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 envrironment in which they are deployed, could continuously collect diverse data related for instance to the temperature, humidity, CO_2 etc. The goal of our paper is to show that with the Qarnot platform, complex in-situ workflows for smart-homes and buildings could be developed. For this, we consider a typical problem that could be addressed in such environments: the detection of alarm sounds. Our paper proposes a general model for orchestrating in-situ workflows on the Qarnot platform and declines both a coarse-grained and fine-grained implementations for alarm sounds detection. We also provide an experimental evaluation of the implemented solutions where we discuss of the accuracy and the runtime of the training process. The results we obtain are encouraging, they comfort the idea that the Qarnot model is certainly one of the most effective platform for the decentralization of IoT and machine learning systems.**

***Keywords*-in-situ machine learning; decentralized IoT systems; alarm sounds detection; performance evaluation**

I. Introduction

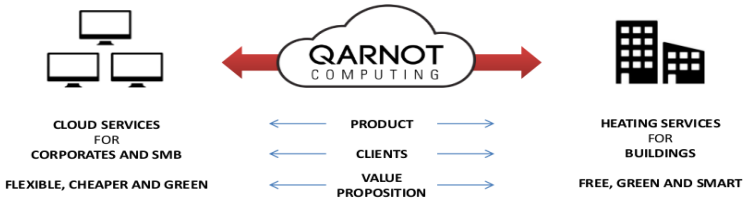
The question of the right computing architecture for the Internet of Things (IoT) has always been a main concern. At the first age of the IoT revolution, this questioning led to the development of huge datacenters that integrate machine learning and Big Data systems. However, this intial model showed 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 environnment (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 addition, for any machine learning problem, we can then consider the calibration or the creation of context-aware datasets. This paper follows this line of thought. In particular, we are convinced of the need to build decentralized machine learning systems that could locally train and improve models from homes or buildings in which their sensory data is issued. Our paper proposes a solution that follows this direction 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 workload to meet the heat demand on Q.rads. For more details about the Qarnot model, we refer the interested reader to [2].

For data or processes, the Qarnot model is very rich for in-situ machine learning. Indeed, in the data viewpoint, its sensing capacities can be used to create local datasets for any machine problem 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 make 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 that with these two tools, complex in-situ machine learning workflows could be developed for smart-homes and buildings. We focus on a problem that is interesting to address with in-situ learning: the construction

[1] www.qarnot-computing.com

(a) A Q.rad



(b) The roles of the Q.ware

Figure 1: The Qarnot model

of alarm sound classifiers. The behavior of an alarm sound classifier will be related to the background noises that it was trained with. Thus, depending on the environment in which we are, it is important to have the right background noises.

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 proposed orchestrator assumes an upper level abstraction 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 abstraction is adjusted to fit with the object oriented interface of the Qarnot SDK. 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, we explore this implementation and evaluate its accuracy, false and true positive rate and runtime performance. We compare two load balancing strategies : a coarse grained implementation in which the grid search is uniformly divided between the available nodes and a fined-grained one in which each combination of parameters is a parallel instance handled by the Q.ware scheduler. The Q.ware scheduler puts the instances in a FIFO queue and runs one when a node is available. This evaluation clearly states how we can decide on the most appropriate 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 perpective 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 [8]. In the second trend, the restriction of the Map-reduce paradim are overcome using more general graph-based abstractions like the DAGs (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 [9]. Our proposition is neither based on Map-reduce, nor on GPUs. As already said, the orchestor 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.

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 [10], [11] 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 that the classifiers be trained with background noises that are representative of the environment in which the classifiers will operate [12]. In other words, *in-situ* and context-aware learning is welcome on the problem.

On this point, the work of Diane Watson et al. [13] shows that music recommender system can be improved if we account on the context in which the listener is. The work that we found closely related to ours is the Auditeur system [15]. The system proposes a context-aware solution for acoustic events detection on smartphones. The Auditeur system is

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

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. But the main difference between our work and Auditeur is that 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 sample rate of $F_s = 44100Hz$ in a wav file. *The goal is to build a detection system that given any wav file of T samples ($\frac{T}{F_s}$ seconds) 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 will mainly focus 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 use to build the classifiers are representative of the environment in which the Q.rads are. We propose to consider two main approaches. The first is to interact with the user in the home in order to record audio that include an alarm sound and those that do not. The data collection and training systems could be seen as a *machine teaching system*. In this philosophy, let us notice that the expertise and level of cooperation of the user will decide on the quality of the generated classifiers. The second approach (that we will prefer) consists in automatic generation of the dataset. For this, we suppose that we have a basis of alarm sounds that are the one we want to recognize. We also assume a basis of significative sounds like a baby cry or a music. In these audio records (alarm and significative sounds), there is no background noises. The goal of the data collection system is to create from this basis two datasets for training and evaluation where the background noises of the environment is introduced. This is summarized in Figure 2. A challenge at this stage is to fix some programmability rules for recording background noises. We must also decide on the date at which we consider that a set of classifiers could be generated for exploitation. 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 in-situ data collection and processing.

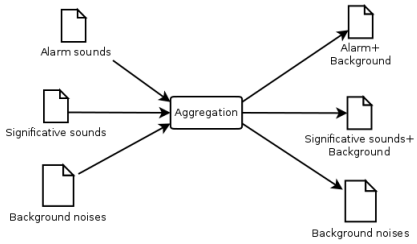


Figure 2: Automatic generation of the dataset: we mix alarm and significative sounds with the local background noises.

C. Training system

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

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 use in the search of the best classifiers.
- **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 training acoustic features provide by aggregating their values in time (see VI-A and [11]). With the refined features, the next activity in the subprocess consists in evaluating 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

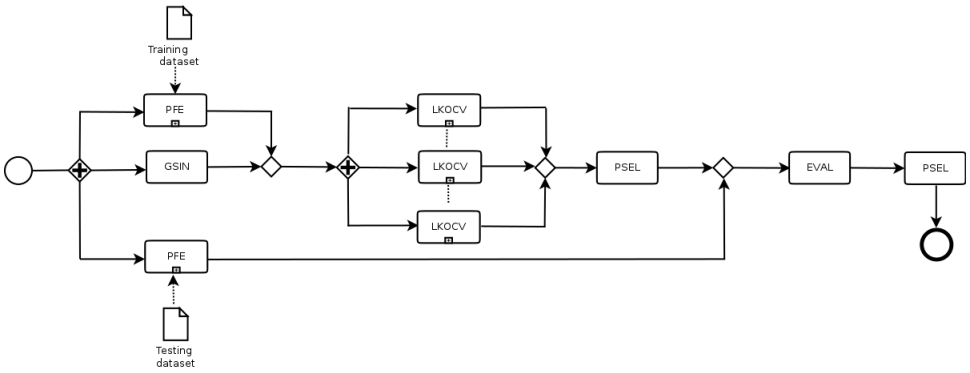


Figure 3: Training system

classifiers on the testing features. The TPR and FPR of each classifier are returned.

The proposed framework is parallel and distributed. In the next, we will discuss of the component of the Qarnot architecture for its implementation.

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 distribute the data to service the data to the applicant program. Let us suppose that this communication can consist of servicing real-time data to the applicant program. An illustration is provided in Figure 4(a). 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 notion of task, the notion of disk and the docker image. A task is an object oriented abstraction that natively refers to a docker or a set of docker 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. The input disk is defined in the list parameter *resources*. A task has a name and can be composed by subtasks (or frames). 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 disks defined in *resources*, and will generate a result disk if the task produces files.

An example of script with the Qarnot SDK is proposed in Figure 4(b). Finally, for in-situ learning, let us notice that in using the constants of a task, we can constraint its execution to be done only on a Q.rad or a set of Q.rad (and therefore a home or a building).

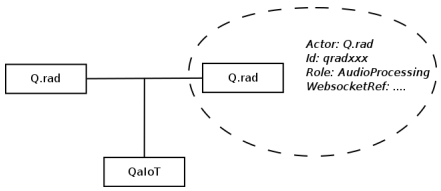
V. Orchestrating the alarm detection framework

A. The orchestrator

The orchestration method implemented for the framework 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 `nextname` 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. QData correspond to arrows in a diagram. 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`.



(a) Data service

```
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.

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 given when instantiating the class
- **QTask:** a QTask is a QWork and a 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 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 its results disk.

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 created 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).
- **QWorklow:** 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 multithreading 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 QWorkflows 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 submited to the Q.Ware scheduler for parallel processing. Moreover, the QTasks can be constrained to run the Q.rads of a house, a building on several buildings. In-Situ machine learning can then be done by running the global QWorkflow on one Q.rad of the house and constraining the QTasks to all the house's Q.rads.

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

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

Algorithm 3 QParallelWork call method

Require: `d` a QData instance
Ensure: returns `d` without my QWorks and QTasks inputs and with the 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`

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`

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 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 with the orchestrator, we defined subclasses of QTask and QFunc for the specific processes.

- **QCreateDisk:** (inherits from QFunc) This function aims to create a disk with the dataset and all the code needed for selection. Its particular inputs are (here we will use a dictionary for multiple inputs)
  - The qarnot connection
  - The directories of the files to put in the disk
  - The labels of the data (one label per directory)
  - A disk (if a disk in given, this function only adds the code for selection)
  - The local root of directories and the one the user wants them to be in the disk
- **QExtractFeaturesTask:** (inherits from QTask) Given a disk of .wav file, a list of frame sizes and frame steps, this task 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) Given a disk of .csv features files, a list of parameters for grid search, this task aims to perform grid search in parallel. Each parallel instance performs 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
  - 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 during the instanciation)
  - Other arguments for cross validation
- **QSelectionTask:** (inherits from QTask) This task 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. 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) This is to download the selected classifiers and the results.

Figures 6 and 7 show a possible implementation of the framework using the orchestrator. Note that we added a QCreateDisk after QExtractFeaturesTask, this is to get the code for selection in the results disk given by the QExtractFeaturesTask. This could be avoided by putting the code in the docker repository used for the tasks.

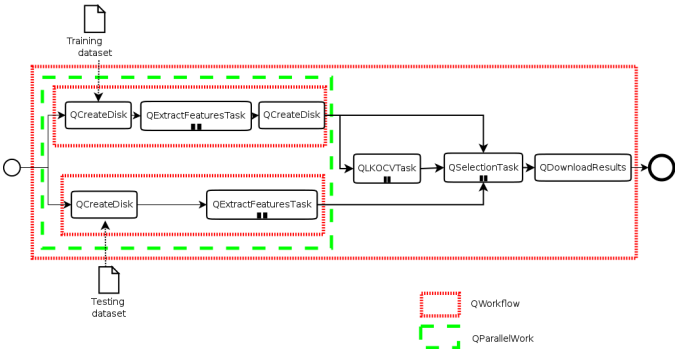
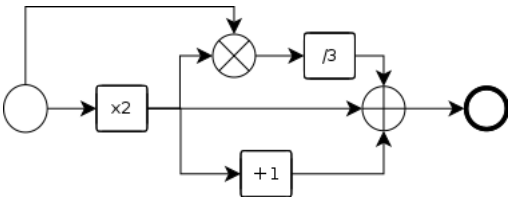


Figure 6: A possible implementation of the framework using the orchestrator.

```
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

```
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'
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 for alarm sounds detection which consists in a binary audio classification problem. Such a problem can be solved following a classic two steps approach : features extraction and classification. After explaining briefly the first step, we present here the results given by the framework. Finally, we compare the work-sharing-like and the work-stealing-like ways to run the framework on the Qarnot infrastructure. We will denote by T the length (in samples) of the signal which is recorded at a sample rate of $44100Hz$ and by $Y \in \{0, 1\}$ the target which is equal to 1 if the signal is an alarm sound and 0 otherwise.

A. Features extraction

Features are extracted on K overlapping frames (called 'analysis frames') of the signal (see the first part of figure 8). 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 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 de features [11]. Classification is then performed on each vector of integrated features and a vote is done at the end to combine results (see figure 8).

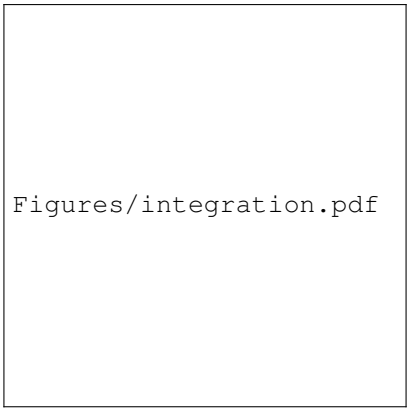


Figure 8: Features extraction and integration

B. Experimental setup

1) Dataset: The dataset used for experiments was built using the method presented in figure 2. The alarm and the significative sounds (among which baby crying, cough, laughters, hammer, power drill) were taken from the internet and ambiance sounds (office sounds and talking crowd) are $5s$ recordings of Qarnot Computing's office. We devided

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 division of the dataset is presented in table I.

	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

2) Parameters: 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' = contatenating all vectores, 'mean' = taking the mean, 'mVar' = concatenating mean and variance).and texture frames sizes. For analysis frames and textures frames we used an 50% overlap. This features are refered as 'features parameters'.
For the classification model, we used different techniques (SVM, logistic regression and KNN) with different hyperparameters possibilities, we then ran the framework 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, stack, mean, mVar
Texture frames size (in number of analysis frames)	$4, 8, 16, 32, \cdots$, all
Hyperparameters	
Logistic regression	$C \in \{0.01, 0.1, 1, 10\}$
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

D. Runtimes

To compute the runtimes, we used logistic regression and tested less parameters : for features parameters we used the same analysis and texture frame sizes and used no integration if the analysis frame size i sT and 'stack' of 'mVar' for the others. For hyperparameters we used $C \in \{0.1, 1, 10\}$. The total number of combinations to test was 129. We ran two different experiments : The first one is the coarse-grained implementation : the number of parameters is fixed (129) and each parallel instance performs a portion of the tests. We ran this experiment for growing numbers of instances $10, 5, 20$ and 129. The experiments were done with a number of nodes equal to 34 which mean that for the last one (129 instances) the instances were put in a FIFO queue by the Q.ware scheduler and ran when nodes were available. The second one is the fine-grained implementation : the number of parallel instance is equal to the number of parameters and we increased the number of parameters (25,50,100 ans 125).
We also divided the workflow in two parts : the features extraction and the cross-validation and classifiers selection since the first part is independent to the parameters (and it is therefore not relevant to run it several times). The second part was ran using the two approaches presented. Figure 9 show the results for the extraction part which was ran with 10 parallel instance for the train set and 10 others for the test set. 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 tests. The global time of the extraction in parallel is the time of the slowest thread i.e the one for train set which gives a global speedup of 1.9. Figures 10 and 11 show the results for both implementations of the second part.

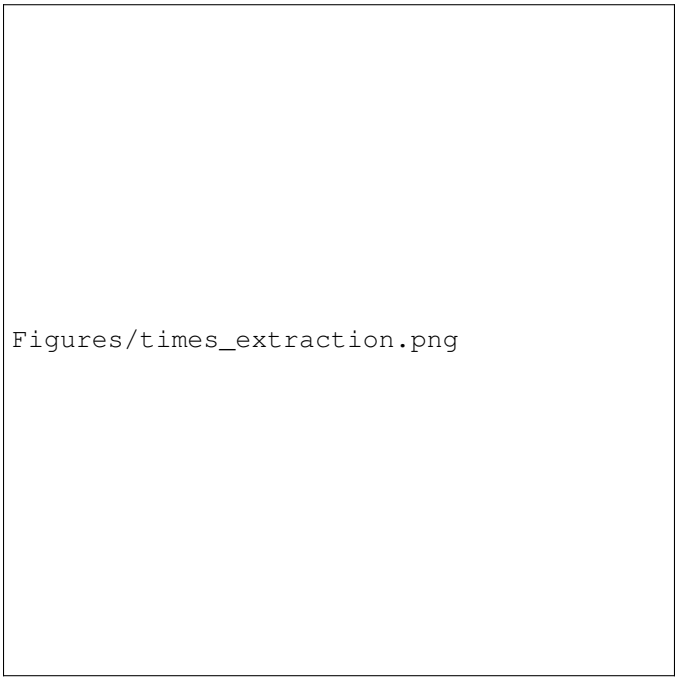


Figure 9: Runtimes for features extraction

VII. Conclusion

[Todo: enrich the monitoring process of the 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.

[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. [Online]. Available: http://arxiv.org/abs/1605.07008

[5] S. B. Davis and P. Mermelstein, "Readings in speech recognition," A. Waibel and K.-F. Lee, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences, pp. 65–74.

[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] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[8] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[9] 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. New York, NY, USA: ACM, 2009, pp. 873–880.

[10] 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.

[11] 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.

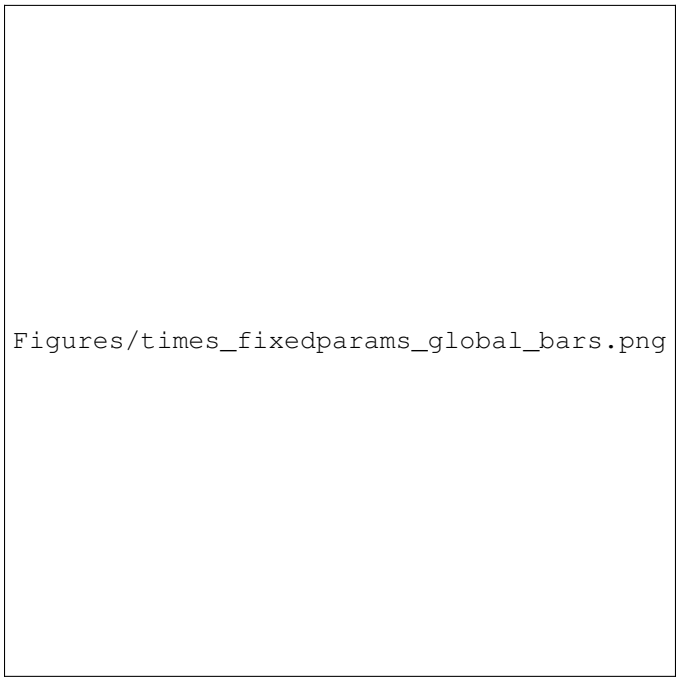
[12] J. Salamon and J. P. Bello, "Unsupervised feature learning for urban sound classification," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*, 2015, pp. 171–175.

[13] D. Watson and R. Mandryk, "An In-Situ Study of Real-Life Listening Context," in *Sound and Music Computing 2012*, Copenhagen, Denmark, 2012, pp. 11–16.

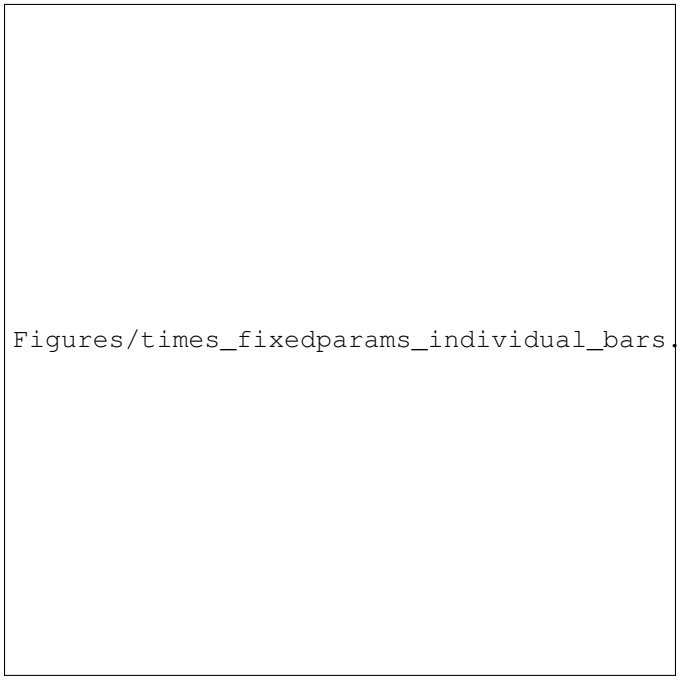
[14] S. Chu, S. Narayanan, and C.-C. J. Kuo, "Environmental sound recognition with time-frequency audio features," *Trans. Audio, Speech and Lang. Proc.*, vol. 17, no. 6, pp. 1142–1158, Aug. 2009.

[15] 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.

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

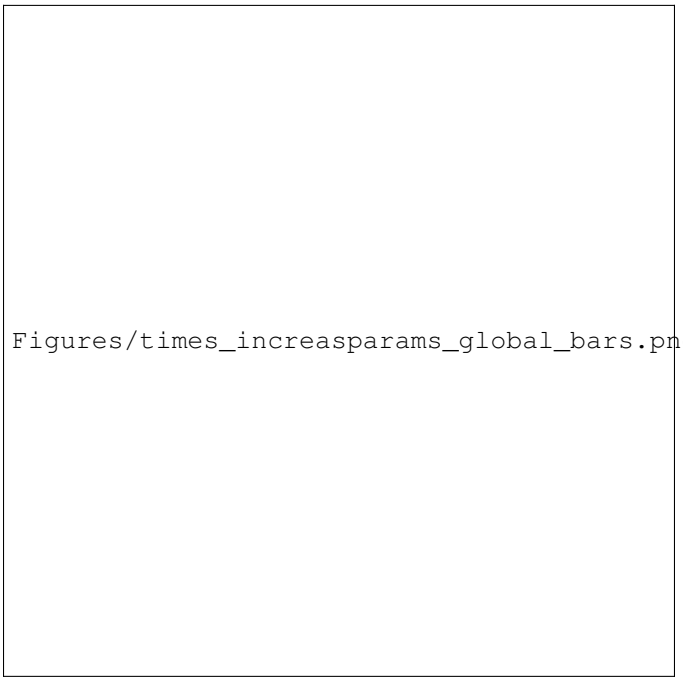


(a) Global results

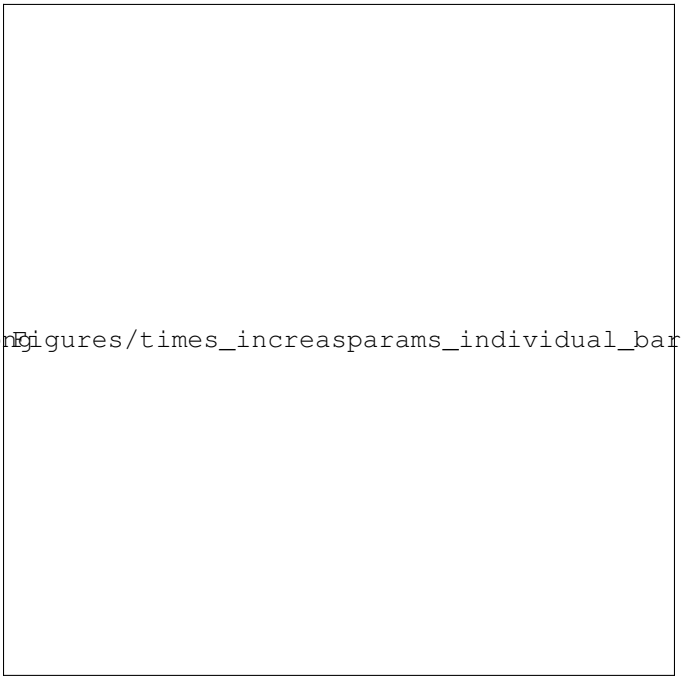


(b) Individual tasks results

Figure 10: Runtimes for coarse-grained implementation



(a) Global results



(b) Individual tasks results

Figure 11: Runtimes for fine-grained implementation