2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA) | 978-1-7281-8470-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICMLA51294.2020.00104

Benchmarking Machine Learning Solutions in Production

Lucas Cardoso Silva*†, Fernando Rezende Zagatti*†, Bruno Silva Sette*†, Lucas Nildaimon dos Santos Silva*†, Daniel Lucrédio†, Diego Furtado Silva† and Helena de Medeiros Caseli†
*B2W Digital - Innovation Lab, São Paulo, SP, Brazil
†Federal University of São Carlos - Computing Department, SP, Brazil
Email: lucascardoso@estudante.ufscar.br, fernando.zagatti@estudante.ufscar.br, bruno.sette@ufscar.br, lucas.silva@estudante.ufscar.br, daniel.lucredio@ufscar.br, diegofs@ufscar.br, helenacaseli@ufscar.br

***Abstract*—Machine learning (ML) is becoming critical to many businesses. Keeping an ML solution online and responding is therefore a necessity, and is part of the MLOps (Machine Learning operationalization) movement. One aspect for this process is monitoring not only prediction quality, but also system resources. This is important to correctly provide the necessary infrastructure, either using a fully-managed cloud platform or a local solution. This is not a difficult task, as there are many tools available. However, it requires some planning and knowledge about what to monitor. Also, many ML professionals are not experts in system operations and may not have the skills to easily setup a monitoring and benchmarking environment. In the spirit of MLOps, this paper presents an approach, based on a simple API and set of tools, to monitor ML solutions. The approach was tested with 9 different solutions. The results indicate that the approach can deliver useful information to help in decision making, proper resource provision and operation of ML systems.**

I. Introduction

In recent years, many critical businesses depend on machine learning (ML) systems. In big companies such as Google [1] and Facebook [2], AI is becoming the business core. In these real-world scenarios, delivering a good machine learning solution is not only a matter of model quality and precise predictions. Many issues need to be addressed. These are part of what is known as MLOps – the operationalization of machine learning. MLOps deals with all aspects related to the process of putting a machine learning solution in production.

Real-world ML applications may present problems such as a lack of memory, long training time, or low-latency serving requirements [3]. Low performance is especially noticeable in algorithms employing techniques to automate their processes (AutoML) [4]. The constant availability of endpoints to perform prediction and training must also be part of a real-world machine learning solution [4]. These problems share the same root cause: resource limitations. Knowing exactly the amount of resources needed for an application is a problem faced by any system, and is no different in the world of AI. Dealing with machine learning problems while meeting high availability and performance requirements in a limited environment is challenging [3]. For example, performance can be increased through horizontal scalability to allow applications to remain responsive even in scenarios of high demand [5], but this requires support for parallel execution.

One way to solve these problems is to dive into a fully-managed cloud-based platform. Many platforms have built-in transparent, easy-to-use, redundancy and scalability functionalities, such as Amazon SageMaker[1], and Google Cloud Platform for AI[2]. In theory, these platforms can deliver resources on-demand, resulting in unlimited resources. However, there is a cost. If not properly controlled and monitored, cost can become exceedingly high. This is a real issue, as observed in a recent survey, where managing cloud spend was cited as one of the main challenges faced by enterprises [6].

Whether by deploying a solution in-house or in the cloud, it is necessary to know exactly how much resource is needed, including memory usage, CPU usage, GPU usage, task completion times (in training and serving), among other key metrics [7], [8], so that the hardware infrastructure can be correctly configured or acquired from the cloud. While data scientists may have a vague idea about which tasks and algorithms have more impact on resource consumption, a more precise measurement is necessary, because in the real world, with real users, new undetected challenges and problems often appear [7]. This is why monitoring the environment is often cited as a critical task in MLOps [9], [10], [11], [12].

There are tools for monitoring system resources. But many data scientists do not have experience with the necessary technologies for deploying and maintaining an ML system in production [13], and this includes knowing exactly what to observe and how to setup a proper monitoring and benchmarking environment. In this scenario, this paper identifies a list of the main resources that need to be observed in a typical ML solution. The paper presents a practical approach for collecting and presenting online and offline data regarding the observed resources, considering the perspectives of both the front and back ends of an ML solution. The paper also shows the approach being used on a collection of datasets selected from the literature.

These contributions are useful for data scientists interested in optimizing techniques or algorithms for resource consumption. With detailed and real-time resource monitoring, it is possible to see the impact of design decisions, identifying

[1] aws.amazon.com/sagemaker/
[2] cloud.google.com/ai-platform

978-1-7281-8470-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICMLA51294.2020.00104

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

memory leaks or excessive long loops. These contributions are also useful for practitioners, interested in deploying an ML solution with more confidence, basing their decisions on real data instead of personal feeling or previous experience.

II. Related work

As discussed before, the subject of this paper is part of MLOps – the operationalization of machine learning. Sridhar et al. [9] and Lwakatare et al. [10] discuss different operational aspects of machine learning, including the need for detailed observation of all aspects regarding a task execution.

Andrews et al. [11] describe how Facebook manages machine learning pipelines at scale, using a platform to encapsulate pieces of code (called operators), to facilitate the process. The paper also claims that the platform automatically monitors CPU and memory consumption so that the developer does not have to deal with them, as it is a relevant requirement for execution at large scale. As another big company example, Baylor et al. [5] present TFX, a TensorFlow extension proposed by Google to simplify the process of deploying an ML solution in a reduced time frame. The platform supports many requirements for keeping the solution running in different situations and under different demands. These last two requirements are particularly related to this research, as being able to monitor latency and resource usage was cited as an important requirement for maintaining the platform.

Nguyen et al. [12] presented a survey to describe the machine learning and deep learning frameworks and libraries that tended to be used in the state-of-the-practice. When discussing the results, CPU/GPU and memory optimization are frequently cited as critical for choosing a particular framework or library, so being able to monitor these resources precisely is an important requirement.

Lim et al. [14] specify an operational lifecycle scheme for machine learning projects (MLOps) in the manufacturing industry to increase productivity. According to the authors, using machine learning in production means more than just training and running models. For example, the use of off-premise services is not adequate due to confidentiality. Therefore, using cloud platforms is not an option. In their study, inference time (time between request and response) was measured as an important indication for the quality of their solution.

Flaounas et al. [7] discuss some implementation aspects for ML solutions, or what they call "productionisation" of a prototype. They argue that implementing is not restricted to making the prototype work. Actual software development is needed, and this implies that testing in the real world is critical, as real users, with real data, may show that the initial requirements are not being satisfied. Monitoring for quality and resource consumption is important, during construction and operation, where the solution has to be maintained running adequately and constantly updated to correct eventual changes in quality caused by new data or new requirements.

According to Breck et al. [8], testing and monitoring are key considerations for assessing the production-readiness of an ML system. The authors present different tests that need to be conducted, divided into four categories: Tests for Features and Data, Tests for Model Development, Tests for ML Infrastructure, and Monitoring Tests for ML. Some tests require monitoring resources, such as detecting training speed, serving latency, throughput, or RAM usage, and also detecting staleness, which is when the pipeline fails to train and deploy sufficiently up-to-date models. These are critical to maintain the system in production.

As it can be seen, being able to know how an ML solution is making use of existing resources is often cited in the literature as an important feature for real-world products. There are already a variety of tools for monitoring these resources. Most operating systems already have built-in tools for monitoring overall and per-process memory and CPU usage, so this is not a new feature. There are also tools such as *Prometheus*[3] and *Docker*, which provide a wide range of monitoring options. However, properly configuring these tools for displaying the desired results in an intuitive way and being able to easily identify which parts of the ML pipeline are responsible for the observed data requires careful tuning and preparation.

III. What to observe and why?

A Machine Learning system is like any other regular system in essence, since it encompasses one or more software processes running on the top of the operating system. However, there are specific tasks that typically exist in an ML solution, forming a pipeline, as shown in Figure 1.

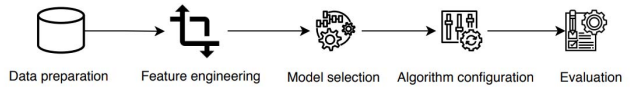


Fig. 1. Prototypical ML pipeline

Data preparation aims at improving the quality of a dataset through methods such as value imputation (replacing missing data with default values) or feature transformation (creating new features from existing ones). Next, the feature engineering intends to select and build features from the original data in order to enable the learning of the model by the chosen algorithm. Then, in the algorithm selection and configuration steps, the expert selects and configures the algorithm that best suits the data structure. Finally, in the last step, the model is trained and evaluated. This pipeline is not always executed in a single direction, one step after another. It is common to revisit some task in order to refine the overall results. This is more frequent during initial development, when the exact parameters for all algorithms are still being adjusted, but is also a reality even after the system is deployed.

When executing these tasks, there are five main observations that need to be considered: task completion time, CPU and GPU usage, memory usage, disk input/output (IO) operations and network traffic.

[3]prometheus.io

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

A. Task completion time

The tasks of an ML pipeline may be performed offline or online. Offline (or "batch") tasks do not have the requirement of delivering the results immediately. For example, gathering data from multiple sources, imputing missing values into a table or training a model are all tasks that may take a long time to be performed. Online tasks, on the other hand, have to deliver a quick response, as they are normally used as part of another system to respond to end-user requests, for example, to perform a prediction or classification in real time. At first glance, it may seem that the requirements are simple: online tasks should execute as fast as possible, and offline tasks may take as long as they need to finish. While these simple assumptions may be enough for some cases, it is often necessary to be more explicit and precise than that.

As discussed earlier, online ML tasks are typically executed as part of a running system. They must respond to other tasks or end-user requests in a time that is suitable to the overall business requirements. Many QoS (Quality-of-Service) contracts define specific performance thresholds [15], sometimes in the order of milisseconds [5]. Therefore, instead of defining the time in terms of "as fast as possible", it must be defined in terms of "in less than `<amount> <time units>` per request", so that it is known if the overall system's QoS contract will be honored. It is also important to consider that these tasks may be executed thousands of times per second, depending on the scenario and on specific conditions. In most cases, it is not enough to have a rough estimate, and performing simulations and tests with high loads is sometimes the most reliable way to determine the exact performance.

Offline tasks may also suffer from time constraints. In most cases, these tasks have to be performed sporadically (e.g. a model is re-trained once a month, or once a week), therefore it makes little difference if the task finishes in three minutes or three hours. However, with the need for constant updates, sometimes with fresh models being trained daily [5], these times are not negligible and need to be specified and measured constantly to avoid staleness [8].

Sometimes, it is also important to observe task progress, in addition to completion time, specially in initial development, when the team may have little idea about how much time some task may last. In long tasks, there must be some way to estimate how much time it will take to complete, either to keep the people/systems that will be using it informed, or to determine how much money will be spent in the cloud.

B. CPU and GPU usage

Measuring CPU and GPU usage throughout an ML task involves determining how much of these resources are being consumed by that particular task. The values vary from 0% to 100%, although some operating systems may report values higher than 100% when multiple cores are being used. Normally, the interesting information is not the percentage by itself, but how long the CPU is being used at values close to 100%. This is an indication that if more processing power is available, task completion time might be reduced.

Knowing these usage patterns is important for decision making and problem diagnosis [16]. Normally, the solution is to try to parallelize the execution to make use of multiple CPUs or GPUs [10], or to run on distributed platforms [14], [11], but this may require refactoring the software and dealing with more complex deployment configurations.

Proper and precise CPU observation over time also allows tests and experiments with different resource configurations for the different ML tasks. For example, a developer might temporarily increase the number of allocated CPUs for a task and measure its usage to determine if this is indeed improving the performance. This knowledge may prevent one from buying new hardware in a desperate attempt to meet tight QoS constraints, only to find out that it was not the lack of CPU that was causing the observed delays.

With precise measurements, it might be possible to pinpoint the exact moment where the CPU is being overloaded, helping the team to focus the effort on improving or parallelizing that particular task, instead of the entire pipeline.

C. Memory usage

Monitoring memory can help to detect code defects [16]. Memory issues are very similar to CPU/GPU ones, as computer memory is another resource that is being consumed by ML tasks and must be monitored for excessive loads during particular tasks. So, everything that was discussed in the previous section also applies here. There is one difference that is worth mentioning, however. Memory is managed differently by each environment, operating system or programming language. Languages that support garbage collection, for example, give programmers fewer choices to contain memory leaks and identify high consumption jobs than languages with explicit memory management routines. This may difficult the identification of the locations where to focus the optimization work. For example, with garbage collection, memory may take a little while to be freed after some task has finished, so the team must take this into account.

At first glance, it may seem that there is not a lot the programmer can do to optimize memory usage, as there are different layers of memory enhancement functionality in the operating system, such as caching and virtual memory. Caching allows frequent data to be more rapidly accessed, and virtual memory allows processes to use more memory than there is actually available as RAM. Using these features do not require explicit programming, but there are some techniques that the programmer can use to make better use of existing resources, such as using bit variables, and reusing strings through pointers.

Again, it is always possible to just acquire more RAM, as it is a relatively cheap resource, compared to CPU, for example. With more RAM, there is less need for CPU and disk access operations, and the task completion times could be reduced. However, without knowing exactly which tasks are consuming more memory and in which conditions, acquiring more RAM or investing in fine-tuning and optimization may not bring the expected return of the investment.

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

D. Disk input/output (IO) operations

Disk access is one of the main causes for performance problems, as this is a costly operation. Even when there is no actual "disk" (e.g. a Solid-State Drive), reading and writing files take more time than memory access operations. According to Zhou et al. [16], disk input/output is an important telemetry data to help in problem diagnosis.

In practice, accessing the disk during an ML task is mandatory, because this is most likely where the training data is stored. So, in data acquisition and preparation, before training and other tasks, the disk will be accessed more frequently. It is possible to reduce disk access in these tasks by implementing application-specific memory caching policies.

But even in those tasks where the ML solution is working exclusively on the memory, the operating system's virtual memory management will perform disk I/O. So the challenge is, again, to identify the moments when disk IO operations are occurring more frequently, to attack the problem, either by increasing the amount of RAM or by optmizing the code.

E. Network traffic

Network traffic is the main source of slowness in any computer process, therefore it must be minimized. It is, as with disk IO operations, practically inevitable, as most sources of information for an ML project will be obtained from the network. For these reasons, it is a helpful metric for diagnosing problems [16].

Although there is not much that can be done, since data has to be transferred from somewhere, the programmer can reduce the amount of traffic through techniques such as local caching and careful choice of communication technologies, for example choosing a reduced payload framework such as gRPC[4]. Knowing exactly which tasks are consuming more network traffic can help to decide what to do.

IV. Benchmarking ML Solutions in Production

As discussed before, there are already many tools available for monitoring the resources directly in the operating system, therefore it is not the point of our contribution to develop an alternative tool or library. Instead, we focus on showing how a machine learning solution can be instrumented for monitoring the resources described in the previous section, with little effort from the developer, so that meaningful information can be obtained to help with the operational aspects of the ML solution. The approach is based on the following elements:

Docker containers are used to run the application. Containers facilitate different MLOps tasks, such as deployment and version control. It also encapsulates an application into a well-defined environment, making it easier to monitor resource usage from an external point of view. There are many tools for working with container monitoring, but in this paper we use *docker stats*, as it is one of the easier ways to gather resource information from a container and it already covers the resources discussed in the previous section.

[4]grpc.io

A simple API, called *Ubenchmark*, was created to allow the developer to easily mark points-of-interest (POI) inside ML tasks. A POI normally represents the beginning or the end of an ML task, such as preprocessing, training and predicting. It can also be used to mark a specific part of a task, such as before/after reading a large file, or before/after applying a specific data transformation technique. These POI will appear together with the final data, so that the developer is able to precisely determine which parts of the code are being executed. This allows to easily separate resource usage for each task and monitor task progress through specific points in time. The API was developed in Python and is publicly available as part of Apache Marvin-AI[5] and on GitHub[6].

Matplotlib[7] is used by *Ubenchmark* to display the resource usage data graphically. It is a Python-based visualization tool that can display 2D graphics.

Apache *JMeter*[8] is used to conduct stress tests with the online tasks. *JMeter* can be configured to flood a server with continuous requests from multiple simultaneous threads, simulating high-demand scenarios. By combining a particular POI from *Ubenchmark* with the beginning and end of *JMeter* tests, it is possible to know exactly when these requests start and finish, making it easier to see the server's response to the simulated period of high demand.

The API usage is illustrated in Listing 1, which shows batch actions for preprocessing (line 4) and training (line 8).

Listing 1. Batch actions annotated with different points of interest (POI)

```
1  poi_marker = POIMarker('times_batch')
2  time.sleep(10)
3  poi_marker.add_poi('a')
4  run_preprocess()
5  poi_marker.add_poi('b')
6  time.sleep(5)
7  poi_marker.add_poi('c')
8  run_training()
9  poi_marker.add_poi('d')
```

First (line 1), a `poi_marker` variable is initialized to serve as a marker for adding points of interest (POI). Next, four POIs are added. POIs "a" (line 3) and "b" (line 5) mark the beginning and end of the preprocessing step (line 4). POIs "c" (line 7) and "d" (line 9) mark the beginning and end of the training step (line 8). In this example, these POIs will allow the developer to know exactly what is happening in these tasks.

Among these lines, a forced delay is introduced (lines 2 and 6) to guarantee that the monitoring API is capturing the resource utilization values for each task separately. These forced delays are optional and should not be used when measuring time. But they help to make the visualization clearer for other metrics, specially when the tasks are very short.

To monitor the execution, the developer must use two scripts from *Ubenchmark*. The first script is a *makefile* script that creates a *Docker* image with the necessary tools, dependencies and configurations. When running this *Docker* image, the

[5]marvin.apache.org
[6]github.com/cardosolucas/ubenchmark
[7]matplotlib.org
[8]jmeter.apache.org

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

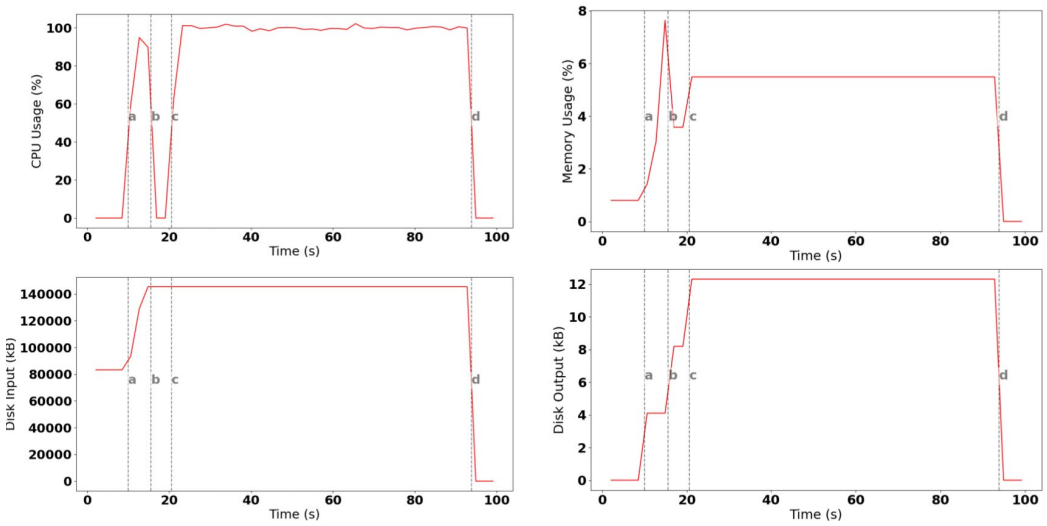


Fig. 2. Resource usage (CPU, RAM and disk I/O) for the code in Listing 1.

resource utilization and POI data will be saved by *docker stats* into a preconfigured folder. Then, a second *Ubenchmark* script will load these data into *Matplotlib* and launch it so that the information can be visualized.

Using *docker stats* incurs in some overhead, but according to the Docker documentation, it is not very large [17]. Adding POIs can also incur in overhead, but since they are meant to mark the beginning and end of a task, they should not interfere with the task itself. Generating the visualization is performed only after the monitoring is complete, hence there is no overhead problem.

Now we show an example of the approach being used to monitor how an ML solution consumes the resources of a computer, running 8 GB of RAM, Dual core Intel Core i3-5005U CPU, 7200 RPM HDD and a Realtek RTL810xE PCI Express Fast Ethernet controller. The following examples are from a solution based on Scikit-learn version 0.23.1, algorithm SVC (C-Support Vector Classification), in Python 3.7 for a dataset from kaggle[9]. For the online tasks, a production server Gunicorn version 20.0.4 was used, with 4 workers configured to serve the endpoints. Figure 2 shows an example for the resource usage data throughout time for the code of Listing 1.

It is possible to see that the execution took a little over 90 seconds, including the preprocessing and training tasks, and the forced delays. The four POIs ("a", "b", "c" and "d") are clearly visible, as well as the forced delays (before "a" and between "b" and "c"). This graphic shows that both tasks (preprocessing and training) are using a single core, as only 100% CPU usage is being registered (top left). Memory usage (top right) is more intense during preprocessing (8% peak) and smaller, but constant, during training (6%). It is also visible that disk I/O happens only during preprocessing (bottom left/right, between "a" and "c"), as the amount of data read/write does not increase during training (bottom left/right, between "c" and "d"). As expected, there is more disk input than output in these tasks, as shown by the vertical axis. It is also possible to see that there was some disk output during a forced delay (between "b" and "c" in the bottom right), probably due to buffered disk operation.

The example of Listing 1 consists of batch actions. These are executed once every now and then, for preprocessing or training, for example. To monitor online tasks, it is necessary to consider a different setup.

Normally, an online task is part of another application, and must respond to requests, normally through a REST or gRPC endpoint. Therefore, there is a server where these endpoints are being served, and a client, where these endpoints are being consumed. Also, online tasks should normally be quick to respond and support a certain amount of simultaneous requests without delays or failures. The exact amount of simultaneous requests depends on each business scenario. For example, a small company's ML solution being used only by upper management will probably have only a few simultaneous requests in the most busy moments, while a large company's ML solution being used by customers during holiday shopping may have peaks of tens of thousands simultaneous requests.

Ubenchmark can also be used to monitor these tasks in both sides and considering these different scenarios. In conjunction with a stress test tool, such as Apache *JMeter*, *Ubenchmark* can help to visualize resource consumption in these conditions. First, the endpoint server must be configured to mark POIs using *Ubenchmark*, as shown before. We recommend the creation of two special endpoints, one for the beginning and one for the end of the stress tests. In each endpoint, nothing is done except for adding a POI marker for these events.

Next the developer must configure the stress test tool (in our case, *JMeter*) to simulate the desired scenario. For example, it

[9]www.kaggle.com/c/santander-customer-transaction-prediction

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

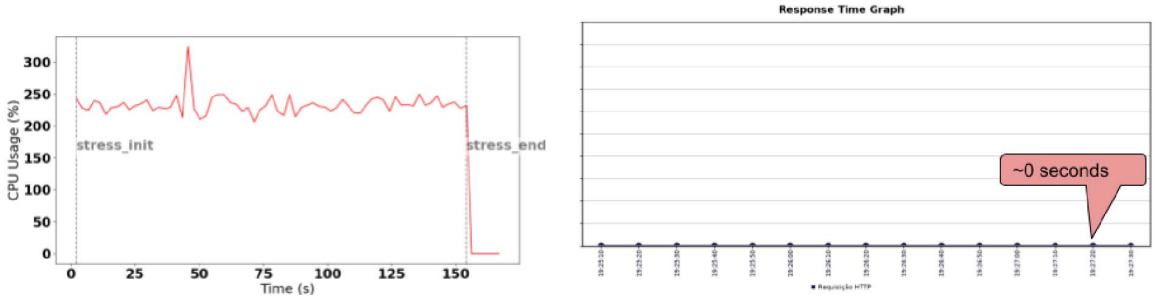


Fig. 3. Example CPU and response time for a stress test of online tasks simulating 50 simultaneous requests.

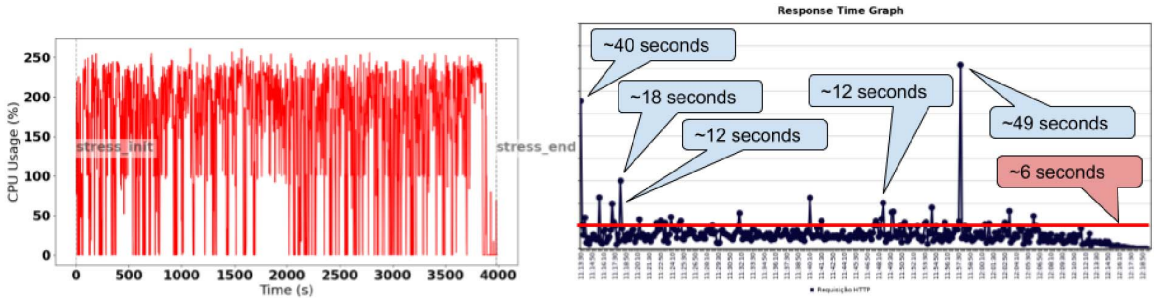


Fig. 4. Example CPU and response time for a stress test of online tasks.

may be configured to start 100 threads simultaneously sending 100 requests in sequence. It should also be configured to send a single request for each POI endpoint mentioned before, before and after the tests, to allow better visualization.

Ideally, this tool should be executed in a different machine than the server. In our case, we used a second computer to run JMeter, running 4 GB of RAM, Dual core Intel Core i3-5005U CPU, 7200 RPM HDD and a Realtek RTL810xE PCI Express Fast Ethernet controller. The two computers were connected in a LAN network using a Fast Ethernet network connection supported by a D-link router, model DIR-615. It would be even better if many machines were used as clients, to avoid bottlenecks in the client and simulate more clients.

Figure 3 shows the results of a stress test simulating 50 threads, each one sending 1000 requests in sequence. This test took around 150 seconds to complete. The left side shows CPU usage in the server during this time, which operates mostly between 200 and 250% to respond to the requests. It is possible to see the POIs "stress_init" and "stress_end" delimitating the exact moments when the test begins and ends. The right side of Figure 3 shows the response times being observed in the client during this time. In this case, it is possible to see that most requests are being returned in very short time, practically indistinguishable from 0 in the graph. *JMeter* also reports the number of errors, which in this case was 0%. These results are an indication that this particular server is being able to respond to this amount of requests properly, and should behave well in production under these conditions.

In contrast, Figure 4 shows the results of a stress test simulating 1000 threads, each one sending 1000 requests in sequence. This test took almost 4000 seconds to complete. The left side shows CPU usage in the server during this time, which operates mostly between 100 and 250% to respond to the requests, but has sometimes reduced to 0%, which indicates that the server stopped working in several occasions. The right side of Figure 4 shows the response times being observed in the client during this time. In this case, it is possible to see that there are some requests that are taking too long to return, with some peaks reporting more than 10 seconds and others taking more than 40 seconds. Also, there are too many requests that are close to the first horizontal line of the graph (the red line), which marks a 6-second response, normally unacceptable for a good quality service. *JMeter* also reports the number of errors (requests without a response), which in this case was 5,36%. These results are an indication that this server is not being able to respond to this amount of requests properly.

As another example of how this information could be used, Figure 5 shows the disk input and RAM usage for a solution using a label encoder[10] for the Microsoft malware dataset[11]. This execution was interrupted during preprocessing (POI "b" does not appear in the graphs). Disk input was high, but this is expected because it is a large dataset. The problem was the lack of memory, as the algorithm used constantly more memory over time, until it reaches close to 100% and the process is interrupted. In this case, either more memory is necessary or a different algorithm must be implemented for the preprocessing task.

[10]scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html

[11]www.kaggle.com/c/microsoft-malware-prediction/overview

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

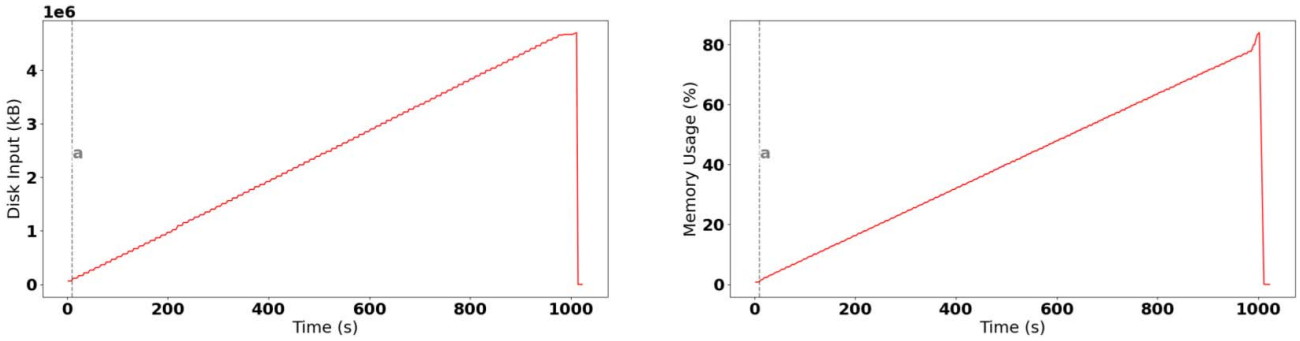


Fig. 5. Disk input (left) and RAM (right) usage for the solution to the Microsoft malware dataset. The values are cumulative over time.

Dataset	Data type	# of training samples	# of test samples	# of attributes	Observations	Tool/ Algorithm
Dog Breed	Images	10222	10357	N/A	120 classes	Keras / CNN
Invasive Species	Images	2295	1531	N/A	Binary classification	Keras / CNN
Plant Seedlings	Images	4750	794	N/A	12 classes	Keras / CNN
Taxi Fare	Attribute-value	55423856	9914	7	Regression	scikit-learn / Random forest regressor
Santander Value	Attribute-value	4459	49342	4991	Regression	scikit-learn / Random forest regressor
House Prices	Attribute-value	1460	1459	79	Regression	scikit-learn / LassoCV
Santander Customer	Attribute-value	200000	200000	200	Binary classification	scikit-learn / SVM
Don't Overfit II	Attribute-value	250	19750	300	Binary classification	scikit-learn / SVM
Microsoft Malware	Attribute-value	8921483	7853253	82	Binary classification	xgboost / xgboost

TABLE I
DATASETS USED IN THE EVALUATION. THESE CAN BE FOUND IN KAGGLE.COM

A. Tests and evaluation

We experimented our approach in 9 datasets. We tried to select datasets with different features and sizes, to check how useful our approach is in diagnosing problems when using them. Table I shows the datasets used in this evaluation and the tool/algorithm that was implemented. Table II shows the main results for the batch tasks. Two solutions were automatically interrupted (Microsoft Malware and Taxi Fare) due to the lack of RAM. The data shows that the maximum amount of RAM used was almost 78% for both cases. The amount of CPU used was around 100%, indicating that the chosen tool/algorithm was not making use of multiple processing cores. For these cases, additional RAM may solve the problem, as well as another algorithm to split processing in multiple cores.

Dataset	Task time (secs)	Max CPU	Max RAM	Max Disk I/O
Microsoft Malware	1082*	101%	77.7%	3.46GB / 0B
Taxi Fare	127*	98%	77.4%	2.93GB / 4.1KB
Dog Breed	600**	404%	27%	437MB / 12.3KB
Plant Seedlings	600**	397%	24%	1.79GB / 12.3KB
Invasive Species	592	390%	22%	2.79GB / 3KB
Santander Value	130	396%	7%	150,000KB / 8KB
Santander Customer	100	100%	7.5%	140,000KB / 12KB
Don't Overfit II	17	15%	0.8%	60,000KB / 15KB
House Prices	15	30%	0.8%	80,000KB / 5KB

TABLE II
RESOURCE USAGE FOR THE BATCH TASKS. * AUTOMATICALLY INTERRUPTED BY THE TOOL. ** MANUALLY INTERRUPTED DURING TRAINING AFTER 10 MINUTES.

Two solutions were manually interrupted after 10 minutes (Dog Breed and Plant Seedlings), as they were taking an excessive amount of time, and the estimated time for completion was several hours. For these cases, all four cores were being used at maximum capacity, as the data for maximum CPU usage shows values close to 400%. We can conclude that to reduce this time, additional CPUs or a GPU would be effective.

For the other solutions, the time varied from a few seconds (House Prices) to almost 10 minutes (Invasive Species). These times may be acceptable, but if less time is desired, additional processing power would probably reduce the time for those solutions with reported CPU values close to 400% (Invasive Species and Santander Value), as these represent the maximum allowed in this hardware. And for the solution where CPU was 100%, a different algorithm, which makes use of multiple cores, could reduce the task time.

Disk input was higher in the larger datasets, as expected, and only a small amount of disk output was noticed. For the larger datasets, a faster disk would probably help to reduce the time. In all solutions, there was no network traffic.

Table III shows the main results for the online tasks. Only the solutions for which the batch tasks were completed are shown, as there were no trained models for the others. Scenarios with 50 and 350 simultaneous threads were simulated. Each thread was configured to submit 1000 requests. For all scenarios with 50 threads there were no errors (requests without a response). The average response times were also adequate, except for the "Invasive Species" dataset, but this is expected, as this task involved processing an image in real time. Since CPU was getting close to the maximum (400%) maybe additional processing could reduce this time.

With 350 threads, all solutions had some percentage of errors, which indicate that this particular server is not able to support this amount of simultaneous requests. Again, the solution for "Invasive Species" had more errors. The maximum

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.

# of threads	Dataset	% Errors	Ave. resp. time	Max CPU	Max RAM	Max Network I/O
50	Invasive Species	0%	1592	352%	17.1%	30MB / 30MB
	Santander Value	0%	141	246%	2.7%	93.5MB / 28.4MB
	Santander Customer	0%	147	255%	2.7%	93.6MB / 28.4MB
	Don't Overfit II	0%	115	232%	2.7%	83.5MB / 28.4MB
	House Prices	0%	79	288%	2.7%	82MB / 29MB
350	Invasive Species	4.18%	9716	352%	17.1%	238MB / 196MB
	Santander Value	1.02%	1102	254%	2.7%	691MB / 199MB
	Santander Customer	1.06%	1156	288%	2.7%	692MB / 199MB
	Don't Overfit II	1.47%	899	261%	2.7%	612MB / 199MB
	House Prices	1.87%	876	239%	2.7%	600MB / 205MB

TABLE III
RESOURCE USAGE FOR THE ONLINE TASKS

amounts of CPU and RAM data confirm these observations. However, in all tests, resources did not appear to reach a critical limit, therefore there are probably other limitations to this environment that is causing the errors.

Finally, the amount of network traffic observed is proportional to the number of threads, what indicates that there is no additional network traffic other than the request/response data. These values may help to estimate cloud costs. These solutions use REST endpoints. If the costs are two high, smaller payload frameworks, such as gRPC, could be an alternative.

V. CONCLUDING REMARKS

Monitoring ML solutions is often cited as important for MLOps, but in relation to efficiency and model quality it is left behind. Affecting factors such as model drifting and feedback loops must be detected in order to keep the solution alive and relevant for a long time. Here we introduce another dimension to this monitoring, which is often cited as important but not dealt with explicitly, particularly in the academy. In the real world, computing resources are limited and may be a real problem. Developing a perfectly trained model, with excellent precision, but that requires a lot of resources, may end up as a business failure.

Monitoring hardware resources and task completion time is not particularly difficult, as there are many tools available. But properly configuring them in order to produce meaningful results in a fast and simple way can be challenging, especially for ML professionals who are not experts in system operations.

In the spirit of MLOps, in this paper we try to reduce this gap by presenting an easy-to-use approach, based on a simple API and set of tools for carrying out this job. The evaluation results show that not only the approach is feasible but can deliver useful information, helping to identify problems and providing detailed data to support deployment and operation in a production environment.

As future work, we plan to investigate more detailed information regarding the resources, aiming to provide better profiling capabilities to *Ubenchmark*, such as different kinds of memory used by the operating system, individual processor information, refined network traffic details, among others. We also plan to investigate how to facilitate the configuration of automatic events for a running server based on the monitored resources, so that the server can respond to high resource consumption automatically.

ACKNOWLEDGMENT

We would like to thank B2W Digital - Innovation Lab - and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - for the financial support.

REFERENCES

[1] B. Zerega, "AI weekly: Google shifts from mobile-first to ai-first world," Available at (accessed may 2020): https://venturebeat.com/2017/05/18/ai-weekly-google-shifts-from-mobile-first-to-ai-first-world/, 2017.

[2] R. Parekh, "Designing AI at scale to power everyday life," in *23rd Intl. Conf. on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, p. 27.

[3] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Softw. engineering challenges of deep learning," in *2018 44th Euromicro Conf. on Softw. Engineering and Advanced Applications (SEAA)*, 2018, pp. 50–59.

[4] J. Zhou *et al.*, "Katib: A distributed general AutoML platform on kubernetes," in *USENIX Conf. on Operational Machine Learning (OpML 19)*, 2019, pp. 55–57.

[5] D. Baylor *et al.*, "TFX: A TensorFlow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, p. 1387–1395.

[6] Flexera, "2020 state of the cloud report," Available at (accessed may 2020): https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020, 2020.

[7] I. N. Flaounas, "Beyond the technical challenges for deploying machine learning solutions in a software company," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1708.02363

[8] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "What's your ML test score? a rubric for ML production systems," in *Reliable Machine Learning in the Wild Workshop - NIPS 2016*, 2016.

[9] V. Sridhar *et al.*, "Model governance: Reducing the anarchy of production ML," in *Usenix Annual Technical Conf.* USA: USENIX Assoc., 2018, p. 351–357.

[10] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic, "A taxonomy of software engineering challenges for machine learning systems: An empirical investigation," in *Agile Processes in Softw. Engineering and Extreme Programming*. Springer, 2019, pp. 227–243.

[11] P. Andrews, A. Kalro, H. Mehanna, and A. Sidorov, "Productionizing machine learning pipelines at scale," in *Proceedings of the 32nd International Conference on Machine Learning, Lille, France*, vol. 37. JMLR: W&CP, 2015.

[12] G. Nguyen *et al.*, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artif. Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[13] W. Finzer, "The data science education dilemma," *Technology Innovations in Statistics Education*, vol. 7, no. 2, 2013.

[14] J. Lim, H. Lee, Y. Won, and H. Yeon, "MLOp lifecycle scheme for vision-based inspection process in manufacturing," in *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. Santa Clara, CA: USENIX Association, May 2019, pp. 9–11. [Online]. Available: https://www.usenix.org/conference/opml19/presentation/lim

[15] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *Intl. Journal of Cooperative Information Systems*, vol. 17, no. 02, pp. 223–255, 2008.

[16] H. Zhou, J. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin, "An empirical study on quality issues of production big data platform," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 17–26.

[17] Docker, "Runtime metrics," Available at (access July 2020): https://docs.docker.com/config/containers/runmetrics.

Authorized licensed use limited to: UNIVERSITY OF WESTERN ONTARIO. Downloaded on May 25,2021 at 21:48:50 UTC from IEEE Xplore. Restrictions apply.