Distributed and in-situ machine learning for smart-homes and buildings: application to alarm sounds detection

Amaury Durand
Telecom ParisTech
Paris, France
amaury.durand@telecom-paristech.fr

Yanik Ngoko
Qarnot Computing and University of Paris 13
Paris, France
yanik.ngoko@qarnot-computing.com

Christophe Cérin
University of Paris 13
Paris, France
christophe.cerin@lipn.univ-paris13.fr

***Abstract*—The need to build real-time, context-aware, secure and intelligent systems is pushing forward in-situ machine learning. In this paper, we consider the implementation of such systems with the computing model promoted by Qarnot computing. Qarnot introduced an utility computing model in which servers are distributed in homes and offices where they serve as heaters. The Qarnot servers also embed several sensors that, in the environment in which they are deployed, are able to continuously collect diverse data related for instance to the temperature, humidity, CO_2 etc. The goal of our paper is to show how, in the Qarnot platform, one can develop distributed and in-situ workflows for *smart-homes problems*. For this, we consider a typical problem: the detection of alarm sounds. Our paper introduces a new orchestration system for in-situ workflows, in the Qarnot platform. The specificity of this system is to be based on the software development kit and the scheduler of the Qarnot platform. We also consider a general parallel framework for training alarm sound classifiers and decline an implementation that makes use of our orchestrator. The implemented framework is next evaluated on different aspects including: the accuracy (of the resulting classifiers), the granularity of parallelism and the scalability of the implementation. The results we obtain are encouraging, they comfort the idea that the Qarnot model is certainly one of the most effective platforms for distributed and in-situ machine learning.**

***Keywords*-in-situ machine learning; distributed machine-learning; alarm sounds detection; performance evaluation**

I. Introduction

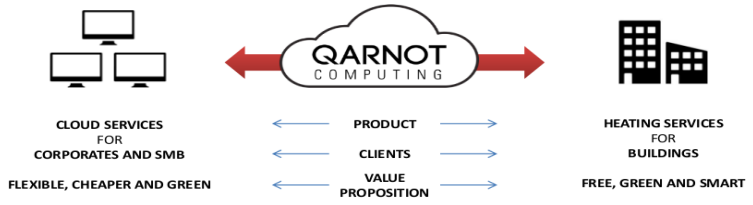
The question of the adequate architecture for the Internet of Things (IoT) has always been a main concern. At the first age of the IoT revolution, this questioning led to the development of huge datacenters that integrate machine learning and Big Data systems. However, this initial model showed several important limitations on privacy, Internet congestion, response time [1]. As an alternative, several works propose to decentralize this original IoT model by deporting the IoT logic of traditional clouds into edge clouds or embedded systems that serve for computing the intelligence of a smart environment (home, building, city etc.). For machine learning, this direction promotes *in-situ* solutions where one of the main challenge is to deploy complex machine learning workloads into *modest* computing platforms available in homes, offices etc. In this paper, we are interested in building distributed machine learning systems that are able to locally train and improve models from homes or buildings in which their sensory data is issued. Our paper discusses such a system in considering the utility computing model introduced by Qarnot computing.

Qarnot computing [1] promotes a cloud computing model where heating, computing, storage and networking is provided from a single infrastructure: a network of geo-distributed servers deployed in homes, offices and buildings. The Qarnot model is based on two main products. The first is a new model of servers (Q.rads) in which the cooling system is replaced by a heat diffusion system, regulated by a thermostat. Each Q.rad (See Figure 1) embeds several processors in addition to sensors related to humidity, CO_2, presence etc. The second Qarnot product is a new middleware (Q.ware) for on-demand computing. Q.ware manages two types of requests: requests for processing cloud services and requests for heating. Its goal is to deploy and adjust the run of cloud workloads to meet the heat demand on Q.rads. For more details about the Qarnot model, we refer the interested reader to [2].

Considering data or processes, the Qarnot model is appropriate for in-situ machine learning. Indeed, in the data viewpoint, its sensing capacities can be used to create local datasets for machine problems to solve at the scale of a home or building. The Qarnot platform supports a data service (QaIoT) that collects sensory data in Q.rads and makes them available to programs designed for Q.rads. In the viewpoint of processes, the distributed learning processes can be performed in the home or building for which we want to build a learning system. The Qarnot platform includes a software development kit (SDK) for building local orchestrations of processes in homes, buildings or cities. The global objective of this paper is to show how with these two tools (QaIoT and the Qarnot SDK), one can develop complex in-situ machine learning workflows for problems related to smart-homes and buildings. We focus on the construction of alarm sound classifiers. Indeed, the behavior of an alarm sound classifier will be impacted by the difference between the background

[1] www.qarnot-computing.com

(a) A Q.rad



(b) The roles of the Q.ware

Figure 1: The Qarnot model

noises, used in its construction, and the natural sound flow in which it operates.

Our paper makes three important contributions. Firstly, we introduce a general orchestration system for implementing in-situ distributed learning frameworks and in particular, a solution for the alarm sounds detection, with the Qarnot SDK. The orchestration system includes an API and a scheduler. The API of the orchestration system is based on abstractions close to the one used in GraphLab [3] (data model and update functions) and the concept of *Processors* used in the `madmom` library [4]. However, the abstractions are adjusted to fit with the object oriented interface of the Qarnot SDK. Differently from the Qarnot SDK, the orchestrator API includes simple constructs for the description of flows. The alarm detection framework we use for validating the orchestrator is based on energy, spectral and Mel-frequency cepstral features [5], [6] and a Pareto-selection method for choosing the best classifiers. We consider three techniques for building classifiers: the support vector machine (SVM), the logistic regression and the K-Nearest Neighbour (KNN). Our second contribution is to propose a possible parallel implementation for training the alarm framework based on the orchestrator. Finally (last contribution), we explore this implementation and evaluate its accuracy, false and true positive rate and runtime performance. We discuss the best granularity (from coarse-grained to fine-grained) and the scalability of the proposed implementation.

The rest of the paper is organized as follows. In Section II, we present the related works. In Section III, we discuss of the acoustic alarm detection framework that we consider. In Section IV, we describe the key services that the Qarnot platform offers for in-situ machine learning. In Section V, we explain how we implement the training process of the framework that we introduced. A performance evaluation is done in Section VI and we conclude in Section VII.

II. Related work

Our contribution is to put into perspective with the recent advances that led to the profusion of tools for parallel machine learning. Our work is also related to prior contributions in machine learning for acoustic alarm detection.

Regarding recent parallel machine learning tools, we propose to distinguish between three main trends. The first trend includes the development of orchestrators for Big Data systems that are based on Hadoop [2] or related systems. In these solutions, the parallelism is generally formulated within the Map-reduce paradigm [7]. In the second trend, the restrictions of the Map-reduce paradigm are overcome using more general graph-based abstractions like the Directed Acyclic Graphs (DAG) (solutions that are related to the distributed GraphLab [3]). Finally the last trend is related to the development of parallel systems for deep learning algorithms in which the key novelty is to exploit the parallelism at the GPU level [8]. Our proposition is neither based on Map-reduce, nor on GPUs. As already said, the orchestrator we introduce is based on an upper layer abstraction close to the one we can find in GraphLab. However, our conceptualization manipulates object oriented concepts of the Qarnot SDK. In addition, our solution is specially designed for in-situ learning.

The field of supervised machine learning for audio classification (that includes alarm sound detection) is well-established. There is a large consensus on the class of features that are meaningful in this context [9], [10] and some classification methods like the logistic regression has been successfully applied in several cases. It was also showed that for sound detection systems to work in the real-world, it is important to train the classifiers with background noises that are representative of the environment in which the classifiers will operate [11]. In other words, *in-situ* and context-aware learning is welcome on the problem.

Few works however dealt with the in-situ perspective of this work (to collect and process data locally). The closer work we found is the Auditeur system [12]. The system proposes a context-aware solution for acoustic events detection on smartphones. The Auditeur system is composed of: a phone client that captures, processes and can run a *classification plan* and a cloud service that, based on audio and contextual information sent by a phone, could generate an appropriate classification plan to be followed at the smartphone level. We did not investigate whether or not the workflow we propose could be applied to any acoustic event. The main difference between our work and Auditeur is that part of the processing could be done remotely with

[2] http://hadoop.apache.org/

Auditeur while we propose to collect and process data in a same environment, which is more interesting for real-time and security.

III. Training framework for alarm sound detection

A. General problem

We assume a home with a fire-alarm and a set of Q.rads. Each Q.rad embeds two microphones and can record the input audio stream. We also assume that the sound is recorded at a fixed [3] sample rate in a wav file. *The goal is to build a detection system which, given any wav file of T samples, could state whether or not it contains an alarm sound.*

The system to build has 3 subsystems: a data collection system, a training system and a decision system. The data collection system creates the dataset of wav files that will be used for training and validation. From the training dataset, the training system is able to generate a subset of dominant classifiers. Assuming we have classifiers that are trained for each input audio, the decision system states whether or not the audio contains a fire-alarm sound.

Our paper mainly focuses on the training system. However, we will shortly discuss the data collection system in order to explain how we create training dataset in-situ.

B. Data collection

The challenging question here is to make sure that the data used to build the classifiers are representative of the environment in which the Q.rads are. Our proposition is to create the training and testing datasets in mixing known alarm sounds with background sounds of the environment for which we are building a classifier. More precisely, we assume that we have samples of the alarm sounds, we want to recognize. We also assume samples of significant sounds like a baby cry or a music. In these audio records (alarm and significant sounds), there is no background noise. The goal of the data collection system is to locally create from these sounds two datasets for training and evaluation where the background noises of the environment are introduced. This is summarized in Figure 2. A challenge at this stage is to set-up some programmability rules for recording background noises. We must also decide on the date at which we consider that we have enough data for training. But due to space restriction, we will not discuss these details in this paper. However we will explain in Section IV, the system architecture that Qarnot proposes for data collection.

C. Training system

In Figure 3, we illustrate the training framework we consider in our problem. The representation follows the BPMN notations (An empty lozenge is a join and a lozenge with a plus is a fork...).

[3] we use $44100Hz$

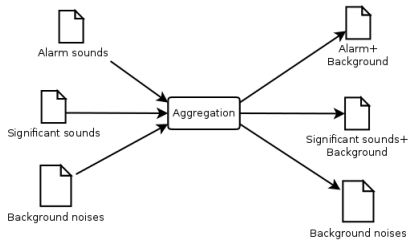


Figure 2: Automatic generation of the dataset: we mix alarm and significant sounds with the local background noises.

The framework is based on 5 activities and subprocesses that we discuss below.

- **PFE:** Given a set of wav files, the goal here is to extract the acoustic features of the wav files. PFE is a subprocess consisting of a set of parallel activities where the acoustic features are separately computed from each dataset file.
- **GSIN:** The framework implements a grid search process whose goal is to find the best classifiers depending on the parameters we use. GSIN consists in the initialization of this grid search. The goal is to generate the possible configurations we will explore. A configuration here includes: parameters related to features (discussed further) and hyper-parameters related to classification techniques (SVM, KNN ..).
- **LKOCV:** Given an input configuration (generated in GSIN), this subprocess starts with the run of a time integration method whose goal is to refine the quality of information which training acoustic features provide by aggregating their values in time (see VI-A and [10]). With the refined features, the next activity in the subprocess consists in building the classifier corresponding to the current configuration with a Leave-k-out cross-validation.
- **PSEL:** From a list of classifiers whose False and True Positive rates (FPR, TPR) are defined (generated in LKOCV or EVAL), we select the ones that are in the Pareto front where we minimize the FPR and maximize the TPR.
- **EVAL:** Assuming a list of classifiers and the input features of the testing dataset, we evaluate here the classifiers on the testing features. The TPR and FPR of each classifier are returned.
- **TRAIN:** We train the selected classifiers with all data from training and testing sets.

It is important to notice that some activities in the proposed framework are parallel and distributed. For instance, at the beginning, the PFE activities could be done in parallel. The LKOCV activities also are performed in parallel (each parallel instance deals with a configuration). In the next, we will discuss of the component of the Qarnot architecture for its implementation.

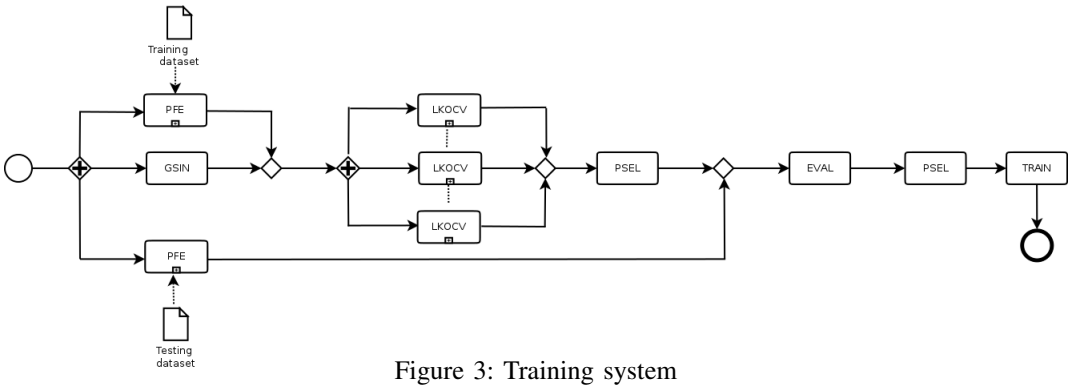


Figure 3: Training system

IV. Architecture and programming model

In the viewpoint of data, the network of Q.rads at the scale of a city or building is a composition of clusters, each associated with a QaIoT server. A QaIoT server aggregates and manages the sensory data issued from the Q.rads to which it is connected. A program deployed on a connected device (or a Q.rad) can get access to these data. For this purpose, it must first register to the QaIoT server by sending an Http request where it declines: the nature of the device (Q.rad, raspberry etc.), the Id of the device, the role it intends to play (e.g: access to the audio stream of the cluster). If the request is accepted a websocket communication will be established between the device and QaIoT in order to service the data to the applicant program. In our case, the communication will consist in servicing real-time data. A view of the QaIoT data service is given in Figure 4a. It is easy to notice that the mixer of Figure 2 can be implemented as an applicant program that request the audio streams related to a home and then *mixes* the data with a set of alarms and significant sounds.

In the processing viewpoint, the Qarnot SDK is based on three main concepts: the notions of task, disks and docker image. A task is an object oriented abstraction that refers to a docker container or a set of containers to deploy and run. A task is associated with a docker image defined in the parameter *constants["DOCKER_REPO"]*. We can specify a set of input files for a task in defining a *resource disk*. Such a disk is defined in the list parameter *resources* of the task. A task has a name and can be composed by subtasks (or parallel instances). In these cases, the run of each subtask will cause the deployment of a docker image. The process run by a task is defined in assigning a command line to the parameter *constants['DOCKER_CMD']*. The execution will assume that the input files are available from the resource disk and and will generate output files in a *results disk*.

An example of script with the Qarnot SDK is proposed in Figure 4. Finally, for in-situ learning, let us notice that in the way we define the constants of a task, we can indicate a specific Q.rad or a set of Q.rads on which we want the task to run.

At this stage, we presented the framework that we intend to automate and tools of the Qarnot cloud we intend to use. In the next, we will now present the orchestration system we propose for automating the proposed framework.

V. Orchestrating the alarm detection framework

A. The orchestrator API

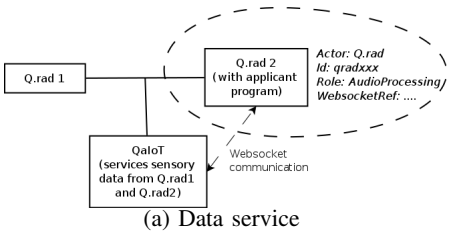
The orchestrator API relies on 2 basic concepts (implemented in `python`):

- **QWork:** This represents a work to be done during the workflow. QWorks correspond to boxes in a diagram. They have 2 main attributes: `name` which enables to differentiate the QWorks and `next_name` which enables to define the next QWork in the workflow. A QWork also has a `process` method which represents the work to do. To make it even simpler, QWork instances are callable which means that they can be used as any function.
- **QData:** This represents the inputs and outputs of QWorks. A QData instance is a dictionary with 3 keys (it can be seen as a 3 columns table and we will keep this analogy for clarity): `data` (the actual data), `sender` (the name of the QWork which sends the data) and `receiver` (the name of the QWork which receives the data).

These two concepts work together as follows (see algorithm 1): if `w` and `d` are respectively instances of QWork and QData, running `w(d)` makes `w` look for its `name` in `d`'s `receiver` column and process the corresponding row using the `process` method. In the `process` method, one can specify how to handle different `senders`.

Subclasses of QWork can then be defined to differentiate what is done locally and what is done in parallel using the Qarnot platform:

- **QFunc:** a QFunc is a QWork which processes a particular function locally.
- **QTask:** QTask inherits from QWork and Task (from the qarnot SDK). The QTask class has two particularities: the first one is that its input must be a disk (which will be added to the resources) or a serializable object (which will be put in a resource disk). This is done with



(a) Data service

```python
import qarnot
conn = qarnot.Connection('samples.conf')
task = conn.create_task('sample2-files', 'docker-batch', 1)
input_disk = conn.create_disk('sample2-files-input-resource')
input_disk.add_file('input/lorem.txt')
task.resources.append(input_disk)
task.constants['DOCKER_CMD'] =
        'sh -c "cat lorem.txt | tr [:lower:] [:upper:] > LOREM.TXT"'
task.run()
```

(b) Example of scripts with the Qarnot SDK

Figure 4: Data service and Qarnot SDK example.

a method called `add_input_to_resources`. The second one is that there are two ways to run a QTask instance: `process` method (or calling the instance) which will submit the task and wait for it to be over, and the `submit` method which does not wait for the task to be over (which is useful for parallel tasks, see below). The output of a QTask is a results disk (as defined in the Qarnot SDK).

With these three tools (QData, QFunc, QTask) one could create basic parallel workflows composed of functions and tasks. To go a step further in the abstraction, we introduce two other subclasses of QWork that represent higher level concepts:

- **QParallelWork:** A QParallelWork instance has a list of QWorks (except QTasks) and a list of QTasks and runs all of them in parallel using different threads (here QTasks and other QWorks were separated because we use the `submit` method for QTasks which does not exist in the others).
- **QWorkflow:** This is the highest concept since it represents the workflow itself. A QWorkflow task has a list of QWorks and runs them in a *greedy* way which means that it only runs a QWork when its inputs are available (i.e when the previous QWork in the list stops).

To sum up, there are 4 subclasses of QWork: QFunc for basic functions, QTasks for tasks, QParallelWork for parallel processing and QWorkflow to combine them all. QData makes the link between them. By combining all these concepts, complex workflows can be implemented. Note that a QWorkflow can be put in a QParallelWork to run multiple workflows simultaneously. Algorithms 1, 2, 3, 4 sum up in pseudo-code the ways these concepts handle a given input. A QWork will call its `process` method. A QFunc's `process` method calls the function. A QTask will add the inputs to its resources and submit the task to the Q.Ware scheduler. A QParallelWork will submit all QTasks and run all other QWorks in different threads and then wait for all of them to finish. A QWorkflow will run all its QWorks.

Note that all QWorks which are not QTasks are processed locally, where the global QWorkflow is running, and all QTasks are submitted to the Q.Ware scheduler. In is important to precise that QTasks can be constrained to run in a specific Q.rad. Hence, our orchestrator manages two levels of scheduling: a local level where as soon as a Qwork can be done, it is processed or submitted to the Q.ware and the Q.ware level (external) where jobs are run according to a FIFO (First In First Out) policy. Thus a QWorkflow can be strictly processed in a home.

In all the algorithms presented, QWorks need a QData as input, therefore if to handle multiple input one can either get inputs as different QData rows and differentiate them using the `sender` (see for example `f211` in figure 5) or put a dictionary in the QData `data` column (see for example QCreateDisk below).

Algorithm 1 QWork call method

Require: `d` a QData instance
Ensure: returns `d` without my input, with my output
  find my name in `d["receiver"]`, remove the corresponding rows from `d` and put them in `d'`
  out ← `process(d')`
  add out to `d`
  return `d`

Algorithm 2 QTask submit method

Require: `d` a QData instance
Ensure: adds the input to resources and submits the task
  find my name in `d["receiver"]`, remove the corresponding rows from `d` and put them in `d'`
  `add_input_to_resources(d')`
  submit the task

Algorithm 3 QParallelWork call method

Require: `d` a QData instance
Ensure: returns `d` without my QWorks and QTasks inputs and with their outputs
  for `w` in my list of QWorks and my list of QTasks **do**
    find `w.name` in `d["receiver"]`, remove the corresponding rows from `d` and put them in `d'`
  end for
  for `w` in in my list of QWorks **do**
    create a thread to run $w(d_2)$
  end for
  for `t` in in my list of QTasks **do**
    `t.submit()`
  end for
  wait for every thread and every task to end
  add all outputs to `d`
  return `d`

In the next subsections, we present a simple example and an implementation of the framework using the orchestrator.

B. A simple orchestration example

We present here a simple example using the orchestration, the workflow consists in basic operations on numbers but shows how to use the different concepts to go from a

Algorithm 4 QWorkflow process method

Require: `d` a QData instance
Ensure: runs the Qworks sequentially
  `in` ← `d`
  for `w` in my list of QWorks **do**
    `in = w(in)`
  end for
  return `in`

diagram to the code. The diagram, the `python` code and a part of the `stdout` are presented in figure 5

C. Implementation of the framework

To implement the framework of Figure 3 with the orchestrator, we defined subclasses of QTask and QFunc for the specific processes.

- **QCreateDisk:** (inherits from QFunc, processed locally and sequentially). At the beginning of the framework of Figure 3, this Qwork creates the disks where training and testing data will be uploaded [4]. The particular attributes of this Qwork are:
  - The qarnot connection (the concept is specific to the Qarnot SDK; a connection will define the building and home where the task will be scheduled)
  - The directories of the files to put in the disk (this can be given as a disk)
  - The labels of the data (alarm or not alarm)
- **QExtractFeaturesTask:** (inherits from QTask, processed on Q.rads and in parallel). This Qwork corresponds to PFE in figure 3. Given a disk of .wav file, a list of frame sizes and frame steps, this Qwork separates the files in a number of portions and extracts features of each portion in parallel. The output is a disk containing .csv files with the features. Its particular attributes (in addition to those inherited from QTask) are
  - The frame sizes and steps for extraction
  - The remote root to find data
- **QLKOCVTask:** (inherits from QTask, processed on Q.rads and in parallel) This Qwork corresponds to LKOCV in figure 3. Given a disk of .csv features files, a list of parameters for grid search, this Qwork performs grid search in parallel. The process to make here consists of several parallel instances where each performs features integration (according to the parameters given) and a Leave-k-out cross validation on the dataset and writes in a file the false and true positive rates. Its particular attributes are:
  - The classification method (logistic regression, KNN, SVM)
  - The length (in samples) of the data
  - The remote root to find data
  - The grid of parameters to test (parameters for extracting features and hyperparameters, the grid is created from both types of parameters)
  - Other arguments for cross validation
- **QSelectionTask:** (inherits from QTask, processed on a Q.rad and sequentially i.e with only one parallel instance) This Qwork implements the processing done by PSEL, EVAL and PSEL in figure 3. It takes the results from QLKOCVTask and performs Pareto-selection to get the best classifiers on the training set. Then it tests these classifiers on the test set and performs Pareto-selection again to select the best classifiers. It returns the selected classifiers trained with all the dataset (train and test sets). Its particular attributes are:
  - The classification method
  - The remote root to find data (train and test)
  - Other arguments for Pareto selection
- **QDownloadResults:** (inherits from QFunc, processed locally and sequentially) This is to download the selected classifiers and the results.

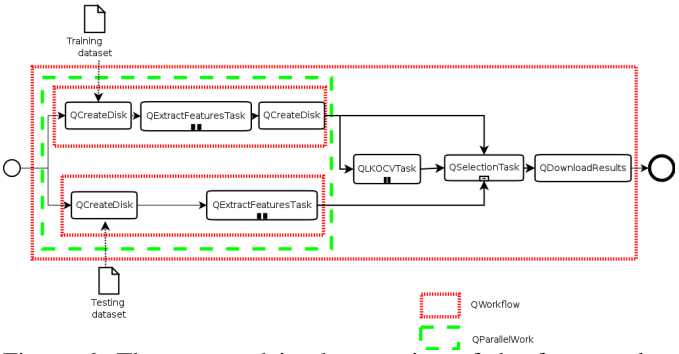


Figure 6: The proposed implementation of the framework using the orchestrator.

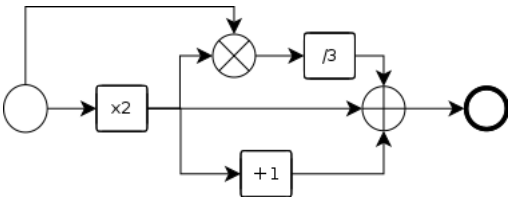
Figures 6 and 7 show the implementation proposed for the framework of Figure 3 in using the orchestrator. All basic blocs are created using the above subclasses and are combined using QWorkflow and QParallelWork. Note that we added a QCreateDisk after QExtractFeaturesTask to add the code needed for selection (this can be avoided by putting the code in the docker repository used for the tasks).

```
import selection as sel
import qarnot
import orchestration as orch
import numpy as np
LENGTH = 5*44100  # length (in samples) of audios
# set qarnot connection
params = 'samples.conf'
conn = qarnot.connection.Connection(params)
profile = 'docker-batch'
cst = {'DOCKER_REPO': "amaurydurand/my-python"}
# set dataset
train_root = 'dataset/Train'
train_directories = [train_root + '/Non_alarm',
                     train_root + '/Alarm']
train_labels = [0, 1]
remote_root_train='data_train'
test_root = 'dataset/Test'
```

[4] See Section IV for the concept of disk.

```
from orchestration import *
# define basic blocks
f1 = QFunc(func=lambda x:
           2*x.get_value('sender', 'init')['data'][0],
           name='f1',
           next_name=['f211', 'f22', 'f3'])
f211 = QFunc(func=lambda x:
             x.get_value('sender', 'f1')['data'][0]
             * x.get_value('sender', 'init')['data'][0],
             name='f211',
             next_name='f212')
f212 = QFunc(func=lambda x:
             x.get_value('sender', 'f211')['data'][0]/3.,
             name='f212',
             next_name='f3')
f22 = QFunc(func=lambda x:
             x.get_value('sender', 'f1')['data'][0]+1,
             name='f22',
             next_name='f3')
f3 = QFunc(func=lambda x:
           x.get_value('sender', 'f1')['data'][0]
           + x.get_value('sender', 'f212')['data'][0]
           + x.get_value('sender', 'f22')['data'][0],
           name='f3')
# define f21 workflow
f21 = QWorkflow(workflow=[f211, f212],
                name='f21',
                next_name='f3')
# define f2 parallel work
f2 = QParallelWork(list_of_func=[f21, f22],
                   name='f2',
                   next_name='f3')
# define global workflow
f = QWorkflow(workflow=[f1, f2, f3],
              name='workflow',
              verbose=True)
# define input
d = QData({
        'data':[2, 4],
        'sender':['init', 'init'],
        'receiver':['f1', 'f211']
    })
# run the workflow
res = f(d)
print(res['data'][0])
```

(a) Source code. The output is 14.33



(b) Diagram

```
FROM workflow: launching "f1"
FROM f1: input:
data     sender   receiver
2        init     f1
4        init     f211
FROM f1: selected input:
data     sender   receiver
2        init     f1
FROM f1: processing

FROM workflow: launching "f2"
FROM f2: selected input:
data     sender   receiver
4        init     f211
4        f1       f211
4        f1       f22

FROM f2: thread "f21" started
FROM f21: selected input:
data     sender   receiver
4        init     f211
4        f1       f211

FROM f21: launching "f211"
FROM f211: selected input:
data     sender   receiver
4        init     f211
4        f1       f211
FROM f211: processing

FROM f21: launching "f212"
FROM f212: selected input:
data     sender   receiver
16       f211     f212
FROM f212: processing
```

(c) Parts of `stdout`

Figure 5: Orchestration example

```
test_directories = [test_root + '/Non_alarm',
                    test_root + '/Alarm']
test_labels = [0, 1]
remote_root_test='data_test'

# set features parameters grid
frame_sizes = [2048,4096]
fpgrid = [{'frame_size': frame_sizes,
           'early_integration': ['stack', 'mean']}]
# set hyperparameters grid
hpgrid = {'C': [0.1,1,10]}
method = 'logreg'

# define all blocks
train_disk_fun = \
sel.QCreateDisk(name='train_disk_fun',
                next_name='extraction_train_task')
test_disk_fun = \
sel.QCreateDisk(name='test_disk_fun',
                next_name='extraction_test_task')

# common arguments for all tasks
task_args = {'connection':conn,
             'profile':profile,
             'task_constants':cst}
# common arguments for extraction
extraction_args = dict({'instancecount':20,
                        'frame_size':frame_sizes},
                       **task_args)
```

```
extraction_train_task = \
sel.QExtractFeaturesTask(name='extraction_train_task',
                         remote_root=remote_root_train
                         next_name='add_code_fun',
                         **extraction_args)
extraction_test_task = \
sel.QExtractFeaturesTask(name='extraction_test_task',
                         remote_root=remote_root_test,
                         next_name='sel_task',
                         **extraction_args)
add_code_fun = \
sel.QCreateDisk(name='add_code_fun',
                next_name=['lkocv_task', 'sel_task'])

lkocv_task = \
sel.QLKOCVTask(name="lkocv_task",
               next_name='sel_task',
               method=method,
               num_samples=LENGTH,
               features_params_grid=fpgrid,
               hyperparams_grid=hpgrid,
               remote_root=remote_root_train,
               **task_args)
sel_task = \
sel.QSelectionTask(name="sel_task",
                   next_name='download_func',
                   method=method,
                   remote_root_train=remote_root_train,
                   remote_root_test=remote_root_test,
                   **task_args)
```

```
download_func = \
sel.QDownloadResults(name='download_func',
                     next_name='',
                     method=method,
                     path=path)

# workflow for training
train_workflow = \
orch.QWorkflow([train_disk_fun, extraction_train_task,
                add_code_fun],
               name='train_workflow',
               next_name=['lkocv_task', 'sel_task'])
# workflow for testing
test_workflow = \
orch.QWorkflow([test_disk_fun, extraction_test_task],
               name='test_workflow',
               next_name='sel_task')
# parallel work for extraction
extraction_parallel = \
orch.QParallelWork([train_workflow, test_workflow],
                   name='extraction_parallel',
                   next_name='lkocv_task')
# global workflow
workflow = \
orch.QWorkflow([extraction_parallel, lkocv_task,
                sel_task, download_func],
               name='workflow', verbose=True)

# define inputs and run workflow
train_args={
    'conn': conn,
    'dirs': directories,
    'labels': labels,
    'name': "train_data1",
    'remote_root': remote_root_train,
    'dirs_root' :train_root,
}

test_args={
    'conn': conn,
    'dirs': test_directories,
    'labels': test_labels,
    'name': "test_data1",
    'remote_root': remote_root_test,
    'dirs_root' :test_root
}
iin = orch.QData({'data': [train_args, test_args],
                  'sender': ['init', 'init'],
                  'receiver': ['train_disk_fun',
                               'test_disk_fun']
              })
workflow(iin)
```

Figure 7: Source code of the proposed implementation.

VI. Experimental evaluation

The implemented framework was tested using a database built using the method presented in figure 2. The alarm and the significant sounds (among which baby crying, laughs,power drill) were taken from the internet and ambiance sounds (office and talking) are $5s$ recordings of Qarnot Computing's office. We divided the ambiance sounds for training and testing sets and mixed them with the other sounds using different signal-to-noise ratio (SNR) to build the dataset. The dataset is presented in table I.

A. A feature centered view of the implemented framework

The features we used are defined in Figure [6]. Features are extracted on K overlapping frames (called 'analysis frames') of the signal. A signal is then represented by K vectors: $(x_k)_{k\in[\![0,K-1]\!]}$ with $\forall k \in [\![0, K-1]\!], x_k \in \mathbb{R}^d$ where d is the number of features. These K vectors are then integrated

	Training	Testing
Background sounds		
office	40	20
talking	100	50
Sounds of interest		
alarm	1	1
others	10	10
Mix		
SNR (dB)	-15,-5,0,5	-20,-15,-10, -5,0,5
Total		
alarm	560	420
non alarm	5740	4270
total	6300	4690

Table I: dataset

using several possible techniques. This integration means that we consider the sequence of features $(x_{i,k})_{k\in[\![0,K-1]\!]}$ as a new signal which is cut in frames (called 'texture frames'). Then, some operations are done on these frames to aggregate the features [10]. Classification is then performed on each vector of integrated features and a vote is done at the end to combine results (see figure 8). Note that the extraction part is done by a QExtractFeaturesTask and integration by a QLKOCVTask.

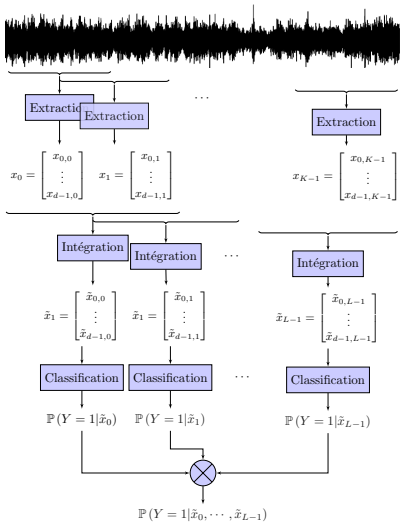


Figure 8: Features extraction and integration

The features extraction was performed using the library `pyAudioAnalysis` [6] which implements well known features for audio signal processing such as Zero crossing rate, energy, spectral centroid and spread or MFCCs. A total of $d = 34$ features were used. These features were extracted using several analysis frames sizes, integration methods ('stack' = concatenating all vectors, 'mVar' = concatenating mean and variance) and texture frames sizes. For analysis frames and textures frames we used an 50% overlap. This features are referred as 'features parameters'.

B. Experimental setup

The parameters we used for the grid search are of two sorts: features parameters (defined above) and hyperparameters (parameters of the classification method). We then ran the framework of Figure 6 to select for each classification

technique the best combination of features parameters and hyperparameters. The list of all parameters used is presented in Table II.

Features parameters	
Analysis frames size (in samples)	$2048, 4096, \cdots 65536, T$
Integration	None (if Analysis frames size = T), stack, mVar
Texture frames size (in number of analysis frames)	$4, 8, 16, 32, \cdots$, all
Hyperparameters	
Logistic regression	$C \in \{0.1, 1, 10\}$
KNN	$k \in \{5, 10, 15, \cdots, 50\}$
SVM	kernel $\in$ {linear, gaussian} γ (for gaussian kernel) $\in \{1, 4, 16, 32\}$

Table II: list of parameters

C. Classifier selection results

We generated alarm sounds classifiers (with the SVM, KNN and logistic regression) in considering the workflow of Figure 6. The classification method which gave the best results is logistic regression with a FPR on the test set of 0.02 and a TPR of 0.83 (against for example FPR = 0.0007 and TPR=0.48 for KNN and FPR=0.24 and TPR=0.8 for SVM). Figure 9 shows the learning curve for the selected classifier. This curve is encouraging since it corresponds to a high-variance situation in which increasing the size of the dataset is likely to give better results. This results shows that, used with the right classification method, the framework outputs accurate solutions.

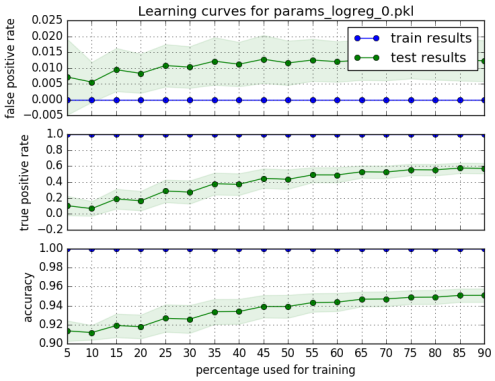


Figure 9: Learning curves for the best logistic regression classifier

D. Runtime

Here we present the runtime we obtained from runs of the workflow of Figure 6 with the logistic regression method. The grid search in this cases visited a total of 129 configurations. We did two series of experiments. The goal in the first serie was to find the best parallel granularity to run the workflow with. Indeed, QExtractFeaturesTask and QLKOCVTask can be done in parallel. A question is then to know how many parallel instances to run them with. We centered the search of the best granularity on the QLKOCVTask. Given 129 configurations, we ran this process with $5, 10, 20$ and 129 instances. For QExtractFeaturesTask, we used 10 instances. This serie of experiments was done with a number of nodes (CPUs) equal to 34.

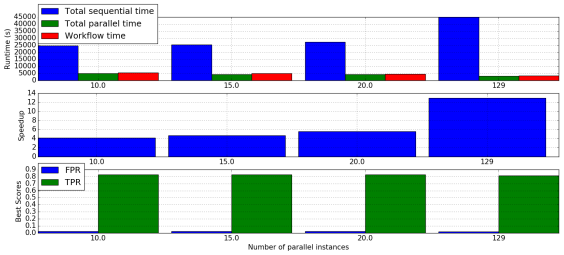
The second serie of experiment evaluates the scalability of the fine-grained solution (the number of parallel instance is equal to the number of configurations). Here we measure runtimes assuming that we want to find the best classifier from 25,50,100 and 125 configurations. In this serie, the number of nodes (CPUs) was equal to 25 .

Figure 10 shows the results for the extraction part which was ran with 10 parallel instances for the train set and 10 others for the test set (i.e 10 parallel instances for each QExtractFeaturesTask). The first bar shows the time it would have taken to run the extraction sequentially (it is divided in two: the train set and the test set). The two other bars show the time it took to run QCreateDisk and QExtractFeaturesTask for both sets. The global time of the extraction in parallel is the time of the slowest instance i.e the one for train set which gives a global speedup of 1.9. Figure 11 shows the results of granularity evaluation. The first observation is that the workflow does perform better than the sequential implementation. In subfigure 11b, are presented the individual runtime for QLKOCVTask and QSelectionTask. The main part of the workflow is QLKOCVTask and its runtime is reduced a lot by parallelism. On the contrary, the QSelectionTask would have performed better if it had been ran sequentially. The reason is simple: we implemented QSelectionTask as a sequential process which performs Pareto selection, evaluate the best classifiers on the test set and selects again. The reason we implemented it as a task is to simulate its behaviour on a Q.Rad. The difference between sequential and parallel times is due to loading times (the docker image, the data). However, this loss is not significant compared to the gain we get in QLKOCVTask. Finally, in subfigure 11a, one can see that the speedup increases with the number of parallel instances and is at its maximum (≈ 13) for the finest granularity where there are as many instances as parameters. This behaviour made us believe that this fine-grained approach is the best and this is the reason why we tested scalability with this strategy. Figure 12 shows the results of this experiment. The main conclusion to have is that speedup increases with the number of parameters to test which is encouraging for scalability. However, the best scores show that testing more parameters is not always a good solution since they attain their optimum for 75 parameters. This question is inherently linked to the grid search strategy in which we decide to test all possibilities and will not be discussed here.

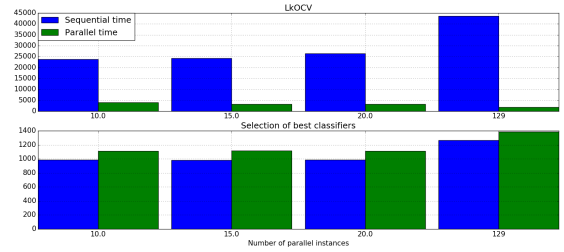
VII. Conclusion

The Qarnot model of computing is ideal for in-situ machine learning since collecting data and selecting the best classifier for the created dataset can be done within the same home or building. This leads to an improvement in terms of privacy

Figure 10: Runtime for features extraction



(a) Global results



(b) Individual tasks results

Figure 11: Runtime for coarse-grained implementation

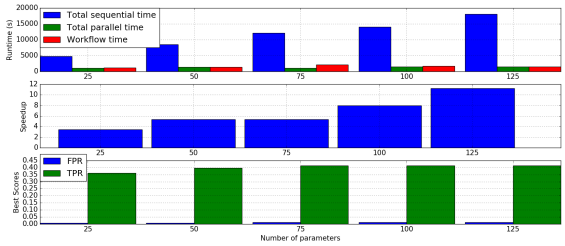
and response time. We introduced an orchestration solution to ease the creation of complex in-situ workflows in the Qarnot platform. We also show how to use this orchestrator in the concrete case of the creation of classifiers for alarm sound detection. The experimental results we obtained are encouraging: we were able to generate accurate machine learning models. However, the performance evaluation revealed that the speedup could be limited when we have a huge amount of data to process. Our main perspective is then to see how with techniques like caching, prefetching, recovery of input/output, we can improve the data management in the proposed orchestrator.

REFERENCES

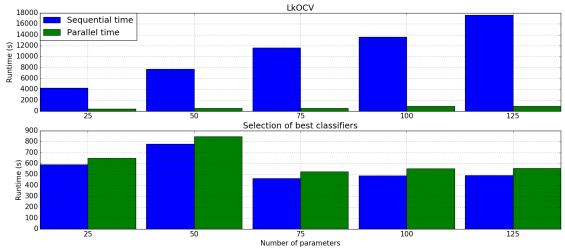
[1] M. Roelands, "Iot service platform enhancement through 'in-situ' machine learning of real-world knowledge," in *38th Annual IEEE Conference on Local Computer Networks, Sydney, Australia, October 21-24, 2013 - Workshop Proceedings*, 2013, pp. 896–903.

[2] Y. Ngoko, "Heating as a cloud-service, A position paper (industrial presentation)," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 389–401.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.



(a) Global results



(b) Individual tasks results

Figure 12: Runtime for fine-grained implementation

[4] S. Böck, F. Korzeniowski, J. Schlüter, F. Krebs, and G. Widmer, "madmom: a new python audio and music signal processing library," *CoRR*, vol. abs/1605.07008, 2016.

[5] T. Ganchev, N. Fakotakis, and G. Kokkinakis, "Comparative evaluation of various mfcc implementations on the speaker verification task," in *Proceedings of the SPECOM*, vol. 1, 2005, pp. 191–194.

[6] T. Giannakopoulos, "pyaudioanalysis: An open-source python library for audio signal analysis," *PLOS one*, vol. 10, no. 12, pp. 1–17, December 2015.

[7] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Commun. ACM*, vol. 53, pp. 72–77, 2010.

[8] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. ACM, 2009, pp. 873–880.

[9] M. Mckinney and J. Breebaart, "Features for audio and music classification," in *Proceedings of the International Symposium on Music Information Retrieval*, 2003, pp. 151–158.

[10] C. Joder, S. Essid, and G. Richard, "Temporal integration for audio classification with application to musical instrument classification," *IEEE Trans. Audio, Speech & Language Processing*, vol. 17, no. 1, pp. 174–186, 2009.

[11] J. Salamon and J. P. Bello, "Unsupervised feature learning for urban sound classification," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*, 2015, pp. 171–175.

[12] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang, "Auditeur: a mobile-cloud service platform for acoustic event detection on smartphones," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '13. New York, NY, USA: ACM, 2013, pp. 403–416.