TÉCNICO LISBOA

Serverless Machine Learning

António Pedro de Carvalho Elias

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno

Examination Committee

Chairperson: Prof. Alberto Abad Gareta
Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Member of the Committee: Prof. Paolo Romano

October 2023

Abstract

Cloud machine learning has been traditionally done using IaaS offerings, which are hard to manage and slow to start-up, or using MLaaS offerings, that facilitate the management of the infrastructure, but are still slow to start-up and are costly. Serverless is a promising alternative since it takes care of resource management while providing high flexibility and scalability, but due to the difficulty in communication between functions, it is usually slower. Machine Learning models also need to adapt to changes, reflecting new trends, and requiring frequent retraining, which increases their costs. Using online learning, we propose a solution where the model is trained incrementally, as data arrives irregularly, as a way to take full advantage of the serverless paradigm and deliver models capable of dealing with concept drift faster while being cheaper than an equivalent deployment in IaaS in situations where the data arrives sporadically.

Keywords

Serverless; Function as a Service; Cloud Computing; Machine Learning; Online Machine Learning.

Resumo

A aprendizagem automática na nuvem é, tradicionalmente, efectuada utilizando ofertas de IaaS, que são difíceis de gerir e de arranque lento, ou utilizando ofertas de MLaaS, que facilitam a gestão da infraestrutura, mas que continuam a ser de arranque lento e são dispendiosas. A tecnologia sem servidor é uma alternativa promissora, uma vez que se encarrega da gestão dos recursos, proporcionando simultaneamente uma elevada flexibilidade e escalabilidade, mas, devido à dificuldade de comunicação entre funções, é normalmente mais lenta. Os modelos de aprendizagem automática também precisam de se adaptar às mudanças, reflectindo as novas tendências e exigindo treino frequente, o que aumenta os seus custos. Utilizando aprendizagem sem servidor, propomos uma solução em que o modelo é treinado de forma incremental, à medida que os dados chegam de forma irregular, como forma de tirar o máximo partido do paradigma sem servidor e fornecer modelos capazes de lidar com a alteração dos conceitos mais rapidamente, sendo mais baratos do que uma implementação equivalente em IaaS em situações em que os dados chegam esporadicamente.

Palavras Chave

Sem Servidor; Função como Serviço; Computação em Nuvem; Aprendizagem Automática; Aprendizagem Automática *Online*.

Contents

List of Figures

List of Tables

Listings

Acronyms

AWS	Amazon Web Services
DLRM	Deep Learning Recommendation Model
FaaS	Function as a Service
IaaS	Infrastructure as a Service
ML	Machine Learning
MLP	Multilayer perceptron
MLaaS	Machine Learning as a Service
PaaS	Platform as a Service
RMSE	Root Mean Square Error
S3	Simple Storage Service
VM	Virtual Machine

1 Introduction

Contents

1.1 Context

Machine Learning (ML) is a prevailing technique that is used to power many real-world applications spanning from recommender systems [4] to self-driving cars [5]. Machine Learning training algorithms are often run in the cloud, due to their computational complexity, using, for example, Infrastructure as a Service (IaaS) offerings. In this setting, the user, usually a Machine Learning expert and unfamiliar with cloud resource management, is forced to allocate virtual machines and deploy all the necessary software to train and serve the model. IaaS Machine Learning training is therefore hard to manage which leads to over-provisioning or under-provisioning. It also has a slow start-up time. In sum, it is time-consuming and error-prone.

A common alternative to this offering is to use Machine Learning as a Service (MLaaS), services that handle resources of the whole ML pipeline. With this type of solution, the user does not have to worry about any of that since the cloud provider handles it. This solution also has some problems. It is not flexible enough for some more complex Machine Learning jobs, including those that need some pre- or after-processing that are not ML. It also leads to vendor lock-in and is more expensive than IaaS. Both IaaS and MLaaS however, are based on Virtual Machines (VMs) which are slow to setup leading to a slow start-up of training when using these cloud deployment models.

A possible alternative to both of these approaches is to use serverless computing. In this model, the user writes the functions that he/she wants to execute, defines the events that trigger the code execution and the cloud provider handles all of the infrastructure management. Using serverless we can bring together the benefits of IaaS and MLaaS. In short, it is possible to have a general-purpose computing system that has easy integration with already existing applications, and automatic infrastructure management.

At the same time, in real-world systems, data often arrives incrementally and data distributions change over time, making it necessary to retrain the models if we want to take advantage of those data. Retraining ML models from scratch can be expensive and time-consuming which can lead to it happening less often than would be ideal. This is especially the case in recommender systems, where it is needed to provide relevant recommendations to users when trends and their likings are constantly changing [6].

1.2 Problem

Machine Learning training algorithms are usually iterative algorithms that go through the data as they update the model until a condition is met. Leading to intensive computation, that relies heavily upon communication between the nodes to share the model. Because of that, it has been shown that it is generally expensive and slow to train ML models in serverless.

Another side of the problem is that, as data is generated and its distribution shifts, the performance of the model degrades. Retraining often is expensive and training incrementally using IaaS can lead to over-provisioning of resources when the training system is waiting for more data or to slow training due to the start-up time of the infrastructure.

1.3 Goal

The goal of this work is to study the viability of serverless computing for Machine Learning training to find in which use cases, models, and situations it makes sense to use this model instead of the alternatives. We aim to measure the performance of these models and compare them with their IaaS counterparts.

1.4 Solution

By designing and implementing a serverless system that can train and serve Machine Learning models where the data arrives incrementally, we can address some of the limitations of the serverless infrastructure while providing training results sooner than with the other options. Using this system we can evaluate when it makes sense to use serverless to train online learning models compared to the existing approaches to train models in this setting or if it still makes sense to retrain the models using batch learning approaches in IaaS or MLaaS.

The solution is focused on models that require frequent updates. For these scenarios allocating IaaS virtual machines as needed or calling MLaaS jobs for each update would be slow and expensive, but just calling a function is fast and cheap.

1.5 Organization of the Document

This thesis is organized as follows: Chapter 1 introduces the thesis, then, Chapter 2 provides an overview of the relevant concepts to understand the rest of the work, both in Machine Learning and cloud computing. Then, Chapter 3, analyses state-of-the-art works that are mostly related to this topic. Chapter 4 describes the architecture of the proposed solution, while chapter 5 describes its implementation, and Chapter 6 describes and analyzes the experiments and their results. Chapter 7 concludes the thesis, and discusses future work directions.

2

Background

Contents

This chapter discusses some important concepts to fully understand the context of the problem. First, it introduces some of the basics of Machine Learning that are more relevant to this work (Section 2.1). Subsequently, the discussion covers the most common cloud delivery models (Section 2.2). And finally, concludes with a comparison of the different ways machine learning can be done using cloud infrastructure (Section 2.3).

2.1 Machine Learning

Machine Learning is the branch of computer science that studies how to make a computer learn from data. This field has been increasing its impact over the last few years with applications in different sectors that have a vast range, from recommendation systems in online stores or streaming services to image recognition or self-driving cars.

The Machine Learning process is divided into two distinct parts, training, and inference. During training, we train a model of the data using a learning algorithm. Most learning algorithms are iterative, meaning that they look at the data, or a portion of it at a time, update the model and do it repeatedly until the model converges to something that meets a threshold in a performance metric. During inference, that model is used to make predictions on new data.

Machine Learning is divided into supervised and unsupervised learning. The main difference between them is that in supervised learning we give data to the learning algorithm with labels from which we wish the algorithm learns. In Unsupervised Learning, the algorithm has to infer how to group the data. Supervised machine learning models are used to solve one of two different problems. In classification, the model assigns each piece of data into predefined categories. While regression aims at predicting continuous numerical values based on the input [7]. Another fundamental concept for this project is the Recommender System. These systems make personalized recommendations for individual users [7], so these systems have particular challenges to them.

2.1.1 Concept Drift

Real-world machine learning models are subject to Concept Drift. This means that as more data is generated, the data distribution changes. This can be for example because user behaviour changes or different internet trends emerge. Concept drift leads to models that are not prepared to deal with this change will suffer a decreasing performance as this newer data does not look the same as the older data. Models that deal with real-world data that change over time must be able to deal with this problem.

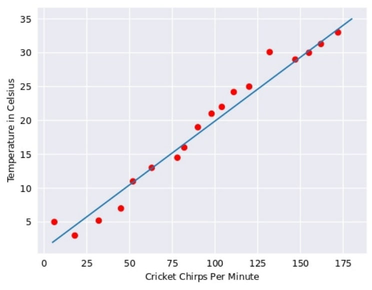
2.1.2 Models

We are going to discuss some of the most relevant ML models. Examples of some of these models are illustrated in Figure 2.1.

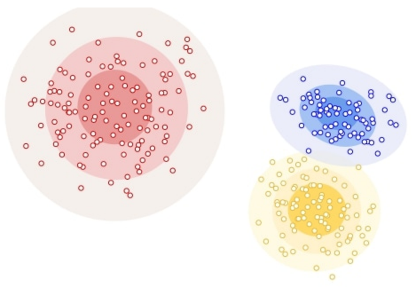
Linear Models are the models that try to either separate in the case of classification problems or estimate a value in the case of regression problems. This type of model tries to learn how the data behaves and the model is a linear function that describes it. An example can be seen in Figure 2.1(a).

Decision Trees learn a set of rules from the training data which are assembled in a tree-like model. When making an inference, the algorithm goes through the tree, following the branches dictated by the features of the data until it reaches the leaves, where there is a label that is assigned to this value. The results of this model are easier to understand for humans but it can start to become too complex very quickly. An example can be seen in Figure 2.1(c).

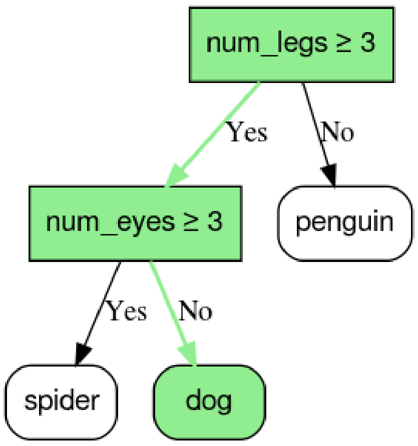
Neural Networks try to replicate the human brain. It has several neurons connected to each other and outputs the result based on their interactions. If we increase the size of this model, by adding more



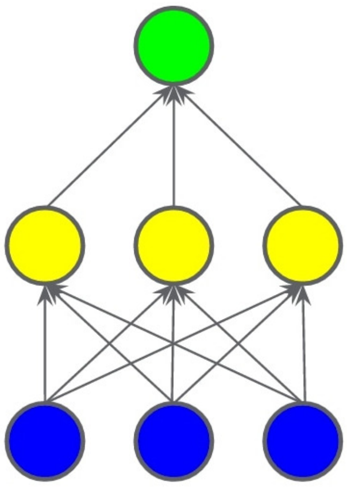
(a) A simple linear model [8].



(b) An example of clustering [9].



(c) An example of a decision tree [10].



(d) The internal structure of a neural network [11].

Figure 2.1: Examples of simple ML models.

layers between the neurons or adding different connections between them we can have what is called a deep neural network, this type of model can solve problems with larger complexity such as image classification or natural language processing. An example can be seen in Figure 2.1(d).

Probabilistic Matrix Factorization [12] attempts to find the missing values of a matrix. This model is often used in recommender systems, where the matrix is, for example, the rating each user gave to each movie and the model tries to guess what the missing ratings are by capturing the iterations between the users and the items. This technique can be combined with user and item biases to also capture variations in the rating based only on the user or an item, such as one movie being subjectively better classified by many different users for example [13].

Clustering algorithms are used in unsupervised learning. Since the data, in this case, is not labelled,

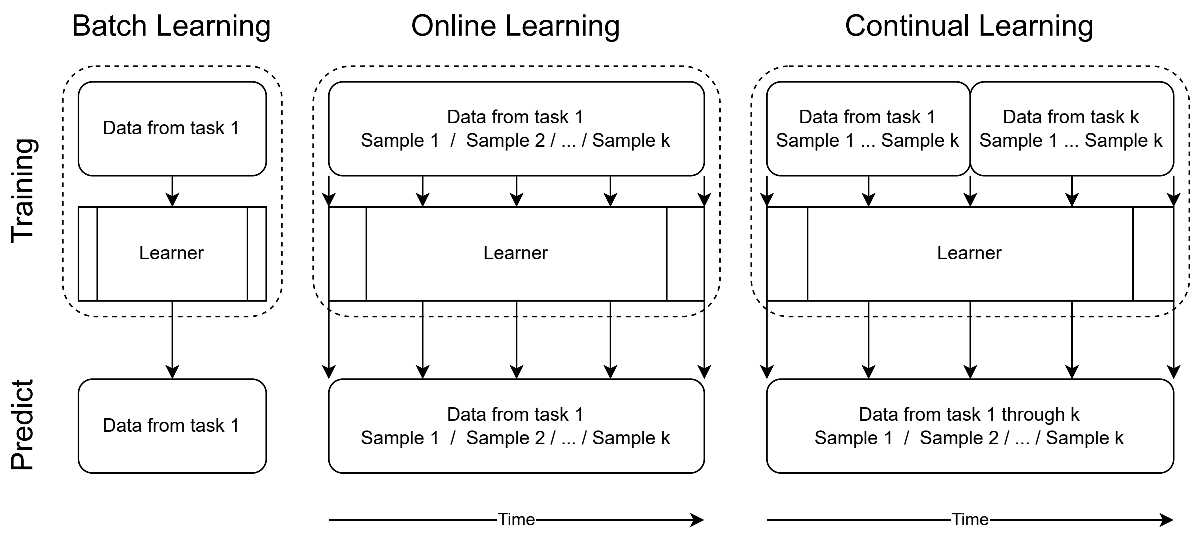


Figure 2.2: Comparison between the most relevant training settings [1].

the learning algorithm needs to find how to group the data according to its features. One common application of these models is also in recommender systems, in which we know what some users like, so we try to group them according to their likings to be able to recommend to them other shows or movies. Due to the focus of this thesis, we will not delve into exploring this type of method. An example can be seen in Figure 2.1(b).

2.1.3 Training Setting

Machine Learning models can be trained in various ways depending on when the data is available, and the type of data we want to make inferences about. We discuss some of the most relevant training settings for this work.

Batch Learning or offline learning is the way machine learning has been done traditionally. It works by giving all the available data to a learning algorithm that will start training over all that data, giving us a model of that data, so that we can, later, make inferences over that data.

The way batch learning algorithms usually deal with concept drift is by triggering retraining when it is found that a certain metric of performance is below a certain limit. These learning methods, however, are expensive to retrain [6], because they have to go through all the data again, which makes them not scalable for real-world applications where data is constantly arriving and the data distribution might change unpredictably.

Online Learning or Incremental learning is an alternative to batch learning in which we train the machine learning model with the data arriving sequentially. Using this method, we can also start making inferences before the training algorithm processes all the data, allowing it, for example, to process data as it is generated, such as new users arriving or in time series analysis. Online learning can also be a solution to problems where fitting all of the data in a single machine would be unfeasible, since, with this approach, it is only needed to look at one data point at a time.

Online Learning models are more robust to concept drift, because they are always training with the most recent data. This type of model may, however, suffer from the opposite problem, catastrophic forgetting. This means that, as more data arrives and the model learns the data distribution of this newer data, the model tends to ”forget” the data distribution observed in the older data and stops being as good at dealing with samples similar to that older data. Dealing with catastrophic forgetting is one of the bigger challenges in online learning research but although some studies have attempted to deal with this, they continue to lack ways of dealing with it in a general way [14].

Real-life recommender systems models, in particular, that are retrained more often have significantly higher quality than those that are retrained less often [15]. For this reason, together with the decreased training cost of not needing all of the data, it may be worth it to train this kind of system using online learning [6].

Continual Learning is a similar concept to online learning. This setting tries to reproduce the human brain that can learn different concepts during its lifetime. Here the model receives data from different tasks sequentially, so it has to learn each of them in an online fashion. This does not mean that in each of the tasks the model learns incrementally so we can have continual learning algorithms that are not online. Continual learning is even more susceptible to catastrophic forgetting because the data from the more recent tasks does not follow the distribution of the previous ones, but we want to still make inferences about those [1].

2.1.4 Distributed Learning

Machine Learning can be done in a distributed setting. This may be useful if the data is too big to fit in a single machine or if we want to reduce the training time. In this case, we have two types of Distributed Machine Learning. We can have Data or Model parallelism. In Data parallelism, the data is split across all the workers, and then each of the workers calculates an update to the model, that is replicated in every worker. The resulting model is the aggregation of all those updates. In Model parallelism, each worker calculates a different part of the model, and the resulting model is the combination of those parts. This type of parallelism is dependent on the type of model and learning algorithm that we are using so it may be harder to apply it in some situations [16]. Both types of parallelism rely heavily on communication between workers to share the parameters of the model that are being calculated by different ones, so the communication channel and pattern between them must be efficient.

One common architecture for achieving distributed learning is the parameter server. Here the model parameters are kept in a centralized server, the workers pull the model, calculate their update, and push

	On Premise	IaaS	PaaS	FaaS
Functions	Costumer	Costumer	Costumer	Costumer
Application	Costumer	Costumer	Costumer	Provider
Runtime	Costumer	Costumer	Provider	Provider
Operating System	Costumer	Costumer	Provider	Provider
Virtualisation	Costumer	Provider	Provider	Provider
Networking	Costumer	Provider	Provider	Provider
Storage	Costumer	Provider	Provider	Provider
Hardware	Costumer	Provider	Provider	Provider

Table 2.1: Manager of different components of the application in different cloud delivery models [3].

it back to the server that aggregates those values to find a new model [17].

Training ML models can be very resource-intensive and take a long time, for these reasons, it is advantageous to use cloud computing to train those models. Some machine Learning algorithms are able to take advantage of GPU or specialized hardware to accelerate the computation and some cloud offerings might make those available.

2.2 Cloud computing

Cloud computing makes computing resources such as servers available to users. This way different users can share the same resources without having to pay for them while not in use. There are several different cloud delivery models, that mainly differ in the amount of control and need for management by the user. See Table 2.1 for a comparison between them.

2.2.1 Infrastructure as a Service

is in one of those extremes. In Infrastructure as a Service (IaaS), the user has to manage all the infrastructure such as virtual machines, manage the operating system, install all the necessary software, manage the communication channels between workers and run the applications. This makes this model cumbersome to manage and with a high start-up time, but also very flexible, which means that it can easily integrate with existing applications. In addition to that, this is the cheapest delivery model when accounting for price per CPU time, making it an attractive choice for heavy workloads. However, since this requires extensive resource management by the user it often leads to resource under-provisioning or over-provisioning, leading to inefficient use of money.

2.2.2 Platform as a Service

Platform as a Service (PaaS) gives the customer a platform in which to extend with the application-specific code. Usually, PaaS offerings are geared towards a specific use case, offering the common features of that kind of use case. We have for example some PaaS offerings that are fit to Machine Learning use cases (Machine Learning as a Service) where some of the workloads of the ML pipeline are handled by the cloud provider.

In PaaS just like in FaaS, the Operating System and run-time are managed by the cloud provider, however, PaaS still leaves some control to the user. Here she can still choose how the application scales and has to manage when the code is run. Since PaaS is in between IaaS and FaaS it has some advantages and disadvantages from each.

2.2.3 Function as a Service

Function as a Service (FaaS) is another type of cloud delivery model. Here the user only has to write the application functions and the cloud provider manages all the infrastructure which makes this model the simplest to use. Since the cloud provider is managing the cloud resources, FaaS allows us to have high elasticity and a fine-grained cost model. Because the functions are so light-weight when compared with the VMs or containers that are used by the other models, cloud functions have low start-up times which makes them good to deal with unpredictable demand.

However, it also has some disadvantages. One of the disadvantages is that cloud functions are often limited in execution time by cloud providers. Another one is that the functions have some technical limitations such as access to specific hardware (like GPUs or TPUs for example) and also that they are not able to communicate efficiently between themselves, having to rely on an indirect form of communication such as a storage system, making the communication slower than ideal. Another possible problem is that the functions are slower in loading data and have more limited computation power. In spite of these limitations, FaaS is still very flexible since it allows the user to run almost any code, as long as it is between the bounds of this model.

Another characteristic of the FaaS functions is that they do not guarantee that the state is preserved between invocations, so, if the application needs to keep this state, it needs to be able to save it somewhere else so that it can be retrieved later [18].

Serverless is a term that can also be used to describe a cloud delivery model where the users do not need to worry about managing the servers. These two concepts are often used as synonyms since most of the time a service that is serverless is also FaaS and vice-versa, nonetheless, it is possible to have a system that is only one or the other. We could have, for example, a serverless offering based on containers instead of functions or a FaaS offering with more fine-grained management by the user.

Serverless is a misnomer since, in practice, there is still a server, it is just not managed by the user.

2.3 Cloud Machine Learning

Training large Machine Learning models is very resource intensive, and for that reason, people often resort to using cloud computing to train those models. Using cloud computing, users do not need to buy all of the hardware infrastructures that they use and only have to pay for the time that they are using it, making it more affordable. For the same reason, users can use better and faster hardware and have more instances of that hardware, which means that it is faster than training them on-premises. This also means that we are able to make more experimentation, with different models and parameters to find the best settings for our specific application. Depending on the cloud delivery model it may also be easier to maintain since the cloud provider may be handling the operating system or the application run-time for example. Usually, users use either IaaS or MLaaS approaches.

2.3.1 Infrastructure as a Service

IaaS gives the user a lot of flexibility since the users get access to the infrastructure very directly, allowing them to use all of its capabilities including direct communication and specialized hardware. It is also the least expensive offering when accounting per processor time. However, since the users are ML experts, managing cloud infrastructure is an obstacle for them. For this reason, IaaS comes with hidden costs. Managing infrastructure is time-consuming and error-prone, which leads to cost inefficiencies. One example of a service that uses this model is Amazon EC2 [19].

2.3.2 Machine Learning as a Service

MLaaS is a kind of PaaS that allows the user to build and run Machine Learning pipelines with little work. These are platforms that are made specifically to train and serve ML models so, they are a good option for people who do not want or need to manage the infrastructure, however, the underlying infrastructure is still based on VMs inheriting some of their characteristics, such as GPU support or high start-up time. This approach has a significantly higher cost per CPU time than IaaS. Also, because each platform has a different API and those are more complex than the ones in IaaS or FaaS, these platforms make the user more susceptible to vendor lock-in. One example of a service that uses this model is Amazon Sagemaker [20].

2.3.3 Serverless Machine Learning Training

The serverless model has some potential benefits to train machine learning models. Serverless has the potential to bring together the benefits of IaaS and MLaaS since it can make the management of resources way easier for the Data Scientists who want to train models, but also be cheaper and more flexible. It also makes scaling up or down the number of workers during the training job much simpler, achieving an increased capacity or reduced costs, as needed.

This model, however, also has some limitations that make it more expensive and slower to train batch-learning algorithms when compared with the other options that were talked about. This is mostly due to the fact that the functions cannot communicate directly with each other, creating a bottleneck there. Other factors slowing it down is that FaaS cannot currently access GPU and that the serverless functions are often capped in time and memory by the cloud provider, so the developer has to create some workarounds to deal with that such as relaunching the functions when they reach those limits. Because of those limitations, for batch learning, the serverless model is slower and more costly than IaaS in most cases. The exception is that, for smaller models with reduced communication, FaaS can be faster but then, it is never much cheaper than IaaS [2].

With that being said, the serverless model presents a promising option to models that require frequent and unpredictable updates, since, for this situation, it is probably too expensive and time-consuming to have to wait for a VM to boot whenever the need to train the model arises. One example of a service that is Serverless is AWS Lambda [21].

3

Related Work

Contents

This chapter, first, discusses some works in the recommender systems space (Section 3.1). Then it is discussed a few different approaches that, like ours, deal with serverless machine learning (Section 3.2).

3.1 ML-based Recommender Systems

This section discusses some works related to recommender systems using Machine Learning. It shows some of the unique challenges of this type of systems, highlighting the necessity of these systems to adapt to change and how there is a lack of research support to dynamic data.

3.1.1 Deep Learning Recommendation Model

Deep Learning Recommendation Model (DLRM) [22] is a Deep Learning Model that combines different ML techniques and intuitions to exploit categorical data, something that traditional Deep Learning Models are not very good at.

This model starts by using embeddings, that transform each categorical feature into a numerical representation and are later combined with the continuous features using many small neural networks. The model also allows for parallelism, using model parallelism for the embeddings and data parallelism for the neural networks. This architecture slightly outperformed another deep learning model, Deep and Cross Network [23], in terms of training and validation accuracy.

3.1.2 Online Learning for Recommendations at Grubhub

Grubhub [6] is an online food delivery and ordering platform that connects customers with restaurants in their local area and their revenue depends heavily on recommendations. For that reason, having a recommender system that can adapt to new preferences, events and new restaurants joining the platform is very important to them.

For that reason, they compared two different ways of training their recommender systems. The first one is using batch learning with frequent retraining on a sliding window of new data. This approach is simple and can adapt to new trends due to the model seeing the most recent data, however, it is expensive. The alternative is using online learning. In this case, instead of retraining, the previous model is trained on the new data saving the cost of seeing the older data.

After making those changes, they found that training with the newer data daily led to a 20.3% increase in Purchase Through Rate, due to the faster adaptation to concept drift and a 45$\times$ cost decrease due to the decreased cloud usage of the batch retraining.

3.1.3 Modyn

There is a gap between applied ML and ML research [24]. While ML in the real world often deals with dynamic datasets, where the data grows and changes, most research focuses on static datasets.

To address this problem, Modyn was proposed as a platform that tries to bridge this gap, by allowing researchers to evaluate dynamic datasets. Modyn trains ML models on dynamic datasets, selecting *when* the training should occur and *what data* should be used for training, selected by the researcher. The triggering of the training can depend on the time passed after the previous trigger, the number of samples received in the meantime or the drift in the data distribution. The data selected to train can be either all of the available data, a random subsample, or a subsample prioritizing newer data.

3.2 Serverless Machine Learning

We will also discuss some works that have considered serverless for machine learning. We will see the different challenges that arise from training and inference and that complex obstacles arise from training

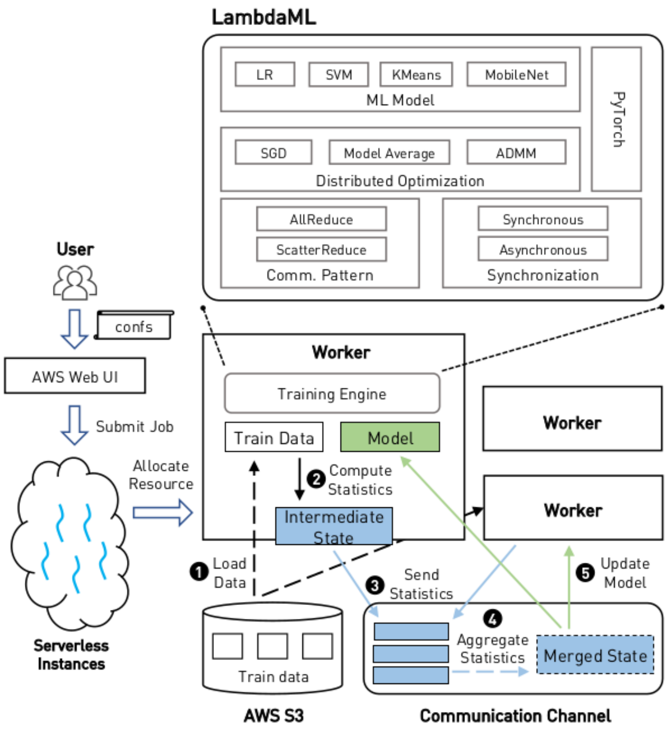


Figure 3.1: LambdaML's architecture [2].

using serverless due to the difficult communication between functions.

3.2.1 LambdaML

LambdaML [2] is a pure FaaS platform that was built to allow comparison between FaaS and IaaS systems to train ML models using distributed learning. This system allows for comparison using different Distributed Optimization Algorithms, Communication Channels, Communication Patterns and Synchronization Protocols. LambdaML was evaluated using several types of ML models including linear models, neural networks and a clustering model for unsupervised learning problems.

For the user to start a job, she configures the configurations on the AWS Web UI and it starts the LambdaML workers. Each worker starts by loading its portion of the data, and then it starts training the model on that data, computing an update for the model. The worker then writes the update to the communication channel that aggregates the updates and sends the new model to the other workers. This system was built using AWS Lambda [21] functions and the data is loaded from AWS S3 [25]. The experiments were done using several different communication channels including S3, Memcached [26], Redis [27], and DynamoDB [28]. The architecture of this system is representative of most of the works in serverless Machine Learning and it can be seen in Figure 3.1.

Training ML models using LambdaML was found to be generally slower and more expensive than with the existing IaaS approaches. The exception is for models that have reduced communication and converge quickly. Even in this case, it is faster, but not much cheaper, to train using LambdaML.

LambdaML does not take full advantage of the FaaS model because it does not vary the number of workers during a machine learning job. It also is focused solely on Offline Learning.

3.2.2 Cirrus

Cirrus [29] is a framework that uses serverless and automates the ML Workflow end-to-end, meaning that it not only performs model training but also hyper-parameter optimization, data loading, and preprocessing. The system however is not implemented fully using FaaS. It uses a parameter server to share the updates to the model between workers. This way the system can address one of the serverless weaknesses, the slow communication between workers.

Cirrus has a parameter server architecture for data parallelism. Here, each worker loads its part of the data, then pulls the model from the server and starts calculating its updates. The server saves and aggregates the updates. In the case of Cirrus, the workers are serverless functions while the parameter server is a distributed data store that runs on cloud VMs.

Using this approach, the authors found that it can be 100$\times$ faster than PyWren [30], a system that allows a user to run any Python code in AWS Lambda [21]. However, when compared to LambdaML [2] it is never the fastest option, because, in the scenarios where FaaS shines, it still has to allocate a VM, getting the worst of both worlds.

3.2.3 MLLess

MLLess [31] presents a system to train ML models on top of IBM Cloud Functions [32]. This system uses a decentralized architecture in which the workers communicate using two different channels for different purposes. For sharing model parameters they use Redis, but to exchange control messages between them they use a messaging service. In addition to the workers, there is also a supervisor, which is also a serverless function, that has the tasks of aggregating statistics, synchronizing worker progress, and removing unnecessary workers as they are no longer needed. MLLess applies some optimizations that minimize the communication between workers, since, as we saw, this is the main bottleneck of the FaaS model.

It uses a significance filter to filter out the updates that least contribute to the model. This way workers can avoid exchanging these updates, saving on communication time.

The other optimization used by this system is the scale-in scheduler. Because serverless functions are very easy to scale up or down the number of workers, it is possible to change the number of workers during the job to maintain cost efficiency. Since the ML training typically makes the most progress in the beginning, it is desirable to have lots of workers during that time. However, as progress slows down, the scheduler reduces the number of workers to reduce the cost.

They find that it is possible to have better performance than both a non-specialized system and serverful systems for models that converge quickly, especially if the model is sparse, being up to 15× faster and 6.3× cheaper than PyTorch in that case.

3.2.4 Hydrozoa

Hydrozoa [33] is another system that trains machine learning models, this time focused on Deep Neural Networks. They achieve this by using a different approach, Hydrozoa is based on Serverless Containers instead of Function-as-a-Service. This allows it to address some of the more significant shortcomings of the serverless model that come from using cloud functions. With this approach, the system can maintain the advantages of the serverless paradigm adding the ability to use GPU computing and direct communication. Direct communication is achieved by using ZeroMQ [34], a system that allows the exchange of messages using sockets. The system also exploits the fact that serverless allows for high elasticity to allow for dynamic worker scaling, meaning that throughout the ML job it can vary the number of workers to reduce training costs.

The system implementation is divided into the worker tasks, a planner that decides how to partition the model and the data so that it can perform both data and model parallelism using the best possible setup, and a coordinator that orchestrates the training job. All of these parts run on Azure Container Instances [35]. Before the training, the data is loaded from S3 and the results are stored there at the end.

The authors compare Hydrozoa with a FaaS-based system similar to LambdaML [2] and to two different MLaaS approaches and find that Hydrozoa using GPUs has, using one of the ML models, 155.5× higher throughput per dollar than the FaaS approach. Even the CPU-based version has 11.5× higher throughput per dollar than the FaaS approach. When compared to MLaaS it also gets some improvement, achieving up to 5.43× higher throughput per dollar, mainly due to the cost reduction.

Since Hydrozoa is based on containers instead of functions, it may suffer from less fine-grained scaling, a higher level of management needed by the user and a higher start-up time.

3.2.5 Serverless Model Serving

Serverless Model Serving [36] compares the use of the serverless model against IaaS and MLaaS for model serving. For a data scientist to deploy a model he/she uses a function that deploys a model to the cloud to be later used. When the user asks for an inference, the function downloads the dependencies and the model, if it is not warmed yet, calculates and returns the result.

FaaS systems suffer from the cold start problem, so the first few requests are slower in this model. As the instances warm up, the baseline latency is similar throughout all systems but when it is needed to scale up the IaaS and MLaaS systems take too much time to do that, which causes delays. Because of those delays, the study finds that serverless is the option that offers better average latency. The GPU-based system was the only one that was able to compete in this metric because it started to struggle later than the other systems, requiring less scaling up.

The study also compares the two dominant FaaS offerings, AWS Lambda [21] and Google Cloud Functions [37] and it finds that the performance of Lambda is generally the better between the two.

Even though this work is not focused on the training phase of the ML pipeline, it is still useful to compare it to our work since it may make sense to compare the end-to-end time to serve a model since the training data was generated.

3.2.6 Summary

All the analyzed works take advantage of different techniques and approaches to try and make machine learning training easier, cheaper and faster to perform. The main point common from these studies is that some specialization is needed for ML training in FaaS to achieve good performance because the more generalized approaches are usually more expensive. However, all of them focus on an offline model, where we just train the model once, so they all use batch learning. Because of that, these systems fail to adapt the models for real-world applications, where data distributions change quickly and unpredictably. The goal of this work is to address that, making a system that can train online, taking full advantage of the scalability of the FaaS model.

4

Serverless Online Machine Learning

Contents

This chapter discusses the architecture of the solution and how it is used to train ML models online using FaaS. First, it shows an overview of the solution (Section 4.1), then, it discusses how it trains models online (Section 4.2). Finally, it shows some complementary optimizations to the system (Sections 4.3 to 4.5).

4.1 Overview

The system trains models online using FaaS. As shown in Figure 4.1, it is comprised of a FaaS training worker, and a controller, that manages the data stream, storage for the model and storage for the data.

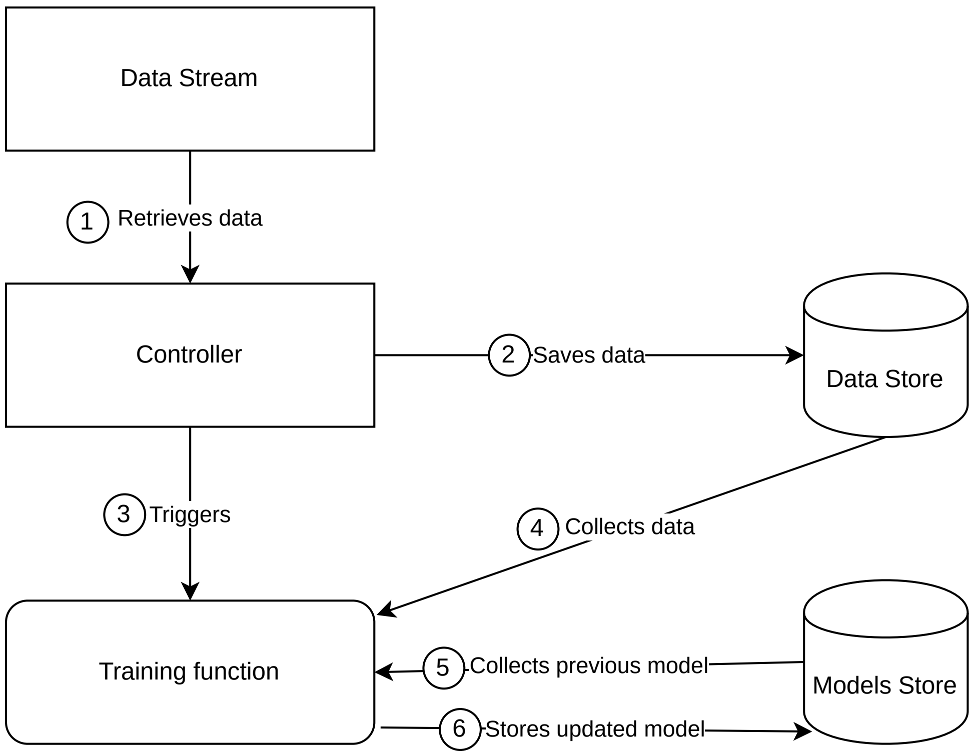


Figure 4.1: System Architecture.

The controller, (1) collects data from the data stream and (2) saves the data in the data store, where it can be read later. Then, when there are enough data samples available to trigger training, (3) it invokes the function, telling it where the data is, as well as other training configurations.

The training worker, that is a FaaS function when invoked by the controller, starts by (4) getting the data and (5) model from storage. Then, (6) it trains the model on the data that just arrived and saves the model so that it can be trained with more data by other functions, or used to make inferences.

Since each invocation of the function is short, and there is always a recent version of the model available in the model storage, at any moment, that model that is in storage, can be used to make inferences.

4.2 Serverless Online Learning

The system trains ML models that can learn online, that is, they train incrementally, as they receive data, using serverless computing. This type of model can adapt to data that evolves, so the model has to be updated quickly with the most recent data, as soon as it arrives, so that there can be made inferences based on the most recent data. With this approach, the system can take full advantage of the serverless paradigm, by only allocating computing resources as more data is generated to be fed into the model.

With this in mind, the training function can start executing as soon as the controller deems there is enough data to go through a training iteration. At that moment, the function starts and downloads the data that will be used to update the model. Then, it checks if the most recent version of the model is already in memory from a previous invocation, and, if it is not, the function loads it from storage.

After this setup, the function gathers performance metrics from the model on this new data and the online Machine Learning code that will update the model is executed, generating a new version of the model.

When it finishes, this new version is uploaded to storage so that it can be used to make new inferences or to serve as a base for the next training iteration. Finally, the function returns to the controller the model performance metrics.

To avoid conflicting versions of the model, the controller only invokes a training function after the previous one returns, meaning there will be a maximum of only one training worker running at a time. Removing this limitation could be a source of future work that would further increase the system's scalability.

As we have seen in previous chapters, one of the most important challenges in serverless Machine Learning is the slow communication between FaaS functions and storage. Because of that, the next sections propose some optimizations to the system that attempt to reduce this communication time or its impact.

4.3 User-specific adaptation

The system supports, optionally, user-specific adaptation, which means it can store a separate model for each user and personalise it better.

When this method is activated by the controller, it sends only data related to this user to the function. If this is a new user, the function will then start training from an existing pre-trained model, that was trained using data from all users. After training on this data, the resulting model is saved separately, so that it can serve inferences requested by this user and continue training from more data generated by her.

This has two potential benefits. The first one is that the model can be trained to this user's specific tastes, therefore it may perform better than a model that is trained in data from every user. The other one is that this allows for trivial parallelism, since the users do not share models, the system could be training two different models at the same time without any concurrence issue.

Using this optimization, the model can be updated in a decentralized way, allowing multiple functions to update the model at the same time. Then, at some point, the global model, that is used by everyone, is retrained using all the new data. This allows all the other users' models to also improve with this data.

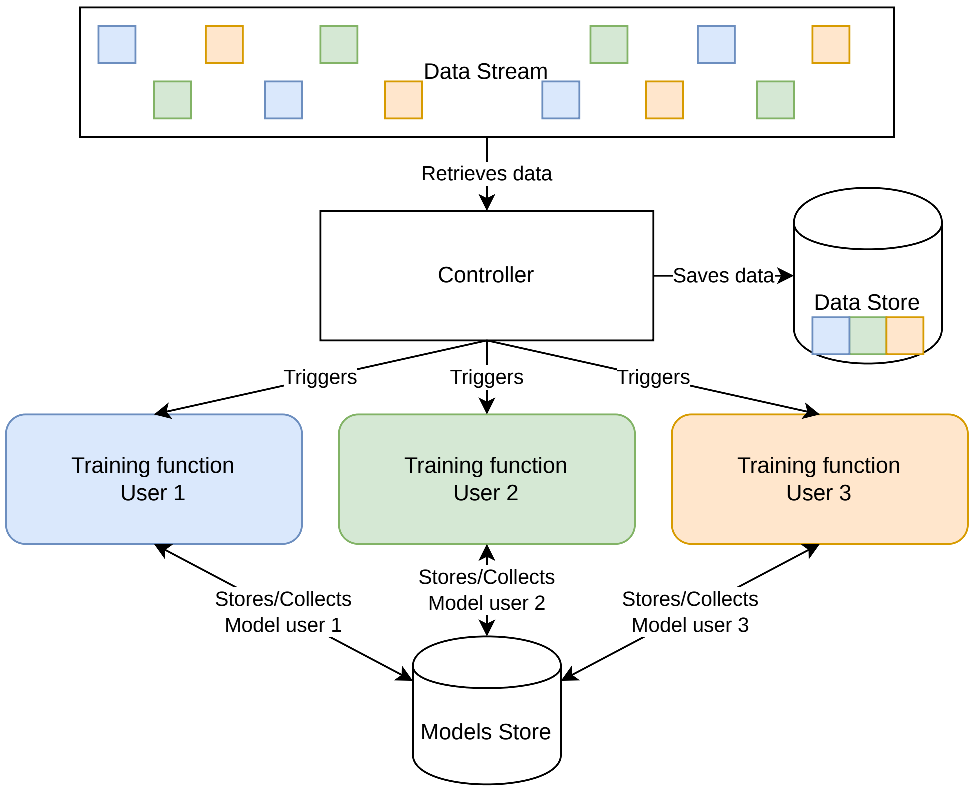


Figure 4.2: System architecture with User-specific adaptation.

The architecture of the system when using this optimization can be seen in Figure 4.2.

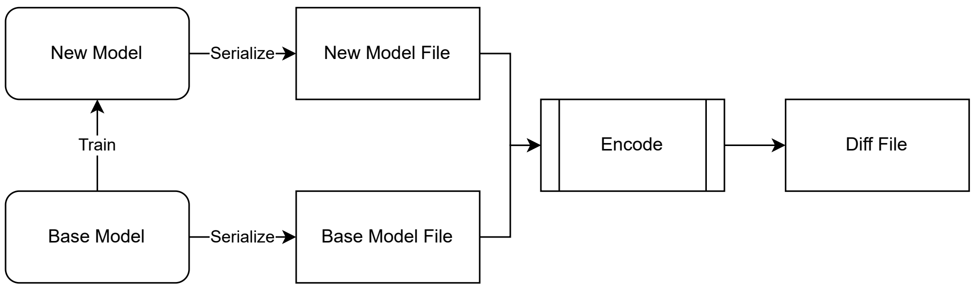
4.4 Delta encoding

Another optional optimization that the system supports is delta encoding of the model file, which means saving just a delta difference file, or diff file that, when paired with the pre-trained model that was the base for this version of the model, may be utilized to generate back the complete new model [38].

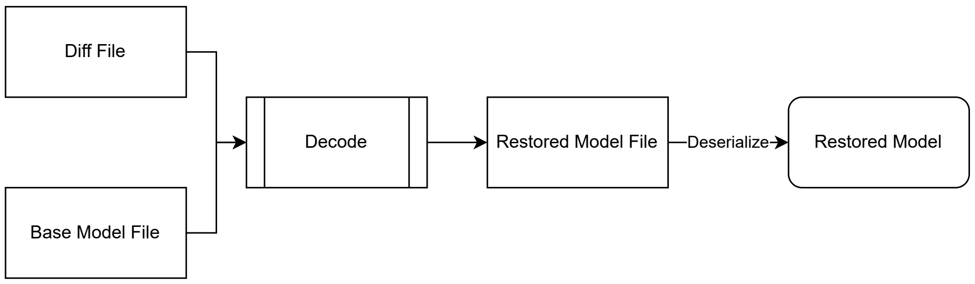
For this feature to be activated by the controller, the pre-trained model must be already in the container where the function is running. That can be done when registering the function [39].

Then, when this feature is activated by the controller, instead of completely downloading the model, only the diff file is. This file, together with the pre-trained model, reconstructs the up-to-date model, as seen in Figure 4.3(a). After training, the reverse process happens, the delta is computed from the pre-trained model and the up-to-date model and it is sent to storage, as seen in Figure 4.3(b).

The process of generating the diff file can be time and memory-intensive and may not even result in a file significantly smaller. However, if both files are similar enough, as are similar models, the generated delta may be orders of magnitude smaller than both input files and the time savings of sending over the network might save some latency.



(a) Encoding Model.



(b) Decoding Model.

Figure 4.3: Delta encoding.

4.5 Cached state

As previously discussed, FaaS functions do not guarantee that the state is kept between invocations. For this reason, the functions need to save the updated model to some kind of persistent storage to use it to make invocations or to continue training. Sending the model over the network to save it and later retrieving it from there is slow when compared to the short-lived duration of each function invocation. Because of this, the model should be kept only in the function's memory whenever possible.

To achieve this, when the function starts, it checks if the state was kept and there is the most recent version of the model is already in memory and ready to be used. If that's not the case, it downloads it from the model store.

Then, after training the model, there are two options, as defined by the controller. Either the model is saved to storage to be used in another invocation, or it is saved only the training history in storage. The training history is a file containing the location of the data that was used to train the model locally but that is not reflected yet in the model that is in remote storage. In this second case, saving only the files' location is faster than saving the full model. To guarantee that the model has always been trained with all the available data, if the state of the function is lost between iterations, the model that is in storage must be trained again using all of the data in the training history. This scenario however is unlikely, since, more often than not, in practice, the state is kept between invocations of the same function. Even if that is not the case, it can still be the cheaper and faster scenario since retraining may be faster than uploading the full model to storage due to its relatively large size.

In order to avoid a scenario where the retraining is needed, the model that is saved in the function could be used and frequently updated. This way the retraining in case of lost of the cache would not be so impactful. To compare this optimization with the original solution, we can look at Figure 4.1 where steps $\textcircled{5}$ and $\textcircled{6}$ are avoided by instead keeping the model always in the Training function.

5 Implementation

Contents

This chapter discusses the implementation of the of the proposed system. After a brief overview (Section 5.1), the implementation of each component is described in a specific section (Sections 5.2 to 5.6).

5.1 Overview

Based on the architecture purposed in chapter 4, it was implemented a solution based on two components, a FaaS function that runs on AWS Lambda [21] and a simple controller that runs locally. We will get into more detail in the following sections.

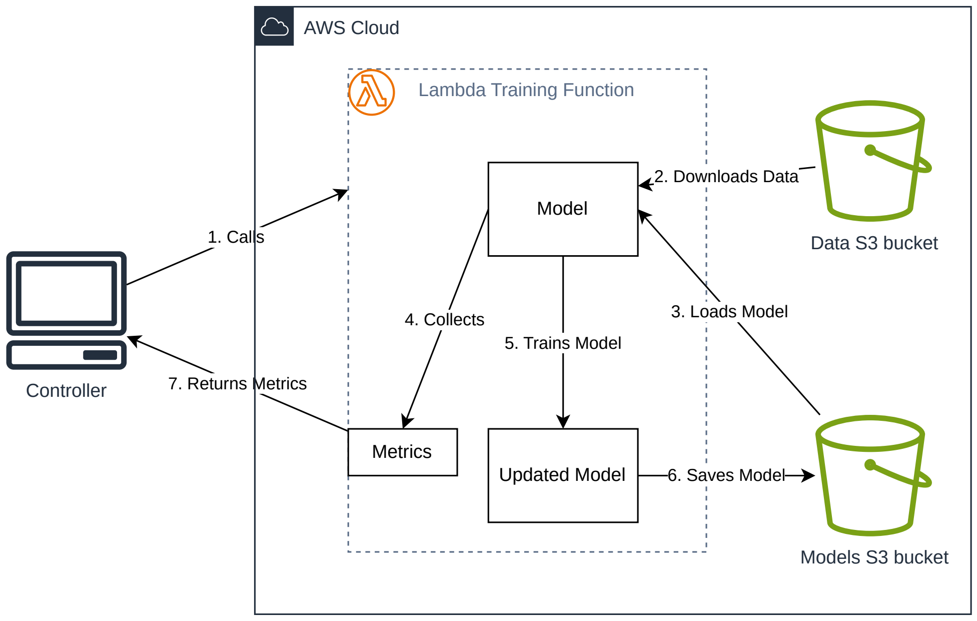


Figure 5.1: Implementation Architecture.

5.2 Deployment

The function was deployed in Amazon Web Services (AWS) Lambda using a Docker [40] container image. For this deployment, an image was created locally using an AWS base image for custom runtimes. This approach was selected as an alternative to the default zip upload method, which has a limit of 50MB, to accommodate the larger size of the dependencies, that zipped go over 90MB of size. By packaging the function and its dependencies as a container image, the limitations associated with the default deployment method were circumvented, since in this case the limit is 10GB for the whole image and the total size of our image is less than 1GB [41]. All of the necessary software and the base model used for delta encoding was added to the image and it was uploaded to Amazon ECR [42]. This service manages container images and is integrated with other AWS services, such as Lambda. From there, the function can be deployed using the Lambda command line interface.

5.3 Function Implementation

The function is implemented using Python. When the function starts, the arguments that were received from the controller are in the event object and are retrieved from there as needed. The function loads the model, metric and data from storage according to the instructions from the controller and the model and metric objects are deserialized using Pickle [43]. Pickle can serialize and deserialize Python objects, such as machine learning models to a binary format. Then, the model makes inferences on the received data to collect performance metrics from the model and, after that, the model is trained. After training the model, it is saved to storage along with the metric object. After that, the metrics are returned to the controller.

5.3.1 River

All the ML code used in the system, including training, inference and metrics, uses River [44], a Python library for online machine-learning models. River focuses on learning from data streams, without looking at the whole data, for this reason, every river model can learn from just one sample at a time. The Listing 5.1 shows an example of River code where it is possible to see the simplicity of training a model online using the library. Each call to `learn_one` trains the model with one sample. In addition to training models, River also offers some features that make online learning easier. For example, most models support emerging and disappearing features, for cases where the data structure changes. Other river capabilities for online machine learning are online model evaluation and data processing. River was chosen because it focuses in online learning, however the proposed system and its optimizations are agnostic to the ML framework and could work with other models that train models incrementally.

River has some compatibility with a well-known machine learning library, scikit-learn [45], implementing methods that allow scikit-learn models and datasets in river and vice-versa. Scikit-learn, as most Machine Learning and Deep Learning libraries, is optimized for batch learning, where all the data is available from the beginning of training. Scikit-learn also has online machine learning capabilities using the `partial_fit` method, implemented in some models but, since the library is not focused on online learning, that method is not implemented in most models. Some river models support mini batching, using `learn_many` method, similarly to the `partial_fit` method in scikit-learn. In this case, the model is trained with more than one sample at a time and the training function is optimized for that.

River has support for metrics for evaluating the models that are also computed incrementally. This metrics are objects that are updated online with the real and predicted values of the labels using the method `update` in River. That object is saved alongside with the model between calls to the function. Listing 5.1 also shows an example of using metrics in River.

Listing 5.1: Example of usage of the river library.

```python
# Iterate over the data stream
for xi, yi in data_stream:
    # Evaluate the current model with the new sample
    yi_pred = model.predict_proba_one(xi_scaled)
    metric.update(y, y_pred)

    # Train the model with the new sample
    model.learn_one(xi, yi)
```

Due to the excessive overhead of initializing the function, loading the model and saving it, it is unfeasible to train purely online using this system, calling the function for each data point. For that reason, the controller has to hold some samples until it makes sense to call the function. When possible, the function uses the `learn_many` method, but with many models, it is still needed to use `learn_one`. In this case, the model is still trained one sample at a time, even if more data is already available, but all this training still happens in the same function call. Some river utilities are not available at all for mini-batching, one example of that is metrics, so if we need to evaluate the model as it trains in mini-batches, we still need to gather one inference at a time to update the metrics.

5.3.2 Delta encoding

When the delta encoding optimization is activated by the controller, it also has to select one of the delta encoding tools that the system supports, Xdelta3 [46] or BSDiff 4 [47]. If that is the case when the function has to save the model after training, the model is serialized, and then, the chosen algorithm is applied to the result together with the base model that is saved locally in the functional and the diff file is uploaded to storage. Then, in the next iteration, when the function retrieves the model, it only has to download the delta and apply the steps in reverse. These two tools, Xdelta3 and BSDiff 4, were chosen because they are open-source implementations and they are widely used, however, they are not the same.

The delta encoding algorithm used by Xdelta3 is designed to be fast and to generate small deltas. Xdelta3 uses VCDIFF [48], a standard for encoding deltas, so it is compatible with other tools that use the same standard. BSDiff was designed specifically to distribute updates to executable files, so it uses a different algorithm that usually finds smaller diff files when encoding binary files. It's algorithm however demands lots of resources when making these encodings [38].

5.3.3 Cached state and retraining

After training one iteration, the model is always saved in a global variable so that it is kept in memory in the function. This is because, in the likely event that the function is reused by the cloud provider, the model does not have to load the model from storage.

In addition to that, the controller can decide whether or not the function should save the model resulting from training to remote storage. If it is chosen to not save it, when the function finishes training, it uploads the training history to remote storage. Otherwise, the full model is saved and the file with the data locations is deleted. This policy can be either always, never, or every chosen number of iterations so that there can be some time improvement while also avoiding losing lots of updates to the model.

If, in the new iteration, the model is not already in memory, it is because it was lost. In that case, the function loads the training history, and before proceeding with training in the new data, it will retrain the model on that older data.

It is also kept at all times a version number of the model to guarantee that the most recent model is the one being trained and that all of the data is incorporated into the model.

5.4 Controller

The implemented controller is a simple Python script that calls the training function periodically. Since this is not a real system where the data is being generated as the model is trained, the implemented controller doesn't manage the data stream, as it does in the idealized system for real-world use. Instead, the training data is already stored, and the controller, when it triggers the function, just has to tell the function where the data is.

The controller receives as arguments the parameters that are passed to the function and also the interval it should wait between triggers to the function. The arguments that are sent to the function are sent as a JSON object in the payload of the invocation of the function.

5.5 Storage

The remote storage used by the system is Simple Storage Service (S3) [25]. S3 is a storage service provided by AWS. This service allows anyone to store any kind of data and scales virtually indefinitely. The data stored in S3 can be accessed, if allowed by the bucket's permissions, from anywhere on the internet, and easily by other AWS services. This set of characteristics makes S3 the preferred choice for a serverless architecture also using AWS. The data in this service is organized into buckets, and inside each bucket, each piece of data has a unique identifier called a key.

The data that is used to train the models is split into batches and each batch is saved in separate a file that was previously uploaded to a S3 Bucket. This way, when the function is called, the function knows where to download the data from. The S3 object key where the data is, is given by the controller in the function call. The models, the metrics, the training history (for when using cached state) and the diff files (for when using delta encoding) are stored in a separate S3 bucket. This allows for better control over the permissions of each of the two buckets.

When using delta encoding, the pre-trained version of the model, which is used as the base for the resulting models, is stored locally in the function. For this, the model is moved to the container where the function is before registering it. This way, this model never needs to be downloaded from any remote storage.

5.6 User-specific adaptation

When using user-specific adaptation with the implemented system, the controller is instructed to feed the samples only from one of the users and the function trains on that data and saves the resulting model separately.

6 Evaluation

Contents

This chapter starts by discussing what it is expected to get from the evaluation (Section 6.1), and then, the setup of the experiments is explained (Section 6.2). After that, the actual experiments are discussed, starting with the comparison between the different deployments (Section 6.3), then the evaluation of the possible optimizations (Sections 6.4 and 6.5) and an experience comparing different ML models (Section 6.6). The chapter ends with a discussion of all the experiments (Section 6.7).

6.1 Evaluation Goals

With this evaluation, we are trying to study the viability of Serverless Machine Learning, more specifically, online Machine Learning, by running experiments with the developed system. The evaluation will attempt to answer the following questions:

- Is it viable to train online ML models using FaaS when compared with IaaS? This question is addressed in Section 6.3, where deployments in those two cloud delivery models are compared;
- How do the proposed optimizations to the system affect that viability? This question is addressed in Sections 6.4 and 6.5, where different scenarios, with and without the optimizations, are compared;
- How well does the system handle different ML models? This question is addressed in Section 6.6, where the system is used with different models, in order to compare them.

For that reason, it was performed some evaluations where different versions of the system were compared.

6.2 Experimental setup

6.2.1 Deployment

As described in the previous chapter, the system was implemented using Python 3.11 and deployed in an AWS Lambda function running with 2GB of memory and 512MB of ephemeral storage. The function and the S3 buckets used for storage were deployed in the `us-east-2` AWS region.

The controller, running locally, sends the requests and saves the timings and performance metrics so that they can be evaluated after the experiments.

6.2.2 Metrics

For each experiment, it was measured a metric of the performance of the model to compare to the benchmark and guarantee that the model does not suffer from the change in the system that is being evaluated. When the function receives the data, it first updates this metric with all the new data, and only after that starts training. This metric is the Root Mean Square Error (RMSE) when the model in evaluation is a regression model and the accuracy when evaluating a classification model. The metric is compared to the one given by a local script, performing training on the same data and it is expected that the performance of the model is the same across deployments.

It was also measured, for each invocation, both the server side and client side (the controller) latency, this way we could measure the platform overhead, that is the time that is not measured by the function,

userId	movieId	rating	timestamp	title	genres
1	296	5.0	1147880044	Pulp Fiction (1994)	Comedy|Crime|Drama|Thriller

Table 6.1: MovieLens sample example with rating and genres.

for example, the warm and cold starts. The function returns, in addition to the performance of the model, the time it took to run each section of the invocation, separated by setup, inferences, training and cleanup. The setup includes downloading the data and preparing the model and the metric according to the chosen parameter. Inferences are the time the system takes to evaluate the current version of the model on the new data, that is, making inferences and updating the metric. Training is, as the name says, training the model on the data that just arrived. Cleanup is saving the model and metric so that those can be used later.

6.2.3 Model and Dataset

The model chosen to experiment with the system was the `BiasedMF` from River [44]. This model is an implementation of the probabilistic matrix factorization [12] that also has user and item biases.

The model was chosen since it is made for recommender systems, and, as we've seen, it is one of the practical applications of machine learning that has shown better results by using online learning techniques.

The dataset used for most of the experiments was the MovieLens 25M Dataset [49]. This dataset includes about 25 million movie ratings given by different users, as well as the genre and tags of the movies so it can be used to experiment with recommender systems. An example of a sample from this dataset with the rating given by the user can be seen in Table 6.1.

The movie genres and tags were discarded to make the data simpler, as the performance of the model is not part of the evaluation goals. So the remaining data is an identifier of the user that made a rating and of the movie that was rated, the score and the timestamp of this rating. To simulate the online training setting, the remaining dataset was ordered by timestamp, then the model was trained with the first 80% of the data (training dataset) and the resulting model (the base model) served as the starting point of the experiments. The remaining data (testing dataset) was used in the actual experiments.

6.2.4 Other training details

Unless noted, the batch size per iteration was chosen as 10000, this way, the training part is a significant part of the function invocation but it is still fast enough to get training results very fast. Each experiment lasted for 30 iterations, after that, the experiment was stopped since more iterations did not yield different results. After the previous iteration finishes, the controller makes the new batch of data available immediately for training.

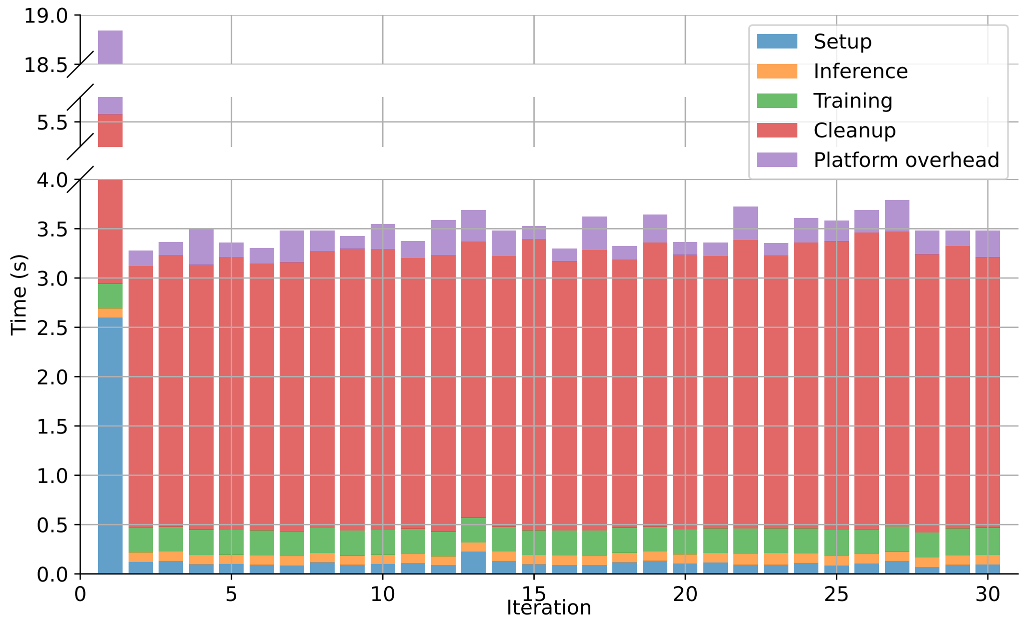


Figure 6.1: Baseline experiment, optimistic scenario, timing split.

6.2.5 Baseline

The baseline for the other experiments is the version of the system deployed in FaaS without optimizations. This means that after each iteration the function saves the whole model to S3, but, in the next iteration it tries to retrieve the model from memory before going to remote storage.

As we can see in Figure 6.1, most of the time spent by the function is spent in the later stage of the call, the cleanup. This is due to the time it takes to upload the model to S3. In this scenario, most of the iterations do not require the model to be downloaded to start the training, which is why the setup time is very small in comparison to the first iteration, where there's no model yet in the function's memory, so it needs to be downloaded. In the first iteration of each of the FaaS experiments, a significant portion of time is spent as platform overhead. This is because, in this iteration, the function is being cold started, so AWS needs to allocate the resources necessary to run the code.

To simulate a situation where a different function container is used and the model is not in memory, an experiment was also conducted where the function does not try to use that model, always downloading from storage. The results of that experiment can be seen in Figure 6.2. There we can see that, other than the increased setup time, the results are not significantly different. We call this one, the pessimistic scenario, while the previous one, where the model is already in memory most of the time, is the optimistic scenario.

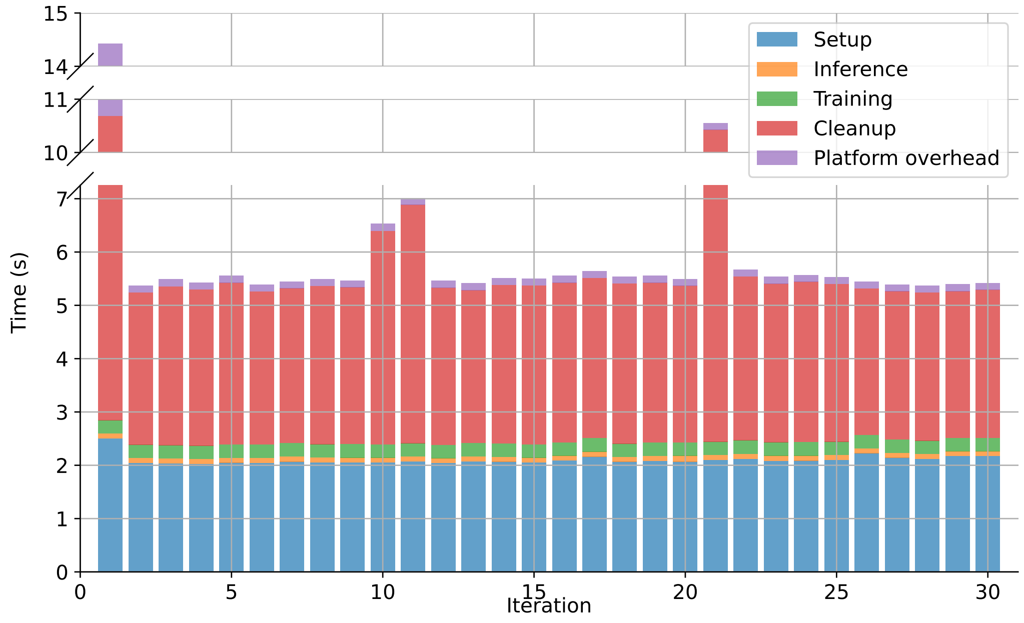


Figure 6.2: Baseline experiment, pessimistic scenario, timing split.

6.3 Comparison between deployments

The goal of this experiment is to evaluate how the proposed system, deployed in FaaS, compares with a similar system deployed in IaaS. For this reason, it was developed a similar version of the system, made for this type of deployment. Then, time and performance metrics were collected from both systems in order to compare them and understand when it is better to use one over the other.

The IaaS version uses a web server using Flask [50], running in an EC2 [19] t2.small instance. This type of instance has the same amount of available memory as the deployment in FaaS has (2GB). Flask enables the programmer to map URLs to Python functions. When a specific URL is requested, the corresponding function is executed. Thus, we set up a specific URL to trigger the execution of similar code as the training function, updating the model. This version is also activated by the same controller, which is just configured to make calls to this URL instead of invoking the function. Even though this deployment could potentially only save the model in local storage to guarantee that the model could be updated later, it was decided that it should, as the FaaS deployment does, save the model to S3 between iterations. This way this model could be used by another VM to make inferences, for example.

According to the policy set by the controller, both the FaaS and IaaS versions of the system also has an option to only make inferences on the data, this way we can see how a system that has no online learning would perform in a similar situation. When this option is set, the system does not save the

Training Setting	RMSE
Online Learning	1.0693
Batch Learning	1.0935

Table 6.2: ML metrics by the end of the experiments.

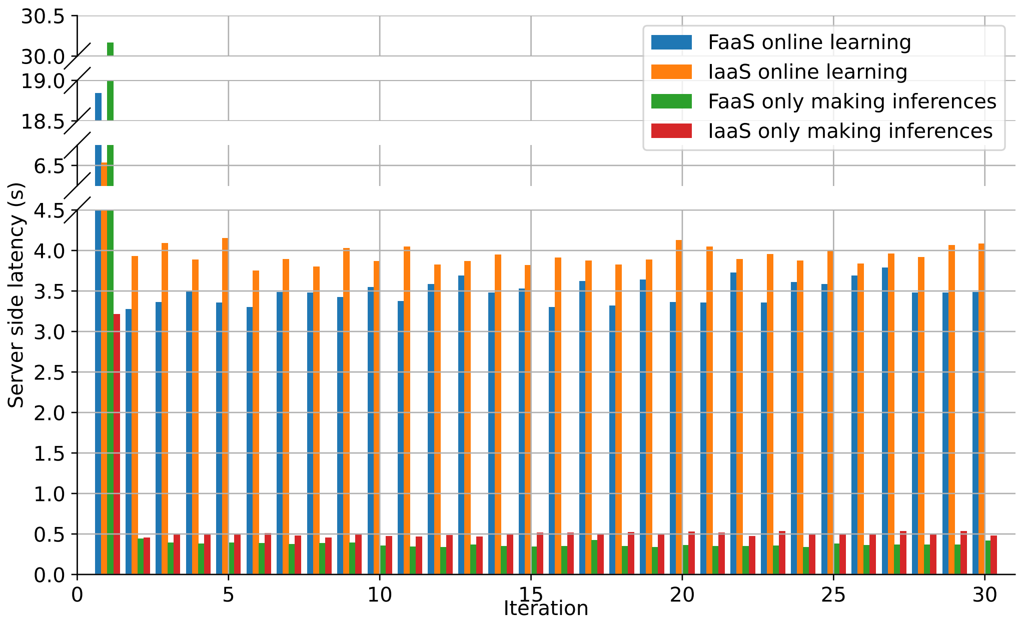


Figure 6.3: Comparison between deployments.

model to S3 since it does not make any changes to it.

The baseline version of the system was compared with the FaaS version making only inferences and both options deployed in IaaS, the one performing online learning, and the one that just makes inferences from the pre-trained model. In both experiments with IaaS deployments, when the experiment starts, the VM has started running.

The time each system took in each iteration can be seen in Figure 6.3. The faster versions of the system are, as expected, the ones that do not perform any online learning. The version running in FaaS is slightly faster than the one performing online learning in IaaS in most cases. The exception is in the first iteration, where the cold start in the FaaS deployment makes it much slower in this iteration and since the VMs are already running this is not a factor in the IaaS deployments.

As for the performance of the models, when training online the RMSE was slightly lower than of the model trained in batch, as can be seen in Table 6.2.

Another important factor in cloud deployments is its cost. For that reason, the approximate cost of each deployment was compared in Figure 6.4. The figure compares the cost of the IaaS deployment

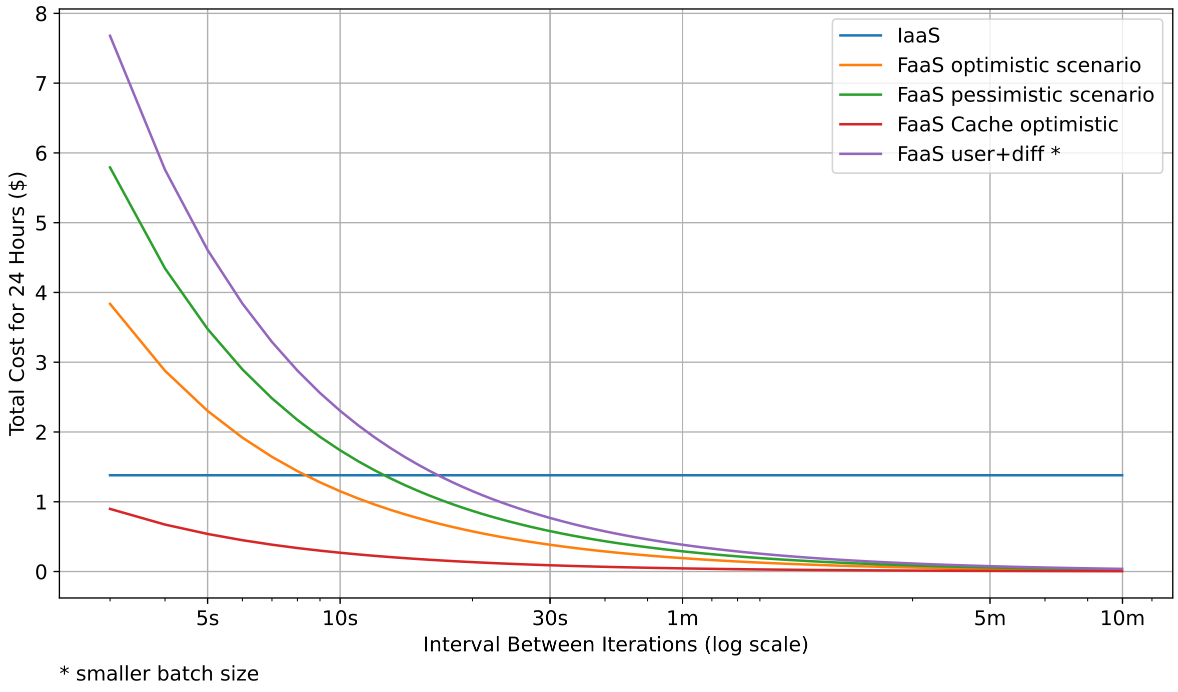


Figure 6.4: Cost comparison between training deployments.

against the some scenarios of FaaS deployments. The two scenarios of the baseline were used since, for smaller average intervals between calls to the function, the model will be most times in memory so the cost will look closer optimistic scenario. But as the time between calls increases, the function cache is probably deleted, looking closer to the pessimistic scenario. The calculation shows that, when the interval between iterations is small, the IaaS deployment has an advantage. But as it increases, the cost of the other deployment decreases, making it more viable. The exact turning point changes between models and the size of the batches used, but in this experiment, at around 12.6 seconds between invocations for the pessimistic scenario, turns it into the cheapest deployment. It is important to note that, for the same price, the IaaS version has a bigger capacity of consuming data, so if we have more data than in these experiments, the FaaS version might not be feasible.

6.4 User-specific adaptation and Delta encoding

The goal of this experiment is to understand how the user-specific adaptation and delta encoding optimizations affect the performance of the model and the training speed. In order to understand how these optimizations impact the viability of the system when deploying in FaaS, the system was evaluated with the user-specific adaptation and delta encoding optimizations enabled. The experiment was timed for the versions with and without the optimizations and those were compared to understand its impact.

File	Size
Pickled base model	142MB
Pickled model after training	143MB
Xdelta3 diff	3.8MB

Table 6.3: File sizes.

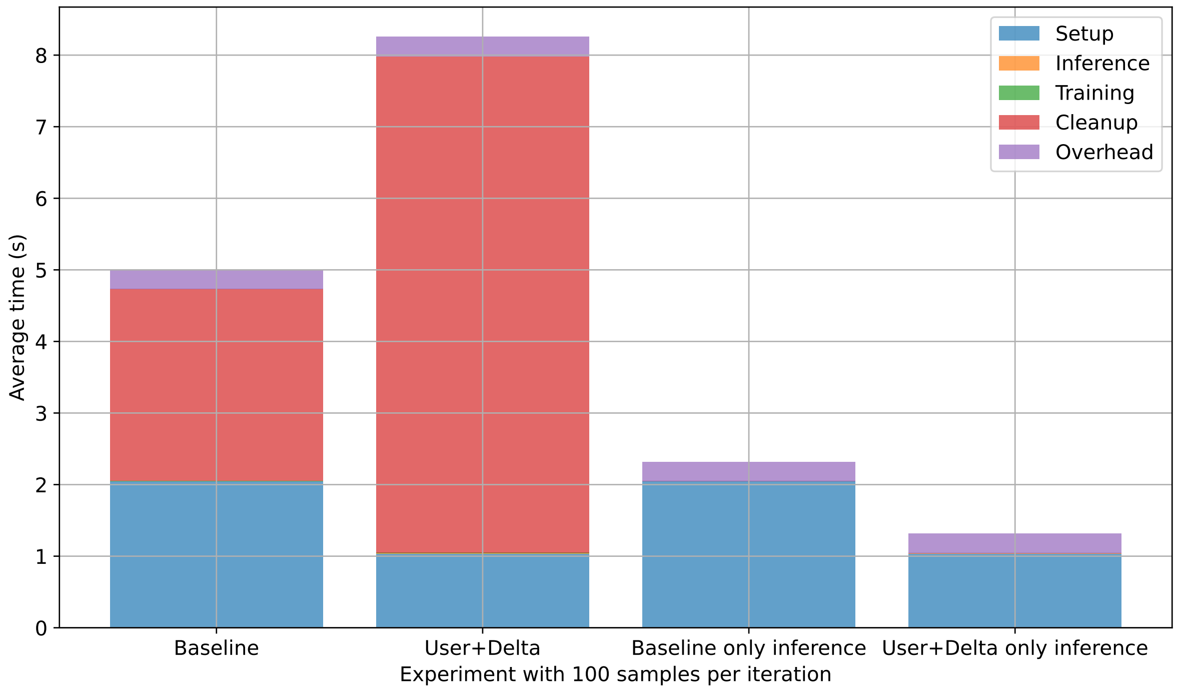


Figure 6.5: Averaged latency comparison, User-specific adaptation and Delta encoding.

The data used for this experiment is only the data from the first user in the testing dataset (user 115102). Since there is not enough data to have a significant experiment with this user, using the same parameters, the batch size was reduced to 100 for these experiments, both in the new experiment, as well as in the baseline. The delta encoding algorithm used for these experiments is Xdelta3 since BSDiff 4, the other algorithm available for the system, used excessive amounts of memory. In order to provide a complete picture of the time each part of the function call takes, both sides of the experiment are not using the cache, so they have to retrieve the model or the diff file and rebuild the model.

In Table 6.3 it is possible to see the sizes of the different files that may be used to save the model. The diff file together with the file of the base model is bigger than the file of the new model, however, if saving models for multiple users, eventually the total size will be smaller using delta encoding because of the reduced size of new diff files in comparison of the sizes of new models, saving cloud storage.

Also, as can be seen in Figure 6.5, even though the total time is bigger in this experiment, the setup time is lower for the version that uses delta encoding, and, because of that, if we were to perform only inferences eliminating the cleanup, the user latency would be smaller than the baseline. Due to the small batch size of these experiments, the inferences and the training times are insignificant and the function spends most of its execution preparing the model for training and sending it to remote storage. In Figure 6.4 we can see that, when this optimization is compared with other FaaS versions for training, it is never the cheapest option.

6.5 Cached state

The goal of this experiment is to understand the impact of the caching of the model in the function in different scenarios. For that, the cached state optimization will be enabled and the new model will not be uploaded to storage after training, requiring the next iteration to rely on the cached model. To evaluate the system, the time each step of the execution is measured and compared between the baseline, optimistic and pessimistic scenarios.

For this experiment, the controller activated the cache optimisation in the function. In this case, the function does not save the model after training, meaning that the next iteration has to rely on it still being in memory, or to re-train the model on those same data. Since, in these experiments, where the function is called immediately after finishing, the memory of the function is almost always saved, to experiment with the two scenarios, the controller can tell the function if it should attempt to use the objects that it has in memory or not. The scenario where the function can use the previous memory is the optimistic scenario and when it is not, it is the pessimistic scenario.

In Figure 6.6 we can see that the time that the clean up after performing the training, in the optimistic scenario using cache, is significantly lower than in the baseline in Figure 6.1, and that the setup time is not affected. The training time is kept at a small impact in the total time because no retraining needs to happen, since, in this scenario, the function was able to recover the fully trained model from cache. However, that may not always be the case. In Figure 6.4 we can see that, this version is the cheapest version between all the options studied, however it may not always be achievable due to the reliance in the cache of the FaaS function.

In Figure 6.7 we can see those two scenarios, as well as the pessimistic scenario. Here we can see that, in this case, the time the function execution takes grows as the cumulative data that has to be downloaded and trained, also grows. Because of that, in scenarios where it is not possible to maintain the model in the function's cache, it is better to avoid using this optimization. Another option to avoid big re-trainings, would be to frequently save the updated model to storage, keeping most of the iterations with this optimization. This way, we could have the fast iterations most of the times, while the worst case where the model is lost, does not pay a big penalty for retraining.

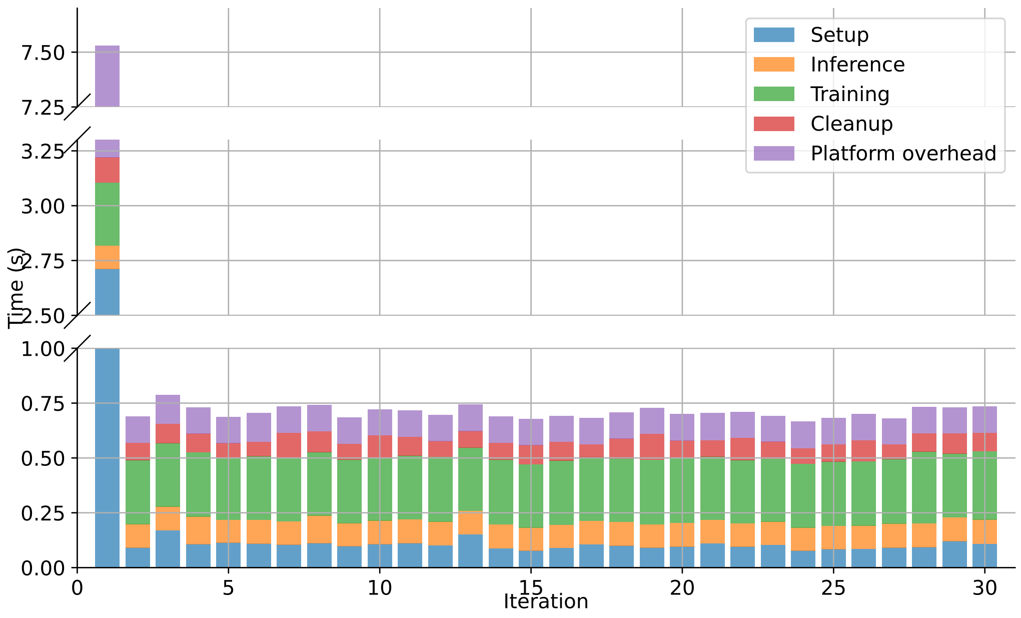


Figure 6.6: Cached model, optimistic scenario timing split.

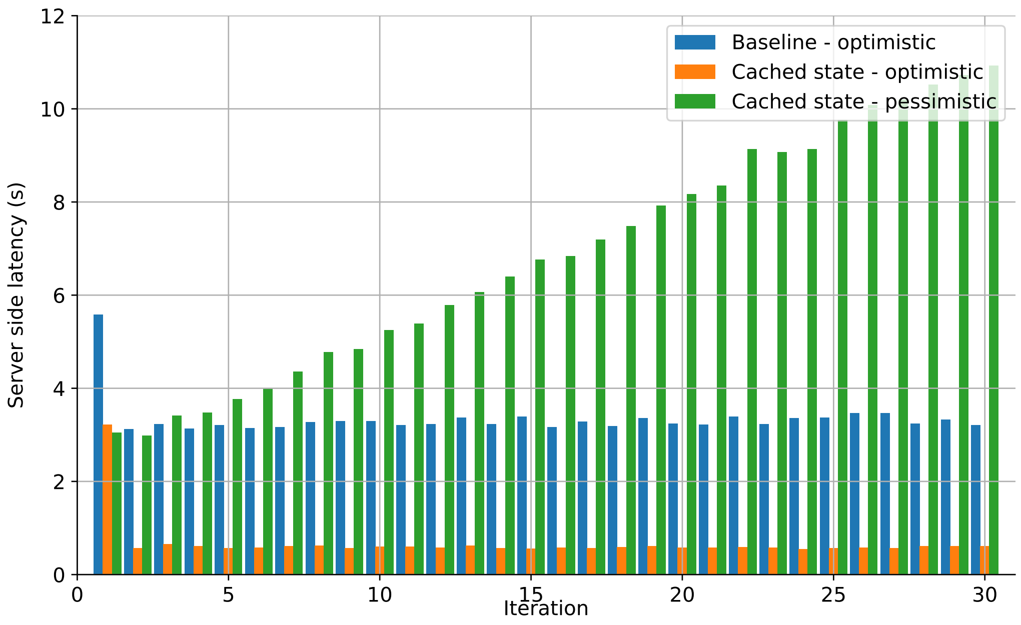


Figure 6.7: Latency comparison in different scenarios.

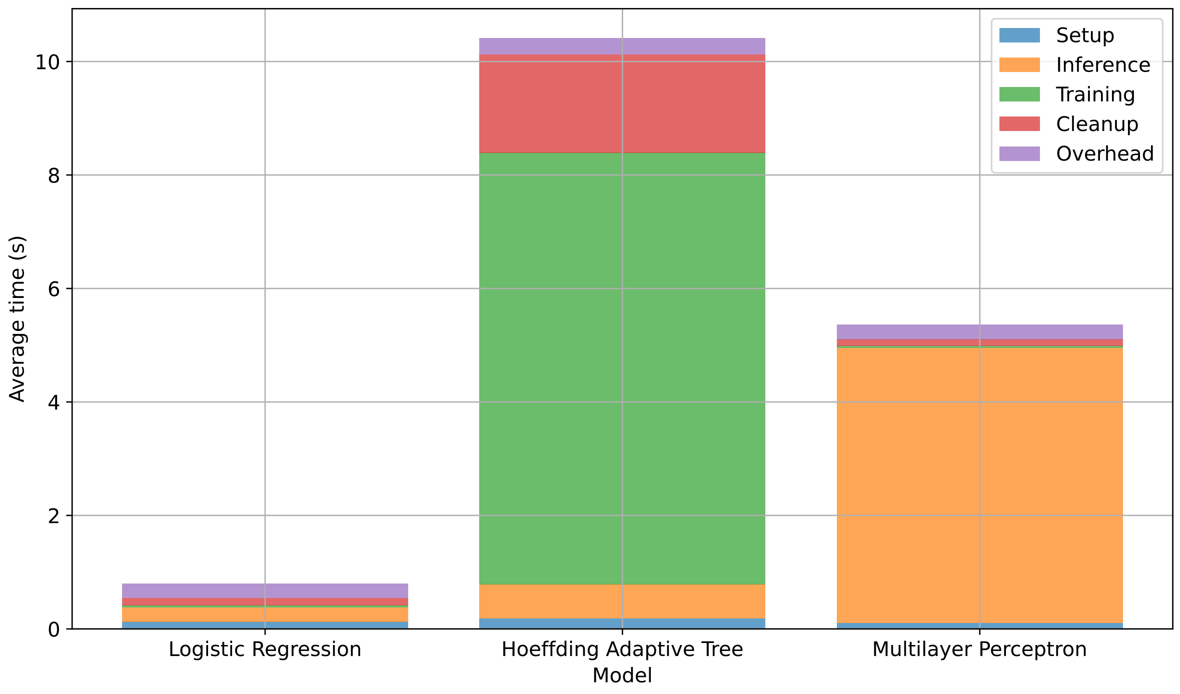


Figure 6.8: Comparisson between different ML models.

6.6 Other models

The goal of this set of experiments is to understand how the system handles different types of ML models. For that, it was tested with models from some of the major types of ML models: one linear model, one tree model and one neural network. The latency of the system when using these models is measured, per part of the function, to understand which models would be a better fit for this system.

Since this set of experiments uses different models, it was also chosen a more appropriate dataset for those models. It was chosen the Criteo 1TB Click Logs dataset [51], this dataset contains data about whether or not someone clicked or not an advertisement so it is used as a classification problem. For the experiments, only the day 0 was used. Since the performance of the model is not the focus of this evaluation, this part of the dataset was modified only to include the dense features and the samples that contain all of these features, to make it easier to build River models that can work with it. As with the previous experiments, the first 80% of the data was used to train a model that served as the starting point for the experiments, and the remaining data could be used for the experiments.

For the linear model, it was chosen the logistic regression. This model supports the `learn_many` method in River, so it was trained with a chunk size of 1000. This means that, while the function still receives 10000 samples per iteration, instead of learning each sample individually, it learns 1000 at a time. Since the metrics in River do not support this method, the inferences still have to be done one at a time. In Figure 6.8 it is possible to see the time each part of the execution took for the three models. Since this model is small, it does not take a long time to upload back to storage. Also, due to the faster training achieved with `learn_many`, making all the inferences and updating the metric takes significantly more time than training itself.

As for the tree model, it was chosen the Hoeffding Adaptive Tree [52], a model that can learn incrementally and can adapt to changes in the data distribution. As this model is slower to train than the other models, it was decided to train the base model only with 1000000 samples. When training this model, the training part is responsible for most of the latency.

The neural network used for the experiments was the Multilayer perceptron (MLP), the only neural network implemented in River at the time of writing. Despite the labels of the dataset being categories, for this experiment, they were treated as a continuous feature because the River's MLP only supports regression. This model also implements the `learn_many` method so it was trained with a chunk size of 1000. In this experiment, as in the linear regression, it is evident that a relatively smaller model and a learning function that can train with multiple samples are important factors in reducing the cleanup time and the training time respectively.

6.7 Discussion

This section presents a discussion of the results of the experiments, highlighting the most significant ones and explaining how they impact the viability on a system like this that trains ML models incrementally and the relevant trade-offs between different versions of the system.

The results of the first set of experiments, in Section 6.3, showed that training ML models online, both in IaaS and FaaS significantly increases the time latency of the system in comparison to training the model in batch and making only inferences as requested. The results also showed that there is a small RMSE improvement of the model when training online, which is explained by the model being able to incorporate the newer data into the model faster. Moreover, in a scenario where data arrives in longer intervals, the FaaS deployment becomes cheaper. In contrast, the IaaS deployment keeps the same price because it stays idle, consuming resources all the time, making the first one more attractive when there is not enough data to be always training.

As for the results of the delta encoding experiments, those showed that it is not worth it to use it in a situation where we need to update the model frequently, due to the increased cleanup time, but if we make more inferences in the same model, we can save some time by restoring the model from the diff file. Also, if there are many versions of a similar model, as is the case when doing user-specific adaptation, using delta encoding saves cloud storage because there is no need to save multiple times similar models.

The cached state experiment informs us that avoiding saving always the model makes a significant difference in the execution time of the function. However, it may not be feasible to avoid saving the model every time, for example, if we need to use the most current model often. Also, if the system accumulates lots of samples over time without saving a model that reflects these samples, when it loses the model, it incurs a big time cost due to the retraining of the model, making this option less viable. With this trade-off in mind, it is important to find a good saving policy to not risk losing big changes while still earning the savings of exploiting the function cached state.

Experimenting with different models, it became evident that having a model that can perform incremental learning using batches is key to having a smaller latency. In these cases, evaluating the model offline, after training might make sense to improve throughput.

In summary, Serverless Online Machine Learning can be viable when the data arrives in big intervals because in that case the savings of not paying for the idle VM outweigh the increased costs of using FaaS. Using the proposed optimization can help with increased savings in certain situation, in particular, relying on the cached state by not saving always the function to storage can have a significant impact in the savings.

7 Conclusion

Contents

7.1 Conclusions

This thesis has addressed the problem of Online Machine Learning using Serverless computing. It started by showing the relevant background and related work that lay the foundation for the rest of the work, informing us of the best use cases for online machine learning and the typical system architectures for systems of this type. Then it presents an idealized solution, that trains ML models online using FaaS and how it was implemented in practice. It also presented some optimizations that attempt to reduce the biggest hurdle of serverless Machine Learning, the communication time and, in certain situations, can improve the performance of the system.

The solution and the proposed optimizations were evaluated, and their results show the trade-offs that could emerge in a system used in the real world and how to overcome them. These results found that training ML models online using serverless can be cost-efficient in a situation where the data arrives sporadically, that using delta encoding for saving these models can be time-saving if used in a situation where it is used a lot to make inferences. The results also found that relying on the cached state between invocations to a FaaS function makes the latency of this system significantly shorter.

7.2 System Limitations and Future Work

One important limitation of the system is its inability to deal with distributed learning, solving this limitation would increase the capacity of the system to consume data. Taking full advantage of the serverless paradigm, there could be a system that scales up or down automatically as there is more or less data available to train. Using distributed learning would also allow the system to be used with large-scale models, more capable but that also require more resources to train, making them infeasible in our system.

Another possible path for future work is the use of different storage options. The biggest source of overhead in this system is downloading and uploading the model files, being able to store them faster would make the system more attractive, reducing communication time. Finding a faster storage, even if more expensive, could also make the system cheaper, since it could reduce the execution time of the function.

A different approach would be to build a complete platform for online Machine Learning that periodically generates new base models for the function, that are included with the image of the function. This way, the functions would not have to rely so much on storage, having the models locally before starting executing. This platform could also include the proposed optimizations and could be ran with larger scale experiments.

Bibliography

[1] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, “A continual learning survey: Defying forgetting in classification tasks,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, pp. 3366–3385, 2022.

[2] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, “Towards demystifying serverless machine learning training,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 857–871. [Online]. Available: https://doi.org/10.1145/3448016.3459240

[3] Customers love solutions, “1.1 on premise vs. serverless.” [Online]. Available: https://customers-love-solutions.com/?p=784

[4] I. Portugal, P. Alencar, and D. Cowan, “The use of machine learning algorithms in recommender systems: A systematic review,” *Expert Systems with Applications*, vol. 97, pp. 205–227, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417417308333

[5] W. Luo, B. Yang, and R. Urtasun, “Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net,” 2020. [Online]. Available: https://arxiv.org/abs/2012.12395

[6] A. Egg, “Online learning for recommendations at grubhub,” in *Proceedings of the 15th ACM Conference on Recommender Systems*, ser. RecSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 569–571. [Online]. Available: https://doi.org/10.1145/3460231.3474599

[7] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, “Dive into deep learning,” *arXiv preprint arXiv:2106.11342*, 2021.

[8] Google, “Descending into ml: Linear regression — machine learning — google developers.” [Online]. Available: https://developers.google.com/machine-learning/crash-course/descending-into-ml/linear-regression

[9] ——, “Clustering algorithms — machine learning — google developers.” [Online]. Available: https://developers.google.com/machine-learning/clustering/clustering-algorithms

[10] ——, “Decision trees — machine learning — google developers.” [Online]. Available: https://developers.google.com/machine-learning/decision-forests/decision-trees

[11] ——, “Neural networks: Structure — machine learning — google developers.” [Online]. Available: https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/anatomy

[12] A. Mnih and R. R. Salakhutdinov, “Probabilistic matrix factorization,” in *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., vol. 20. Curran Associates, Inc., 2007. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2007/file/d7322ed717dedf1eb4e6e52a37ea7bcd-Paper.pdf

[13] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[14] S. C. Hoi, D. Sahoo, J. Lu, and P. Zhao, “Online learning: A comprehensive survey,” *Neurocomputing*, vol. 459, pp. 249–289, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221006706

[15] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela, “Practical lessons from predicting clicks on ads at facebook,” in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ser. ADKDD’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–9. [Online]. Available: https://doi.org/10.1145/2648584.2648589

[16] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, “Communication-efficient distributed deep learning: A comprehensive survey,” 2020. [Online]. Available: https://arxiv.org/abs/2003.06307

[17] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, “Communication efficient distributed machine learning with the parameter server,” in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/1ff1de774005f8da13f42943881c655f-Paper.pdf

[18] M. Roberts, “Serverless architectures,” May 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html

[19] Amazon Web Services, “Amazon ec2.” [Online]. Available: https://aws.amazon.com/ec2/

[20] ——, “Aws sagemaker.” [Online]. Available: https://aws.amazon.com/sagemaker/

[21] ——, “Aws lambda.” [Online]. Available: https://aws.amazon.com/lambda/

[22] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, “Deep learning recommendation model for personalization and recommendation systems,” *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: http://arxiv.org/abs/1906.00091

[23] R. Wang, B. Fu, G. Fu, and M. Wang, “Deep & cross network for ad click predictions,” in *Proceedings of the ADKDD’17*, ser. ADKDD’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3124749.3124754

[24] M. Böther, F. Strati, V. Gsteiger, and A. Klimovic, “Towards a platform and benchmark suite for model training on dynamic datasets,” in *Proceedings of the 3rd Workshop on Machine Learning and Systems*, ser. EuroMLSys ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 8–17. [Online]. Available: https://doi.org/10.1145/3578356.3592585

[25] Amazon Web Services, “Amazon s3.” [Online]. Available: https://aws.amazon.com/s3/

[26] A. Vorobey, B. Fitzpatrick, and A. Kasindorf, “Memcached.” [Online]. Available: https://www.memcached.org/

[27] R. Ltd, “Redis.” [Online]. Available: https://redis.io/

[28] Amazon Web Services, “Amazon dynamodb.” [Online]. Available: https://aws.amazon.com/dynamodb/

[29] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, “Cirrus: A serverless framework for end-to-end ml workflows,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1145/3357223.3362711

[30] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, “Occupy the cloud: Distributed computing for the 99%,” 2017. [Online]. Available: https://arxiv.org/abs/1702.04024

[31] P. G. Sarroca and M. Sánchez-Artigas, “Mlless: Achieving cost efficiency in serverless machine learning training,” 2022. [Online]. Available: https://arxiv.org/abs/2206.05786

[32] IBM, “Ibm cloud functions.” [Online]. Available: https://cloud.ibm.com/functions/

[33] R. Guo, V. Guo, A. Kim, J. Hildred, and K. Daudjee, “Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers,” in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 779–794. [Online]. Available: https://proceedings.mlsys.org/paper/2022/file/ea5d2f1c4608232e07d3aa3d998e5135-Paper.pdf

[34] The ZeroMQ authors, “Zeromq.” [Online]. Available: https://zeromq.org/

[35] Microsoft, “Azure container instances.” [Online]. Available: https://azure.microsoft.com/en-us/products/container-instances/

[36] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, “Serverless data science - are we there yet? a case study of model serving,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1866–1875. [Online]. Available: https://doi.org/10.1145/3514221.3517905

[37] Google, “Google cloud functions.” [Online]. Available: https://cloud.google.com/functions

[38] R. DRUTA, C.-F. DRUTA, and I. SILEA, “Evaluation of remote data compression methods,” *Studies in Informatics and Control*, vol. 31, no. 1, p. 59–70, 2022. [Online]. Available: https://doi.org/10.24846/v31i1y202206

[39] Amazon Web Services, “Working with lambda container images.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/images-create.html

[40] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[41] Amazon Web Services, “Lambda quotas.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[42] ——, “Amazon elastic container registry.” [Online]. Available: https://aws.amazon.com/ecr/

[43] “Pickle - python object serialization.” [Online]. Available: https://docs.python.org/3/library/pickle.html

[44] J. Montiel, M. Halford, S. M. Mastelini, G. Bolmier, R. Sourty, R. Vaysse, A. Zouitine, H. M. Gomes, J. Read, T. Abdessalem, and A. Bifet, “River: machine learning for streaming data in python,” *Journal of Machine Learning Research*, vol. 22, no. 110, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1380.html

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[46] Josh MacDonald, “Xdelta3.” [Online]. Available: http://xdelta.org/

[47] C. Percival, “Naive differences of executable code,” 2003. [Online]. Available: http://www.daemonology.net/bsdiff/

[48] D. Korn, J. P. MacDonald, J. Mogul, and K.-P. Vo, “The VCDIFF Generic Differencing and Compression Data Format,” RFC 3284, Jul. 2002. [Online]. Available: https://www.rfc-editor.org/info/rfc3284

[49] F. M. Harper and J. A. Konstan, “The movielens datasets: History and context,” *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, dec 2015. [Online]. Available: https://doi.org/10.1145/2827872

[50] M. Grinberg, *Flask web development: developing web applications with python.* ” O’Reilly Media, Inc.”, 2018.

[51] Criteo, “Criteo 1tb click logs dataset.” [Online]. Available: https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[52] A. Bifet and R. Gavaldà, “Adaptive learning from evolving data streams,” in *Advances in Intelligent Data Analysis VIII*, N. M. Adams, C. Robardet, A. Siebes, and J.-F. Boulicaut, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 249–260.