch can be executed on a single computer or on a distributed environment. To do so, it provides distributed communications libraries which implement commonly used collective communication algorithms for ML.

Just as other ML frameworks, PyTorch allows storing trained and reading pre-trained models. Moreover, it includes modules such as *torchaudio*, *torchtext* or *torchvision* which provide pre-trained models for specific problems that have been successfully solved in the past.

2.4.2 PyWren-IBM Cloud

PyWren-IBM Cloud (IBM PyWren) is a Python cloud big data processing framework which allows to run user-defined Python functions within serverless functions in the cloud. It is based on the MapReduce architecture [5], and allows running up to a thousand parallel serverless map and reduce tasks. It was originally created as a fork of the original PyWren prototype for AWS (Amazon's cloud), adding new features and support for IBM Cloud. IBM PyWren's architecture consists of three main components:

- **Client**. The client is responsible of launching and monitoring jobs. The client serialises and uploads the user-defined functions, along with their dependencies, to the storage backend. It also takes care of evenly distributing the data among the functions.

  The client provides, mainly, three execution calls: `map`, `map_reduce` and `call_async`. The `map_reduce` call corresponds to a MapReduce execution, where some user-defined map functions are applied to the input data, generating intermediate results that are aggregated by a user-defined reduce functions. The `map` and `call_async` calls run a certain number of user-defined functions with the main difference being that `map` is blocking and `call_async` is not blocking and requires the programmer to call `get_results` or `wait` to obtain the results of the execution.

- **Runtime**. The runtime is responsible of executing the user-defined functions. It is based in Docker, which allows using custom images that contain libraries that do not

come installed by default in the IBM PyWren runtime. At the time of IBM PyWren's development, up to 1000 functions could be launched, with an execution time limited to 600s. Within the runtime run **workers**. Workers are the execution units responsible for deserialising the user-defined function, its dependencies and parameters, running it and storing the results.

- **Storage**. IBM PyWren relies on IBM Cloud Object Storage (COS) as the storage backend. It stores all the data, including the serialised user-defined functions and their dependencies, the intermediate results of the tasks and the final results. It is also used to communicate the functions, due to the lack of communication mentioned before.

Figure 2.3 shows PyWren's architecture.

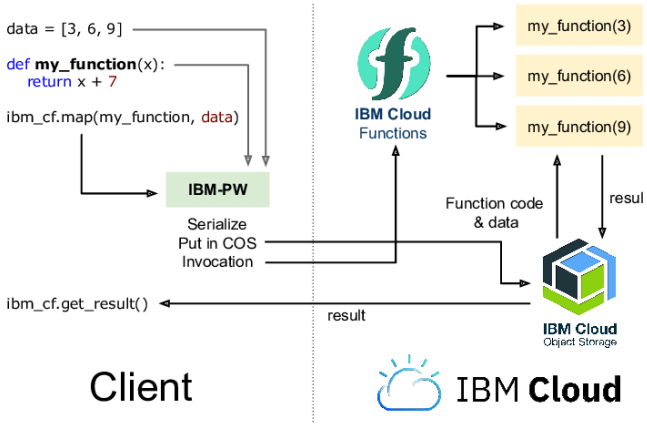


Figure 2.3: PyWren-IBM Cloud architecture overview [19].

Chapter 3

Implementation

In this section, we provide an overview of the architecture of the proposed serverless implementation, as well as technical details such as programming languages and the technologies used to build it. We also introduce some optimisations on the implementation and explain the impact they have on the performance.

3.1 Architecture overview

The proposed serverless architecture is implemented within the IBM Cloud, and is based on the following main components:

- **Client**. The client is in charge of launching and monitoring the execution. It also aggregates the results and provides execution statistics.
- **Serverless functions**. FaaS is the execution backend of the architecture. Workers are executed within functions, which are launched and monitored by the client. Serverless functions require communication servers in order to communicate.
- **Communication servers**. Serverless functions are unable to communicate directly among them. To provide communication, a RabbitMQ message queue server and a Redis server are introduced. The communication in this implementation includes sharing local model updates, synchronisation and aggregating the results.
- **Storage**. Datasets are uploaded to IBM Cloud Object Storage (COS). Workers download mini-batches directly from COS, and update the local model.

Figure 3.1 shows the architecture's components and how they interact.

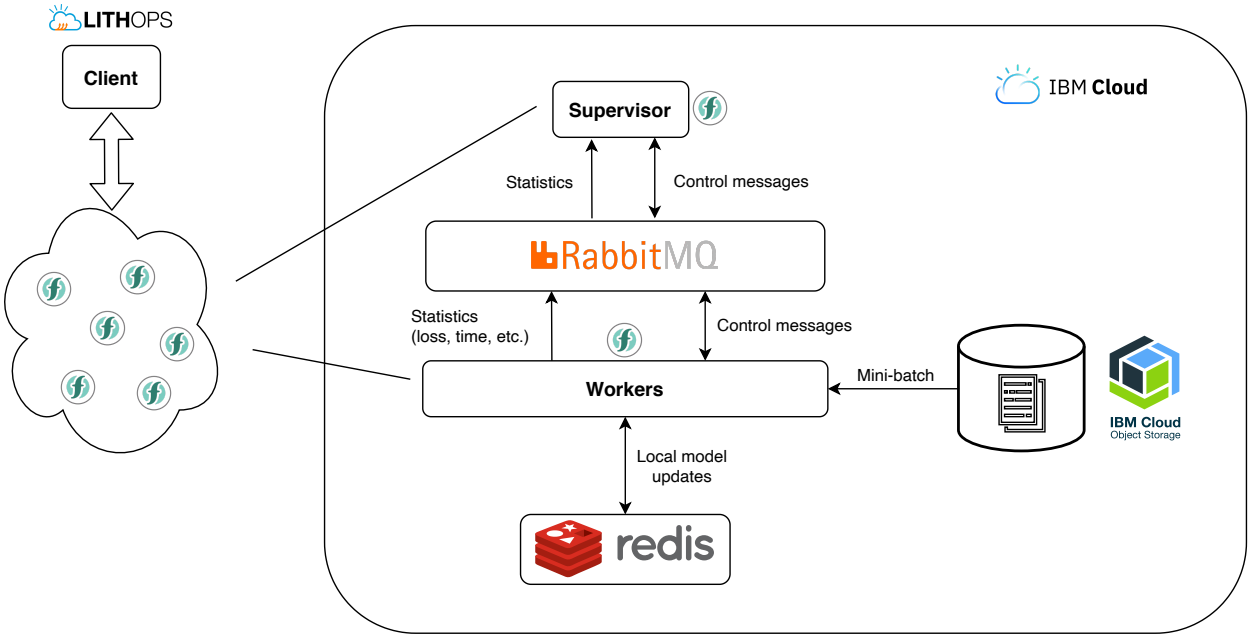


Figure 3.1: Serverless implementation architecture overview.

In the following subsections, we provide a thorough explanation of the above-mentioned components.

3.1.1 Client

The client is a part of the system which is run within the user's local machine and is responsible for launching executions, orchestrating the serverless functions, gathering and aggregating results and generating statistics. Most of these tasks are handled by **Lithops** [12], a Python serverless framework based on IBM PyWren which allows running custom Python code on multiple serverless platforms. Lithops provides a simple API for managing serverless jobs, which are run on a FaaS runtime in the cloud.

3.1.2 Serverless functions

Serverless functions are the execution backend of the architecture. Functions are run within a Docker container that executes Lithops' default Docker image. Docker is an open-source virtualisation software which allows running applications within an isolated environment called container [6]. Docker containers have some properties that make them suitable for serverless computing: Lithops supports custom Docker runtimes containing Python libraries that might be missing in the base runtime. In order to add support for the specific libraries that have been used to develop the serverless implementation, we created a custom Lithops Docker runtime.

There are two types of functions that are used during the training:

- **Worker**. Workers are responsible for training a local copy of the model and sharing the updates with their peers. Workers also collect step times which are sent to the supervisor or the client to generate statistics. There are multiple workers running in parallel, and the number of workers is a parameter that can be set in the client before training.

- **Supervisor**. The supervisor is in charge of execution control tasks such as synchronising workers and determining when the training is over, as well as merging local loss values and generating time and loss statistics. These control tasks performed by the supervisor could be done entirely by the client, but since the client is usually far from the data centre where the calculations take place, having a supervisor close to the workers speeds up the training.

The communication between workers, client and supervisor is performed through the communication servers.

3.1.3 Communication servers

Serverless functions are unable to communicate directly; hence, this implementation requires two communication servers in order to allow functions to share data between them: a **Redis** server and a **RabbitMQ**.

Redis: the Redis server is used to communicate local model updates among workers and sending the local models to the client when the training is over. Redis is an in-memory data structure store, which is often used as a database, cache or message broker [18]. Redis supports multiple data structures such as lists, strings or values, among others. It also provides simple operations such as `set`, `get` or increasing counters, or even more complex operations which can execute external Lua scripts. For this implementation, only simple operations are used, since the data stored in Redis is previously serialised, which allows storing it as a value.

When a step is over, workers share their local gradients using the Redis server, carrying out an AllReduce-like collective communication. The Redis application is hosted in an IBM Virtual Server instance, but we could replace it for a Redis serverless instance available on some cloud providers.

We chose to use Redis as the backend for sharing local model updates due to some of its properties. Firstly, it is open-source, well documented and its deployment is straightforward. Moreover, since Redis keeps its contents loaded in RAM as long as they fit, the latency for accessing data is in the order of sub-milliseconds, which makes it ideal for the task at hand.

Additionally, we chose Redis because it can be easily replaced with a serverless instance, depending on the cloud provider's availability for serverless Redis instances.

RabbitMQ: the RabbitMQ server is responsible for communicating the workers with the supervisor and the client. RabbitMQ is an open-source message broker software [17]. It supports multiple patterns of communication, from which we used direct communication (sending messages directly to queues) and the publish-subscribe pattern, where a producer publishes messages to an exchange, which sends them to all the queues that were subscribed to the exchange. Consumers will consume the messages from their respective queues. Figure 3.2 shows the publish-subscribe pattern on RabbitMQ.

Workers, supervisor and client have message queues assigned, where they receive and consume the messages. Some messages are broadcast through a fanout exchange, while others are sent directly to a specific queue using the queue's routing key. The messages implemented are mostly control messages and include:

- **Worker messages**. Workers send **step end messages** to the supervisor when a step is over. These messages contain the iteration loss and iteration time. When workers are terminated by the supervisor at the end of the execution, they send the client a **results** message, telling it that the results are available in Redis for downloading.

- **Supervisor messages**. The supervisor implements a **barrier** that stalls the workers at the end of the step until the supervisor has received their step end messages for the current step. When the condition is met, the supervisor then sends a **continue** message if the execution end criteria is not met. In the opposite case, an **end message** is sent to all the workers, finishing the execution. The end criteria include an execution time limit, a maximum number of epochs or reaching a loss threshold. The supervisor also sends the client the **execution times** and **loss history** in a message when the execution is over.

This component is also implemented using an IBM Virtual Server instance. Though this is

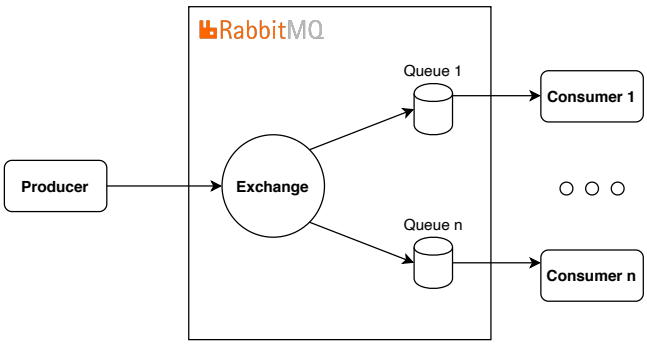


Figure 3.2: Publish-subscribe pattern on RabbitMQ.

a serverful component, some cloud providers offer a serverless message queue system, such as AWS SQS [2], which could bring this implementation closer to a fully serverless implementation.

The system could work using only the Redis server for communications, but a number of issues arise. Firstly, workers would have to periodically poll the Redis server to look for local model updates from other workers or control messages from the supervisor. Additionally, Redis is a single-threaded application. Workers continuously polling Redis would greatly reduce the overall performance of the architecture. Hence, we introduced the RabbitMQ server to handle control messages and relieve these issues.

3.1.4 Storage

As seen in the previous chapter, datasets are partitioned in mini-batches, which require being stored in a Cloud storage service in order to be processed by the workers. Since the chosen Cloud provider for the implementation is IBM Cloud, an IBM Cloud Object Storage (COS) instance has been used as the storage backend. The mini-batches stored in COS have been previously serialised compressed using Python's **pickle** and **zlib** respectively.

3.2 Optimisations

The architecture that we have just described would be sufficient to run ML algorithm training tasks in the cloud, but there is still a margin for improvement in terms of performance. Hence, we introduce two optimisations: Cython and sparse data structures to share model updates.

3.2.1 Cython

Python's syntax and ease of setup and use has attracted data scientists. However, most Python ML frameworks are not fully implemented in this language. Python is considered to be an unsuitable programming language to build ML frameworks from scratch. Usually, Python ML frameworks provide a Python front end which interacts with compiled optimised C/C++ implementations. This allows obtaining a high performance from lower level programming languages, while providing a relatively simple, high-level syntax for running ML workflows.

Following this trend, the proposed implementation provides a simple Python front end which allows instantiating, running and obtaining statistics from the algorithms, while the most critical parts such as the models and their methods are implemented in Cython. Cython is a static compiler for Python that allows writing code using mostly Python syntax, but introducing variable typing, loop and array access optimisations and C/C++ libraries. This code is compiled with optimisations and added to a custom Lithops Docker runtime.

The custom Docker runtime is based on Lithops' default runtime, which is also based in a Linux image. Hence, the Cython code must be compiled on a Docker container running, at least, a Linux image with all the project requirements installed. We provide compile scripts to ease this process.

3.2.2 Handling sparsity

The datasets described in the previous section are highly sparse. Over 10^6 numerical features compose the Criteo dataset's samples after the preprocessing. However, samples in the dataset only contain 13 purely numerical features and 26 categorical features transformed in numerical after the hashing trick, leaving almost a million zeros in every sample. In both MovieLens datasets, users do not rate all the movies in the catalogue. In fact, most users in the MovieLens datasets only rate a few movies, leaving tens of thousands of movies unrated.

Since the nature of the serverless functions does not allow direct communication between them, the communication of model updates must be as optimal as possible to avoid a bottleneck caused by the Redis server, responsible for handling this process. To do so, updates are sent in a sparse structure resembling a COO matrix (Table 3.1). For instance, a sparse matrix like the following:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 2 & 0 & 0 & 0 \end{pmatrix}$$

can be compressed using the sparse data structure as follows:

Row	0	1	3	3
Column	1	0	0	1
Value	1	3	4	2

Table 3.1: Sparse representation of a matrix using a sparse data structure.

This way, only those parameter values different from 0 will be sent, relieving the Redis server usage considerably.

Chapter 4

Evaluation

In order to obtain performance results of the serverless implementation that we have provided, we carry out a series of experiments. During this chapter we provide the methodology followed to obtain results, including the resources that we have used and introducing the systems that are compared. Moreover, we run multiple benchmarks to ensure that the serverless implementation and its components scale properly. Finally, we provide a comparison in terms of convergence and execution time between various similar systems.

4.1 Methodology

Competing systems. In order to establish a comparison in terms of execution time and convergence and study the evolution of loss during the training of the selected algorithms, two frameworks with similar specifications to the proposed implementation's have been chosen. These are a specialised ML framework, **PyTorch**, and a serverless framework for big data analytics, **PyWren-IBM Cloud**.

- **PyTorch** is a well-known ML framework which provides a simple Python interface to an underlying optimised C++ implementation of ML algorithms. PyTorch provides all the algorithms, objective functions and optimisers described in previous sections. PyTorch often runs the training process in GPUs, but also provides support for CPU execution. To make a fair comparison, the training is executed distributed CPU-based environment, since IBM Cloud Functions do not provide any form of hardware acceleration, including GPU acceleration. The communication backend in the PyTorch CPU implementation is configurable, but it is recommended to use Gloo [10]. Gloo is a collective communications library which includes collective communication algorithms usually found in ML training. In this particular case, workers average their local gradients and loss values using an

`AllReduce` Gloo call. Datasets are read from IBM Cloud Object Storage (COS), as in the serverless implementation.

- **PyWren-IBM Cloud** [19] is a serverless data analytics Python framework. PyWren allows running user-defined functions on a serverless environment within the IBM Cloud. PyWren-IBM Cloud can be considered a baseline, non-optimised version of the serverless implementation we propose, since Lithops was originally built on top of PyWren. This implementation is based on the **MapReduce** architecture [5]. Workers are launched as map functions, while a reduce function is also invoked to aggregate the local models of the workers. This process is carried out every step. Workers communicate with the reduce function and read their mini-batches from IBM COS.

IBM Cloud resources. In order to carry out experiments, a series of IBM Cloud resources have been used. As seen in the Architecture overview, part of these resources do not require provisioning due to their serverless nature, e.g. Cloud Functions or COS. However, other serverful components such as Redis and RabbitMQ require dedicated virtual server instances. Additionally, the distributed CPU-based PyTorch implementation is executed in a cluster of virtual servers. Table 4.1 provides the specifications of the virtual server instances used in the evaluation.

Instance	vCPU	RAM (GB)	NIC Bandwidth (Mbps)	Description
M1.2x16	2	16	1000	Redis server
C1.4x4	4	4	1000	RabbitMQ server
B1.4x8	4	8	1000	Pytorch cluster (x6)

Table 4.1: IBM Cloud virtual server instances' specifications.

We launch the Cloud Functions in the custom Lithops runtime described in the previous chapter with 2GB of RAM. We choose to use 2GB to obtain a full vCPU and make FaaS workers equivalent to PyTorch's CPU workers.

4.2 Compute scalability

Scalability is an important aspect in distributed ML architectures. In order to test the scalability of IBM Cloud's Functions, a simple experiment has been executed. The experiment consists in measuring the number of model updates per second without considering communication on different number of worker configurations. The workers are launched with 2GB of RAM to ensure receiving the equivalent of a vCPU.

The results in Figure 4.1 proof that, for every number of workers, serverless functions scale almost linearly in terms of model updates/s. Considering the standard deviation of the bars, there appears to be a certain variability. This is, most likely, due to the unpredictable cold start times of functions and the allocation of resources.

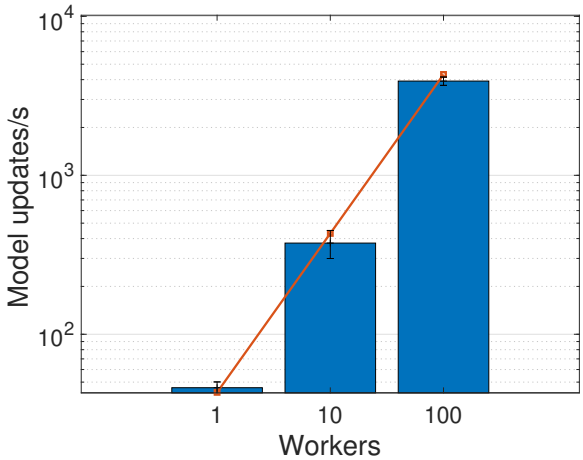


Figure 4.1: IBM Cloud Functions' scalability. The orange line represents the ideal linear scalability.

4.3 Redis benchmark

Redis is the main component responsible of the communication of local model updates during training; hence, a basic benchmark is implemented to asses its performance. The main goal is to determine whether the selected RAM, vCPU and bandwidth configuration is suitable for handling the traffic generated during the execution of the evaluation experiments.

This benchmark measures both read and write throughputs in MB/s while performing read/write operations, and varying the number of workers between 5 and 50. The size of the objects used in the evaluations varies between 0.01 and 25MB.

Observing Figure 4.2, it is clear that up to 25 workers, and up to an object size of 10MB, the selected instance provides a good performance. Increasing the number o workers or the object size any further degrades the writing performance, while the reading performance keeps scaling. Thus, the selected configuration appears to be appropriate for experiments up to 25 workers and 10MB objects.

Moreover, a peculiar result arises from the experiment: the maximum throughput when reading is close to 800 MB/s, but the network bandwidth is theoretically capped to 1Gbps. This could mean that IBM might actually be providing a higher bandwidth when there are idle network resources. To prove this, a network evaluation was carried out. The network was

tested using a 1Gbps private network connecting two identical virtual server instances. To carry out the evaluation, a well-known bandwidth measuring tool called **iperf3** [7] was used. After multiple measurements, the average bandwidth obtained was 6.1Gbps, meaning that the real bandwidth used by the Redis server during the previous evaluation whas significantly higher than 1Gbps.

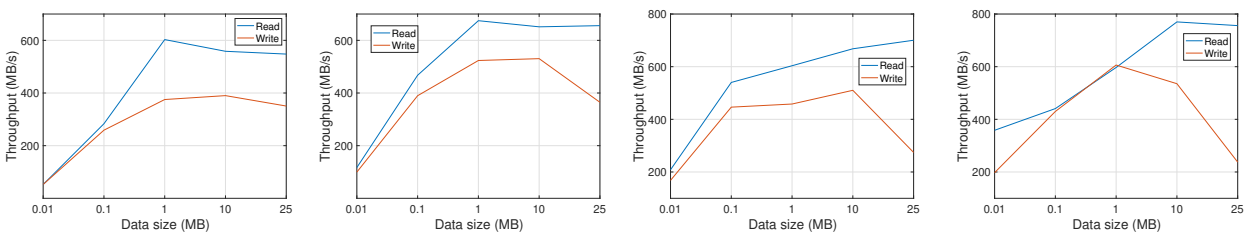
However, if we were to launch larger experiments in terms of model updates' size or number of workers, the selected virtual server instance may not be powerful enough due to scaling issues. In order to solve this problem, two different strategies could be followed: scale the virtual server **vertically**; thus, increasing the memory (Redis will not benefit from increasing the number of vCPUs, since it is a single-threaded application), or scaling **horizontally**, increasing the number of Redis instances and applying sharding. Nevertheless, it is not expected that experiments with more than 25 workers and model sizes bigger than 15MB will be launched during the evaluation.

4.4 Performance comparison

To compare the performance of the proposed implementation with the above-mentioned competing systems, we carry out an evaluation of the training loss and execution time. Before running the final experiments, we have performed a sanity check to make sure that the ML algorithms' implementations in the proposed implementation and PyTorch are equivalent.

4.4.1 Experiment parameters

We launched 4 experiments, which have some parameters in common: all of them have been run with 24 workers and a supervisor, and the stop criteria is running for 480s. Particular hyperparamters from every algorithm are omitted, since most of them are set to their default values. These parameters apply to all the implementations (serverless, PyTorch and PyWren). Table 4.2 shows a summary of the experiments that we launched and their parameters.



(a) 5 workers (b) 10 workers (c) 25 workers (d) 50 workers

Figure 4.2: Redis throughput for multiple worker sizes.

Algorithm and dataset	Workers	Mini-batch size	Learning rate η
LR - Criteo (13 numerical features)	24	6250	0.0075
Sparse LR - Criteo (10^6 hashed features)	24	6250	0.0075
PMF - ML-10M	24	6250	8.0
PMF - ML-20M	24	12000	8.0

Table 4.2: Experimental parameters summary.

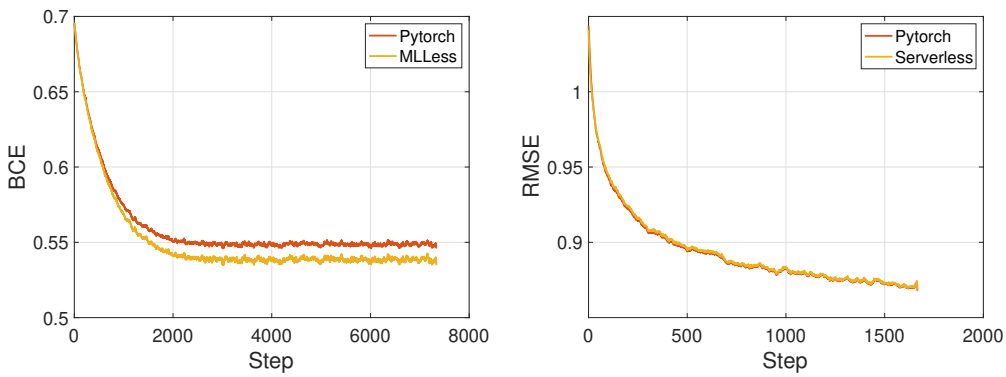
4.4.2 Sanity check

In order to ensure the comparability of the implemented algorithms, a sanity check is performed. The sanity check consists in running all implementations with a single-worker configuration for a whole epoch and observe the behaviour of the loss curves. The models will be considered identical if they behave similarly.

It is worth mentioning that it was not possible to run PyWren for the test due to how slowly it performs the steps. However, the PyWren implementations are, in terms of the models, a non-optimised version of the proposed serverless implementation. By looking at Figure 4.3, we observe that PyTorch and the proposed implementation perform similarly with a small gap that slightly benefits the PyTorch implementation in PMF, and our serverless implementation in LR, probably due to numerical precision; hence, we can conclude that the models are equivalent.

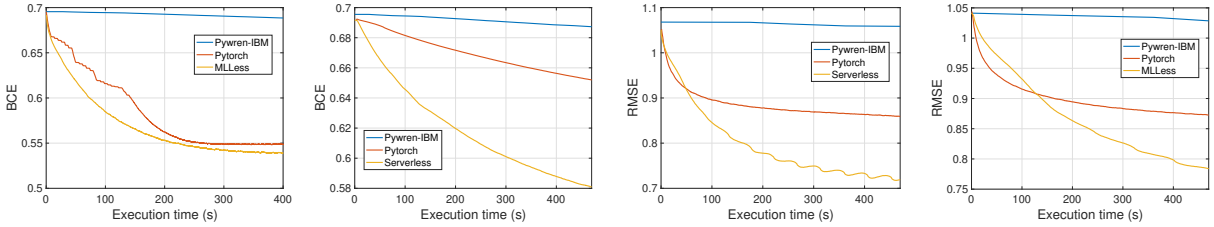
4.4.3 Comparison results

Once the models have been proven to be equivalent, a fair comparison can be established. The aim of this comparison is to determine how far the implementations can converge in a period of time. To do so, the loss is measured for all the implementations for 480s, and the curves are



(a) Criteo (LR) (b) MovieLens-20M (PMF)

Figure 4.3: Sanity check for the LR and PMF algorithms.



(a) Criteo (b) Criteo (sparse) (c) MovieLens-10M (d) MovieLens-20M

Figure 4.4: Loss vs. execution time comparison among PyTorch, PyWren-IBM and MLLess for 24 workers.

plotted and compared. The results seen in Figure 4.4 are clear. Firstly, PyWren-IBM Cloud is inefficient in all scenarios. Since it is not optimised, especially for handling sparsity, it is unable to keep up with PyTorch or the serverless implementation.

Secondly, the proposed implementation outperforms PyTorch and PyWren in most executions, proving that FaaS can be a serverless alternative to classic ML frameworks and computing cluster ML architectures.

However, there is one case in which the serverless implementation falls short when compared to PyTorch: dense LR. The datasets chosen for the experiments are highly sparse, and the ML algorithms used are meant to handle sparsity efficiently. Nevertheless, PyTorch and other specialised ML frameworks seem to suffer performance issues when working with highly sparse data [1]. If we take a look at the dense LR, there seems to be an issue during the first 150s of execution, probably due to a problem related with IBM COS. Nevertheless, PyTorch converges significantly faster considering the gap with our implementation seen in the sanity check. In normal conditions, PyTorch would beat the serverless implementation for dense Criteo with numerical features.

Chapter 5

Conclusion

Along this last chapter, and to sum up the work presented during the FMP, we enumerate a series of conclusions and propose future work that can be performed from the results obtained in the evaluation.

5.1 Conclusion

The main goal of this FMP was to implement and test a series of ML algorithms and running them on a serverless architecture in order to obtain results that could be compared with other well-known frameworks.

To do so, we have firstly introduced the fundamental concepts regarding serverless computing, FaaS, ML, dataset preprocessing and the technologies used. Taking into account the limitations of FaaS, we have provided a convenient serverless, FaaS-based implementation capable of running popular ML algorithms and scaling up to dozens of workers.

Once the system has been implemented, we have carried out a series of experiments to evaluate the performance of the serverless implementation, and provide a comparison with other systems. The results obtained in the evaluation prove that, with the optimisations introduced by the proposed implementation, that serverless-based ML can be suitable for running ML workflows, providing faster training compared to classic IaaS approaches, high parallelism and dedicating less time and resources to provisioning and scaling.

Despite the positive results, we have encountered some issues, especially related with the scaling of the communication process of local model updates. This appears to be the bottleneck in the serverless architecture, especially when running non-sparse ML algorithms. These results, along with some unexplored properties of FaaS, lay the ground for future work in order to sort them out.

5.2 Future work

From the results we have obtained during the evaluation, there are multiple topics that are open for future work.

Firstly, the algorithms that we have implemented and tested are meant to work with sparse data, such as the datasets that we have used. The results from LR's performance comparison show that our serverless implementation struggles with dense algorithms and datasets. In the future, it would be interesting to assess the effect of working on dense data on the proposed architecture and how to tackle them.

Additionally, the Redis throughput test showed that the Redis server could scale up to 25 workers and 10MB updates with no issues whatsoever, but seems to struggle when increasing the parameters further. It may be interesting exploring ways of scaling the sharing of updates such as using Redis sharding techniques or looking for a different backend.

We could also introduce some optimisations to the way workers synchronise. The proposed serverless implementation is synchronous, meaning that it relies on a barrier which stalls workers at the end of every step. Other options exist, such as an SSP implementation [9], where workers can perform steps freely and only synchronise every few steps, relieving the effect of the barrier.

Moreover, some of FaaS' most relevant properties have not been explored. FaaS' billing model is a *pay-as-you-go* model, where the user only pays for those resources they actually use. It would be interesting to compare our implementation with other similar competing systems not only in terms of performance, but also in terms of cost.

Bibliography

[1] *DLP-KDD '19: Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, New York, NY, USA, 2019. Association for Computing Machinery.

[2] AWS. Aws sqs. https://aws.amazon.com/es/sqs/.

[3] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Criteo. Criteo display advertising challenge dataset. http://labs.criteo.com/2014/02/kaggle-display-advertising- challenge-dataset/.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[6] Docker. Docker overview. https://docs.docker.com/get-started/overview/.

[7] Esnet. iperf. https://github.com/esnet/iperf.

[8] Andrew Gibiansky. Bringing hpc techniques to deep learning. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/.

[9] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

[10] Facebook Incubator. Gloo. https://github.com/facebookincubator/gloo.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[12] Lithops. Lithops. https://github.com/lithops-cloud/lithops.

[13] Andriy Mnih and Russ R Salakhutdinov. Probabilistic matrix factorization. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008.

[14] Movielens. Movielens datasets. https://grouplens.org/datasets/movielens/.

[15] PyTorch. Pytorch. autograd tutorial. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html.

[16] PyTorch. Pytorch. https://pytorch.org.

[17] RabbitMQ. Rabbitmq. https://www.rabbitmq.com.

[18] Redis. Redis. https://redis.io.

[19] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 1–8, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Towards Data Science. 7 ways to handle missing values in machine learning. https://towardsdatascience.com/7-ways-to-handle-missing-values-in-machine-learning-1a6326adf79e.

[21] Towards Data Science. Sgd with momentum. https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d.

[22] Wikipedia. Feature hashing. https://en.wikipedia.org/wiki/feature_hashing.

[23] Wikipedia. Feature scaling. https://en.wikipedia.org/wiki/feature_scaling.

[24] Wikipedia. Logistic regression. https://en.wikipedia.org/wiki/logistic_regression.