POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Suitability analysis of the Raspberry Pi to Run Streaming Machine Learning Algorithms

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ing. Informatica

Author: **Franck Edmond Luc Dura**

Student ID: 965266
Advisor: Prof. Emanuele Della Valle
Co-advisors: Giacomo Ziffer, Alessio Bernardo
Academic Year: 2021-22

Abstract

The advent of the Internet and information systems in recent decades has led to a dramatic increase in the number of data streams in the digital ecosystem. Therefore, machine learning applied to data streams is a significant stake but also a challenge due to the resource constraints involved. Indeed, applying machine learning directly to data streams has significant advantages in terms of flexibility and speed. It offers a range of solutions to sudden or gradual changes in the statistics of these streams, something that standard machine learning cannot do. Massive Online Analysis (MOA) is an open-source software library developed in Java, allowing these streaming machine learning algorithms to be implemented and tested. Paradoxically, embedded systems, which are by nature limited in resources, are more and more used, especially with the advent of connected objects. The Raspberry Pi is an embedded computer, or microcomputer, that offers features comparable to those of a basic desktop computer, in an extremely compact format. It is in this context that this thesis is written, which proposes to evaluate the suitability of the Raspberry Pi to run streaming machine learning algorithms on data streams, using MOA. First, the feasibility of the project was assessed with the implementation of Raspberry Pi emulators. Then, the performance of a real Raspberry Pi was evaluated, with synthetically generated data and real data. Finally, a quantized, and therefore lighter, version of MOA was produced and tested on the Raspberry Pi to evaluate the gain in execution performance in balance with the loss of precision. Thus, there are two main focuses in this thesis, the first being the evaluation of the performance of streaming machine learning algorithms, using MOA, on the Raspberry Pi, and the second being the development of an original solution that improves these results, through quantization.

Keywords: Streaming Machine Learning, Raspberry Pi, Quantization, MOA

Sommario

L'avvento d'Internet e dei sistemi informatici negli ultimi decenni hanno portato ad un aumento esponenziale del numero di flussi di dati nell'ecosistema digitale. L'apprendimento automatico applicato ai flussi di dati costituisce quindi un'importante opportunità, ma anche una sfida a causa dei vincoli di risorse. Infatti, applicare l'apprendimento automatico direttamente ai flussi di dati presenta importanti vantaggi in termini di flessibilità e velocità, e offre una serie di soluzioni a cambiamenti improvvisi o graduali nelle statistiche di questi flussi, cosa che l'apprendimento automatico standard non è in grado di fare. Massive Online Analysis (MOA) è una libreria software open source, sviluppata in Java, che permette di implementare e testare questi algoritmi di apprendimento automatico in streaming. Paradossalmente, i sistemi embedded, che per loro natura hanno risorse limitate, sono sempre più utilizzati, soprattutto con l'avvento degli oggetti connessi. Il Raspberry Pi è un computer embedded, o microcomputer, che offre funzionalità paragonabili a quelle di un computer desktop di base, in un formato estremamente compatto. È in questo contesto che si inserisce questa tesi, valutando l'idoneità del Raspberry Pi per eseguire algoritmi di apprendimento automatico in streaming su flussi di dati, utilizzando MOA. In primo luogo, è stata valutata la fattibilità del progetto con l'implementazione di un emulatore Raspberry Pi. Poi, sono state valutate le prestazioni di un Raspberry Pi reale con dati generati sinteticamente e con dati reali. Infine, una versione quantizzata, e quindi più leggera, di MOA è stata prodotta e testata su Raspberry Pi per valutare il guadagno in termini di prestazioni di esecuzione in rapporto alla perdita di precisione. In questa tesi, quindi, ci sono due obiettivi principali. Il primo è la valutazione delle prestazioni degli algoritmi di apprendimento automatico in streaming utilizzando MOA su Raspberry Pi, mentre il secondo è lo sviluppo di una soluzione innovativa che migliori questi risultati.

Parole chiave: Apprendimento Automatico in Streaming, Raspberry Pi, Quantizzazione, MOA

Contents

1 | Introduction

The technological and digital transformations of the last decades led industries and service companies to invest in and use information technologies massively. Years later, they realized the potential added value of judicious use of the huge amounts of data produced, collected, and transiting in their systems. At the same time, research and innovative industries (or start-ups) continued to deepen and develop technologies based on artificial intelligence, a field whose theory was built and developed since the 1950s, and whose field of possibilities and applications is also emerging with information technologies and the evolution of hardware. One subfield of artificial intelligence will be of particular interest to most companies, who will see it as a major strategic asset due to its unprecedented predictive capacity: machine learning. Today, the development of the Internet, coupled with the phenomena described above, led to a paradigm shift, where it is no longer a question of storing and then processing static data, which arrives in excessive quantities, but rather of considering transitory data as streams, which must be processed and analyzed as is. This is a major challenge for those who use machine learning models, which usually make use of static training and validation data sets, and which need to be re-trained each time new data are stored, requiring huge amounts of resources. The problem is amplified by the increasing use of connected objects, and thus embedded systems, which are by nature limited in resources, and to which industries wish to delegate an increasing amount of tasks, to perform computations as close as possible to the data source.

1.1. Streaming Machine Learning

This new paradigm has given rise to streaming machine learning [1], which proposes machine learning models directly adapted to data streams. Such a model processes an instance at a time, as the data stream grows, inspects it only once to make a prediction, and then gets updated incrementally. This field has its roots in machine learning and many of its models are direct adaptations of existing machine learning models. It is also possible to use them on traditional data sets, too large to be processed in one go. This new paradigm brings a gain in flexibility and speed, but also a gain in resources, since

data is only in transit, and does not need to be stored. Streaming machine learning also allows for more flexibility in addressing problems of changing data stream statistics (or concept drift) [2], that standard machine learning cannot address effectively. These are generally problems related to the change in the distribution of data over time. Its lower resource consumption also opens the field of possibilities, and suggests new applications, especially in the field of the Internet of Things, where embedded systems are limited in resources. This field is still relatively young, and there is much to explore. This thesis aims to study the ability of the Raspberry Pi, a compact system that can be embedded, to run streaming machine learning models, and thus contribute to the exploration of this field.

Massive Online Analysis [3] (or `MOA`) is an open-source software implemented in Java, which offers a range of tools and algorithms in the field of streaming machine learning. It is related to the `WEKA` project, a collection of machine learning algorithms for data mining tasks implemented in Java. Two interfaces are proposed: a graphical interface and a command line interface. All the experiments presented in this thesis were carried out using the command line interface of `MOA`. A compact version of `MOA`, called `tinyMOA`, which only includes the most common algorithms and tools for weight gain, is also used. This compact version comes from a related work [4] which is presented in Section 2.4.

1.2. Raspberry Pi

The Raspberry Pi [5] is a microcomputer, developed and designed by the Raspberry Pi Foundation, in the form of a single board with an ARM processor. In that, it can be suitable for embedded use. ARM processors [6] are Reduced Instruction Set Computer (`RISC`) processors, an important feature to be considered when an emulated version is implemented on a host system based on an `x86` Complex Instruction Set Computer (`CISC`) processor, widely spread among desktop computers. First released in 2012, it benefited in the course of its evolution from the increased capacity to build smaller and more powerful components, until it can offer in its latest version, the `Raspberry Pi 4b`, features close to those of some personal computers. In short, up to 8GB of RAM memory, a cortex A72 ARM-v8 64-bit processor, and physical ports such as micro-HDMI, or ethernet, as shown in Figure 1.1.

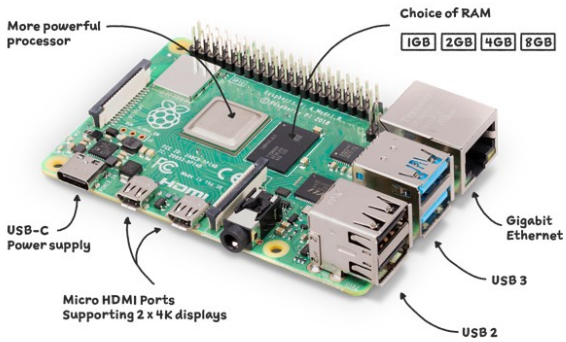

Figure 1.1: Raspberry Pi 4b, src. [7]

1.3. Contributions of the thesis

This thesis aims to :

- Demonstrate through Raspberry Pi emulations that running streaming machine learning models, including both learning and evaluation, on this type of system is possible.
- Define and carry out a set of experiments with the Massive Online Analysis (or `MOA`) software to evaluate the performance of the physical Raspberry Pi.
- Produce a lighter version, therefore more suitable for embedded systems, of `MOA` via the quantization process.
- Evaluate the performance gains of this quantized version, and put them in perspective with the induced accuracy losses.

1.4. Thesis outline

Following this introductory chapter, Chapter 2 sets the context and presents the state of the art of the topics of interest in this thesis as well as public works related to the topics addressed in this thesis. More specifically, the state of the art of streaming machine learning is discussed, and details on edge computing and quantization are provided. Chapter 3 formulates the problem and the approach adopted to answer it. It describes the different steps and details the experiments that were conducted. Chapter 4, which is more technical, details the different implementations, scripts and algorithms implemented or used throughout the thesis. Finally, Chapter 5 presents the results of the experiments detailed previously. The last chapter, Chapter 6, concludes this thesis and discusses possible developments to continue to deepen the topic.

2 | State of the Art and Related Work

This chapter is divided into 4 sections. The first Section 2.1, presents the basic concepts of streaming machine learning, starting by explaining the initial approach of standard machine learning, then specifying the important concepts and components that constitute the current state of the art. Section 2.2 presents the two main current paradigms related to embedded systems used to perform complex computational tasks, such as machine learning. Section 2.3.1 presents the concept of tiny machine learning, and more specifically of quantization, which was applied in this thesis on `tinyMOA` to lighten it. Finally, Section 2.4 gathers the different research works that were published around the topics of this thesis, i.e., streaming machine learning on embedded systems.

2.1. Streaming Machine Learning

Streaming machine learning is a direct adaptation of standard machine learning. It mainly includes the same families of models, adapted to streaming use. There exist many uses for streaming data processing, such as sensor data processing, received at a certain frequency, an anomaly detection system that monitors a data stream and that needs to be up to date at any given time, or a recommendation system evolving in real-time. Adapting models is not enough, this paradigm shift requires redefining the entire framework, including for example, performance evaluation methods or data sampling methods such as those traditionally used for bootstrap aggregating [8] (or bagging). In addition to this redefinition part, there are new methods made possible by the nature of streams, such as those related to concept drift detection. These are mainly methods that use sliding windows, statistics, and detection conditions directly on the fly.

2.1.1. Machine Learning Approach

Machine learning [9] is a subfield of artificial intelligence, which has its roots in several scientific fields, such as statistics, computer science, neurobiology, and logic. It is a set of methods which seek to adjust their parameters optimally, for a given problem, thanks to a set of input data. The dataset is usually divided into three parts, a training set, a validation set, and a test set. The training set is used to adjust the model parameters, while the validation set is only used to evaluate the model performance during training. The test set is used only once, at the end of the learning process, and is not used to adjust the model parameters or hyperparameters in any way. Machine learning tasks are generally classified into two main categories, supervised learning and unsupervised learning. Note that is now possible to define a third, intermediate category, named semi-supervised learning, which combines the first two. Each category and sub-category of tasks has its own set of models, which are adapted to the nature of the task.

Supervised and Unsupervised Methods

Supervised learning is a class of machine learning tasks that uses a training set composed of pairs of inputs and outputs. The objective is to estimate the parameters of a model that predicts the expected output from the input data, as shown in Figure 2.1. The two main families of supervised learning tasks are classification and regression. Classification is a task that consists in predicting a class, or category, from a given input, while regression is a task that consists in predicting a continuous value from a given input. Unsupervised learning is a class of machine learning tasks that use training sets composed of inputs without outputs. These methods are often used to find structures or patterns in a given dataset, for instance during the data exploration and analysis phase of data mining.

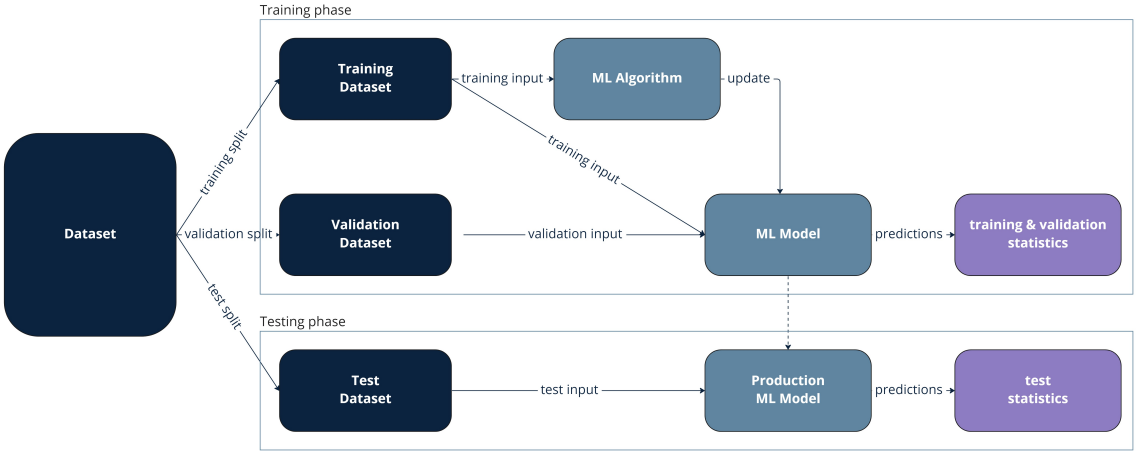

Figure 2.1: Representation of the supervised learning principle.

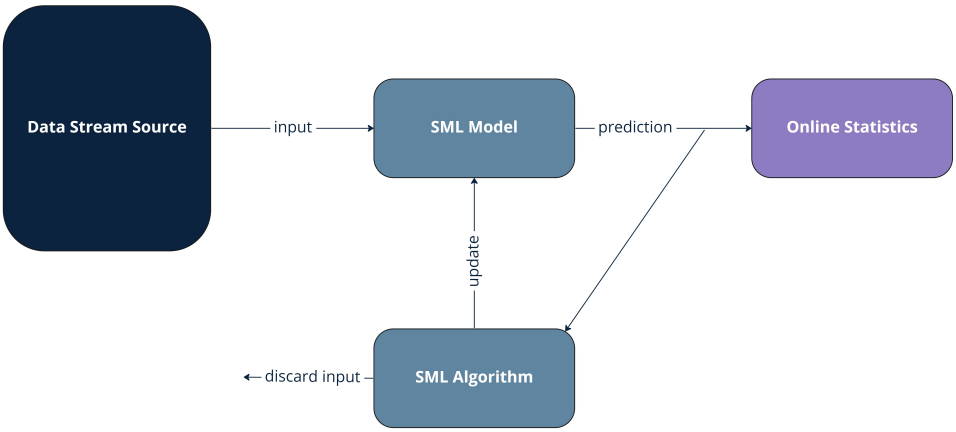

Figure 2.2: Representation of the streaming machine learning principle.

2.1.2. Streaming Machine Learning Approach

Streaming machine learning is an extension of machine learning in principle. It consists in updating a model, from a dataset, finite or infinite (theoretically unbounded), which is received as a stream, as illustrated in Figure 2.2. Within this framework, it is not possible to define a priori different datasets for learning, validating, and testing. This thesis focuses on supervised streaming machine learning models for classification tasks, which already allows for the implementation of experimental setups requiring large amounts of computation. This work might be extended to other tasks, such as regression or unsupervised learning tasks.

Classification is the only task studied in this thesis. The input is made of attributes, which are numerical, categorical, or ordinal variables. The output is a class, or category, which is a categorical variable.

2.1.3. Concept Drifts

The concept drift [2] is a phenomenon that occurs when the distribution of data changes over time. From a probabilistic point of view, given a set of input data $X1, X2, ..., Xn$, classification of Xn amounts to finding the class c^* that maximizes the conditional probability of the class, knowing the input data, see Eq. 2.1.

$$c^* = \arg\max_{c \in C} P(c|X_n) \tag{2.1}$$

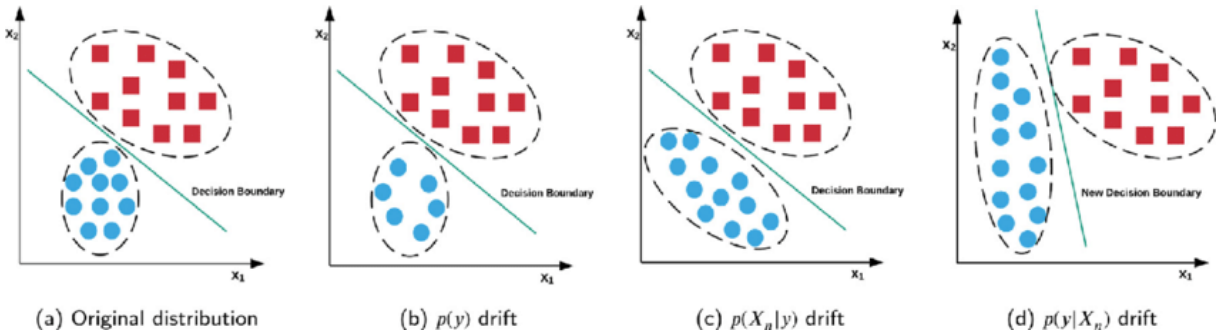

Figure 2.3: Representation of the three different types of concept drift, src. [10]

Or, after development with Bayes' law, see Eq 2.2,

$$c^* = \arg\max_{c \in C} \frac{P(X_n|c)P(c)}{P(X_n)} \tag{2.2}$$

where C is the set of classes, $P(c)$ is the prior probability of the class c, $P(X_n|c)$ is the likelihood of the input data X_n given the class c, and $P(X_n)$ is the marginal probability of the input data X_n.

A concept drift, sudden or gradual depending on the number of data points (or window) leading from one distribution to another, can thus derive its characteristics from a change in the distribution of $P(c)$ and/or a change in $P(X_n|c)$ which can result in a change in $P(c|X_n)$. It is usually complex to identify these changes, especially because they are interdependent. The fact is that when $P(c|X_n)$ changes, the machine learning model loses its ability to correctly predict the class of Xn, as shown in Figure 2.3. If the change is sudden or even brutal, the model can completely lose it. To overcome this problem, a standard machine learning model must be re-trained on a new dataset, collected after the distribution change. This can be costly in terms of time and resources and may be problematic before the model is retrained as it no longer performs as expected. Within the streaming machine learning framework, as each data point is processed as it arrives, it is possible to apply concept drift detection methods using sliding windows and to automate the decision-making to forget or not some old parameters of the model.

ADaptive WINdowing (ADWIN)

`ADWIN` [11] is a concept drift detection method that uses a sliding window. It is currently one of the most efficient common methods to address this problem. With each new data point coming from the stream, the method tries to find two complementary subwindows from the sliding window, as illustrated in Figure 2.4, which are statistically different according to a threshold tuned as a hyperparameter. If two subwindows meeting these

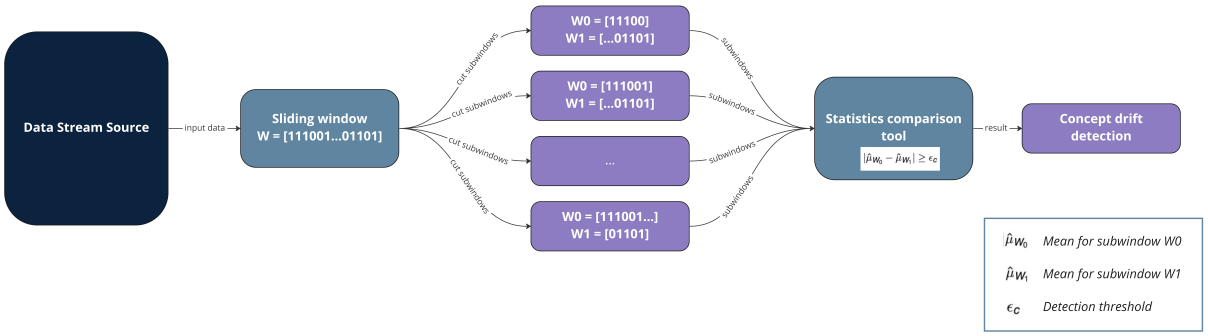

Figure 2.4: Representation of ADWIN concept drift detection method, inspired by [12].

requirements are found, the method keeps the one with the most recent data. Adaptive models use this data to train an alternative model or part of the existing model that replaces the original one if it performs better.

2.1.4. Performance Evaluation

Evaluating the performance of a model is an essential step before its deployment. While a standard machine learning model is evaluated on a dedicated validation set during training, streaming machine learning has no such set. As data points arrive on the fly, a palette of dedicated solutions must be used:

- **Prequential evaluation**

  The prequential evaluation method [13, 14], shown in Figure 2.2, is directly adapted to the principle of online learning. It consists in running the model on each new data point to make the corresponding prediction, before using it for training. A common approach consists in giving more weight to recent data, to avoid, in the case of an infinite or too large stream, that the evaluation is no longer representative of the current state of the model, as it would become increasingly difficult to vary the value of the metric. Two methods are generally used for this purpose:

  – use a sliding window and evaluate the model only on the k last data.

  – use a decay factor α, with $\alpha \in [0, 1]$ and weight the data by a decreasing power of α.

- **Interleaved Test Then Train**

  The interleaved test then train evaluation method is the same as the as the prequential method, without using any weighting method.

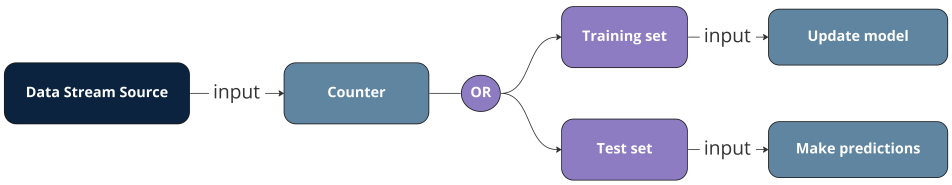

Figure 2.5: Representation of the Periodic Hold-Out performance evaluation method.

- **Periodic Hold-Out**

  The periodic hold-out evaluation method, illustrated in Figure 2.5, closer to the evaluation methods used for standard machine learning, consists in evaluating the model on a test set, periodically taken from the data stream, whose data are thus not used for learning. The size of each test set is a hyperparameter, as is the evaluation frequency. This method, unlike the two previous ones, has the disadvantage of not using all the available data for learning.

Accuracy

Accuracy is a classical performance metric for a classification task. It is defined as the ratio between the number of correct predictions and the total number of predictions, see Eq. 2.3. Accuracy is therefore comprised between 0 and 1, 1 being the ideal value.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{2.3}$$

A usual way to track performance in a classification task is to use the confusion matrix, see Table 2.1, which is a table that summarizes the results obtained so far.

It contains the number of true positives, true negatives, false positives, and false negatives. From this perspective, accuracy can be defined as in Eq. 2.4, where TP, TN, FP, and FN are the number of true positives, true negatives, false positives, and false negatives respectively.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.4}$$

2.1.5. Streaming Machine Learning Classification Algorithms

Many machine learning algorithms exist, and many of them were adapted for streaming machine learning. Still in the context of classification, it is possible to group the main models into three groups:

	Classified as pos.	Classified as neg.
Data labeled pos.	True Positives (TP)	False Negatives (FN)
Data labeled neg.	False Positives (FP)	True Negatives (TN)

Table 2.1: Confusion matrix.

- Decision trees.
- Naive bayes.
- Ensemble models, which use several models, usually decision trees.

Decision Tree

Decision trees [15] are non-parametric machine learning models, i.e., they do not adjust any parameter. These models split at each node into sub-trees, depending on a condition on a variable, up to the leaves that make the final prediction, on a voting or probabilistic basis for example. The principle is illustrated in Figure 2.6. Thus, each node tests a variable, each branch represents a range of values of that variable, and each leaf assigns a class. The problem that arises in the streaming configuration lies in the incremental construction of the tree. The objective is to build a tree that is identical, with a high probability and theoretical guarantees on the error rate, to the one that would be obtained on the stored dataset. This is possible thanks to the Hoeffding bound, giving birth to the `Hoeffding Tree` algorithm, and its derived version the `Hoeffding Adaptive Tree`.

- **Hoeffding Tree**

  The `Hoeffding Tree` [16] (or `Very Fast Decision Tree` - `VFDT`/`HT`) algorithm is an incrementally constructed decision tree that uses the Hoeffding bound to determine whether there is enough statistical evidence to split a new node. This prevents the model from deciding to split or not on a too small sample that is not representative of reality. The condition for creating a new node is given by the formula of Eq. 2.5, where $G(x)$ is the impurity measure of a node, e.g., entropy or gini, of the variable x. x_{parent} and x_{child} are two informative variables, N is the number of observations, δ is the confidence bound and R is the range of G.

$$G(x_{parent}) - G(x_{child}) > \epsilon = \sqrt{\frac{R^2 * \log \frac{1}{\delta}}{2N}} \tag{2.5}$$

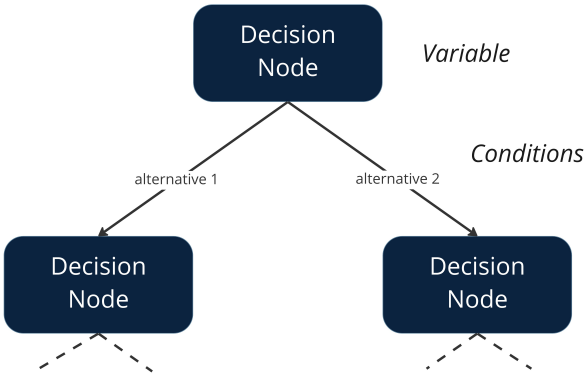

Figure 2.6: Decision tree principle.

$G(x_{parent}) - G(x_{child})$ represents the information gain of creating a new node at x_{child}. δ is defined as in Eq. 2.6

$$\delta = 1 - \text{Desired proba. of choosing the correct attribute at any given node} \tag{2.6}$$

- **Hoeffding Adaptive Tree**

  The `Hoeffding Adaptive Tree` [17] (or `HAT`) algorithm is an extension of the `Hoeffding Tree` algorithm, which allows to incrementally build decision trees with an increased capacity of reaction to concept drifts. The `HAT` algorithm uses the `ADWIN` concept drift detection algorithm at each branch (each node), to replace it when a concept drift is detected. When the performance of a branch degrades, `ADWIN` provides the subwindow of recent data that is expected to follow the new distribution, and the `HAT` algorithm uses it to build an alternative branch, ready to be used if its results are better than those of the original branch.

Naive Bayes

`Naive Bayes` [18] is a non-parametric probabilistic classification model, which is based on Bayes' theorem. The name `naive` comes from the fact that the model assumes that the variables are conditionally independent, which remains a strong assumption. The assumed conditional independence amounts to writing Eq. 2.7, where x_i is the i^{th} independent variable and y the class.

$$P(x_1, x_2, ..., x_n|y) = \prod_{i=1}^{n} P(x_i|y) \tag{2.7}$$

Using this assumption in Bayes' formula results in Eq. 2.8.

$$P(y|x_1, x_2, ..., x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, ..., x_n)} \tag{2.8}$$

Finally, the `Naive Bayes` classifier returns the class y^* that maximizes the probability $P(y|x_1, x_2, ..., x_n)$, as described in Eq. 2.7, which gives Eq. 2.9.

$$y^* = \underset{y}{\text{argmax}} P(y|x_1, x_2, ..., x_n) \tag{2.9}$$

Adapting this model to a data stream is simply a matter of updating the estimators of the useful distributions for the prediction in an incremental way [19]. For a mean, the incremental update is given by Eq. 2.10, where s_i is the sum of the first i observations, and $\hat{x_i}$ is the mean of the first i observations [20].

$$\begin{cases} s_i = s_{i-1} + x_i, & \text{(2.10a)} \\ \hat{x_i} = \dfrac{s_i}{i} & \text{(2.10b)} \end{cases}$$

For a variance estimator, the incremental update is given by Eq. 2.11, where q_i is the sum of the squares of the first i observations, and ${\sigma_i}^2$ is the variance of the first i observations.

$$\begin{cases} q_i = q_{i-1} + {x_i}^2, & \text{(2.11a)} \\ {\sigma_i}^2 = \dfrac{1}{i-1} * (q_i - \dfrac{{s_i}^2}{i}) & \text{(2.11b)} \end{cases}$$

Ensemble Methods Approach

Ensemble methods [21] for classification use multiple base learners, typically decision trees, whose individual predictions are combined to produce a more robust final prediction. It is possible to induce diversity by several means, to combine models in different ways, and to combine their predictions in different ways as well. For streaming machine learning, ensemble methods [22] could also be used to enhance the ability of the overall model to adapt to evolving data [23].

To induce diversity, the main options are:

- **Horizontal Partitioning**

  For horizontal partitioning, a common method is bagging [8]. It consists in combining several models, each with a large variance, i.e., that fit the training data

particularly well, but which are more easily subject to the phenomenon of overfitting [24]. M models are trained, each with a different bootstrap sample of the training data. Each bootstrap sample is a random sample with replacement of the same size as the original training data, where each data point is chosen K times with $P(K = k)$ following a binomial distribution. Within the streaming machine learning framework, bagging was adapted into online bagging, see Section 2.1.5, by using a Poisson distribution.

Another common method is boosting [25], which consists in combining several models with a large bias, i.e., which are not able to fit the training data particularly well, but which are more robust to overfitting. M models are trained, each with a different training dataset, which is a weighted version of the original training dataset. The weights are updated at each iteration so that the next model is trained on the data points where the previous model made the most errors.

- **Vertical Partitioning**

  For vertical partitioning [22], a common approach is to consider random subspaces of the original feature space. The number of features in the subspace is usually much smaller than the original number of features. Each learner is then trained on a different subspace. If decision trees are used as base learners, local randomization or global randomization can be applied, as shown in Figure 2.7. Local randomization consists in randomly selecting a subset of features at each node of the tree, while global randomization consists in randomly selecting a subset of features for the whole tree. The difference is illustrated in Figure 2.7.

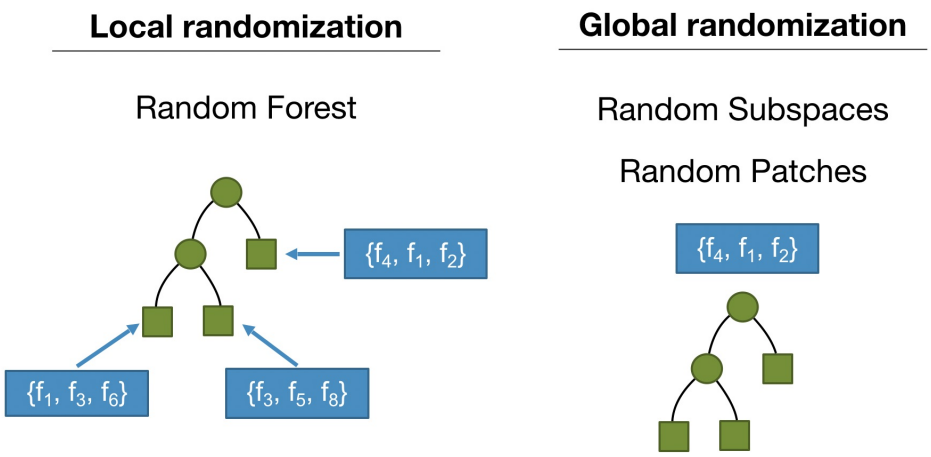

Figure 2.7: Vertical partitioning for decision trees, src. [26]

- **Others**

  Other methods for inducing diversity include base learner manipulation, i.e., varying the parameters of the same base learner, or using different base learners to obtain members with different biases.

To combine predictions, the main aspects are:

- **Combination Architecture**

  As illustrated in Figure 2.8, base learners can be organized in different ways, from a flat architecture where each learner directly feeds the voting combination layer, to the meta learner architecture, where a meta learner takes the predictions of the base learners as input, and outputs the final prediction, or to a hierarchical architecture where the base learners are organized in layers where different combinations are possible.

- **Voting Combination**

  Base learners' predictions can also be combined in different ways, as shown in Figure 2.9. The simplest way is to use majority voting, i.e., to consider the class that is predicted by a majority of base learners. Another common method consists in using weighted voting, where the weight of each learner is proportional to a criterion such as its accuracy on the training dataset. Each base learner could also rather provide a ranking of the probable classes, and all rankings could be combined to produce a final one.

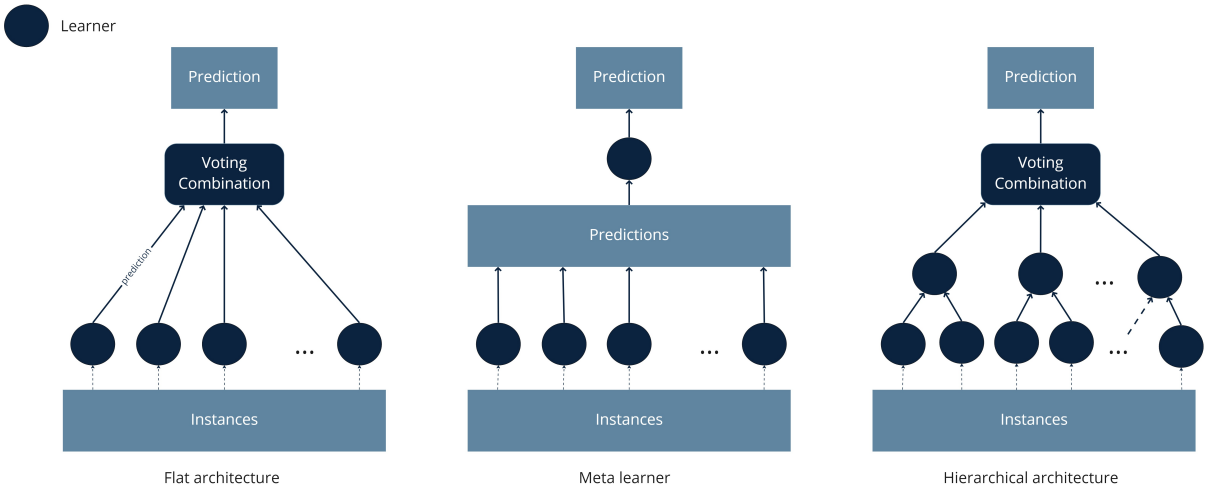

Figure 2.8: Main combination architectures, inspired by [27].

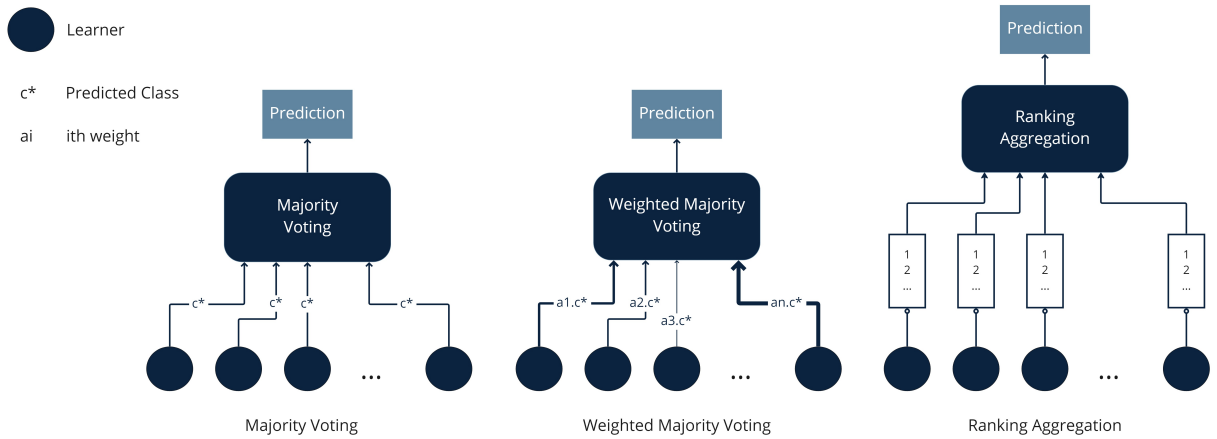

Figure 2.9: Main voting combinations, inspired by [27].

Finally, to adapt to evolving data, the main aspects are:

- **Cardinality**

  A fixed number of learners can be used, or learners can be added dynamically on the fly.

- **Learning Mode**

  Data streams can be processed in different ways by learners, depending on the type of windowing used. For instance, learners can be trained incrementally on the data stream, on a sliding window, on a landmark window or an adaptive window, etc.

Ensemble Methods: Online Bagging

Within the online framework, the training data is not available in advance, and it is consequently impossible to use the bootstrap sampling method in the same way as in the offline framework. To overcome this problem, the online bagging method, also called the `Oza Bagging` algorithm after one of its authors [28], uses a $Poisson(\lambda = 1)$ [29] distribution, see Figure 2.10. instead of a binomial distribution. Indeed, for a large number of samples, the binomial distribution tends to be a $Poisson(\lambda = 1)$ distribution. For each new arriving sample, the $Poisson(\lambda = 1)$ distribution determines the number of times the sample is used to train each base learner.

Ensemble Methods: Leveraging Bagging

`Leveraging Bagging` [30] is an online bagging method that uses the $Poisson(\lambda = 6)$ distribution instead of the $Poisson(\lambda = 1)$ distribution as in `Oza Bagging`. The rationale behind this choice is that the $Poisson(\lambda = 6)$ distribution adds more randomization. In

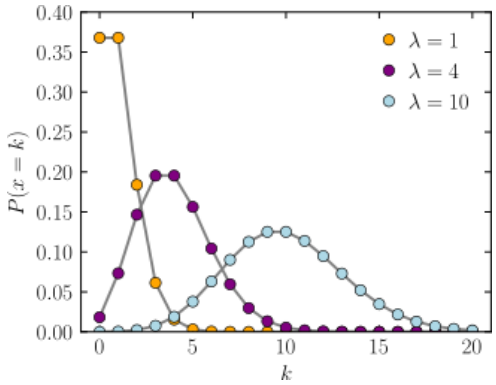

Figure 2.10: Poisson distribution for $\lambda \in \{1, 4, 10\}$ src. [29]

addition to increasing the resampling, `Leveraging Bagging` also uses the `ADWIN` concept drift detector to adapt better in case of concept drift. Each learner feeds its predictions to an `ADWIN` detector. After a change is detected, the worst learner of the ensemble, based on the error estimation by `ADWIN`, is replaced by a new one.

2.2. Cloud and Edge Computing

When embedded systems are used to collect data at the source, there are two paradigms for processing it. Those are cloud computing and edge computing, depending on where the computations occur.

2.2.1. Cloud Computing

Traditionally, data collected by embedded systems is sent to remote servers in the cloud. Servers then take care of data processing, for example, to update a machine learning model. This paradigm has several advantages, including:

- **Local bandwidth savings**

  The data is sent to a remote server, not to another embedded system through a local network.

- **Storage savings**

  Data is not stored on the embedded system.

- **Removal of resource constraints**

  Embedded systems are often resource-constrained, and cloud computing makes it possible to perform complex tasks without limitations due to available resources.

However, this paradigm also has disadvantages, notably a massive use of telecommunication networks, at the risk of congestion, and more seriously, dependence on them. This solution is unthinkable in contexts where telecommunication networks are not available or are not reliable. This solution can also be costly, as it requires the implementation of a cloud computing infrastructure.

2.2.2. Edge Computing

In recent years, edge computing [31] has become an increasingly popular solution for processing data collected by embedded systems. It consists in processing the data collected as close as possible to the source, as represented in Figure 2.11. This solution eliminates the disadvantages of cloud computing but presents engineers with new challenges. It requires dealing with a limited amount of resources, while applications are becoming increasingly complex. To overcome this problem, it is necessary to define an architecture adapted to the number of computations, and deploy it on a network of embedded systems. Thus, the multiplication of embedded systems allows to distribute the computation load but requires new cooperation algorithms between systems. On the other hand, it is also possible to work on data processing algorithms, to make them more efficient or lighter so that a single embedded system can support the computations.

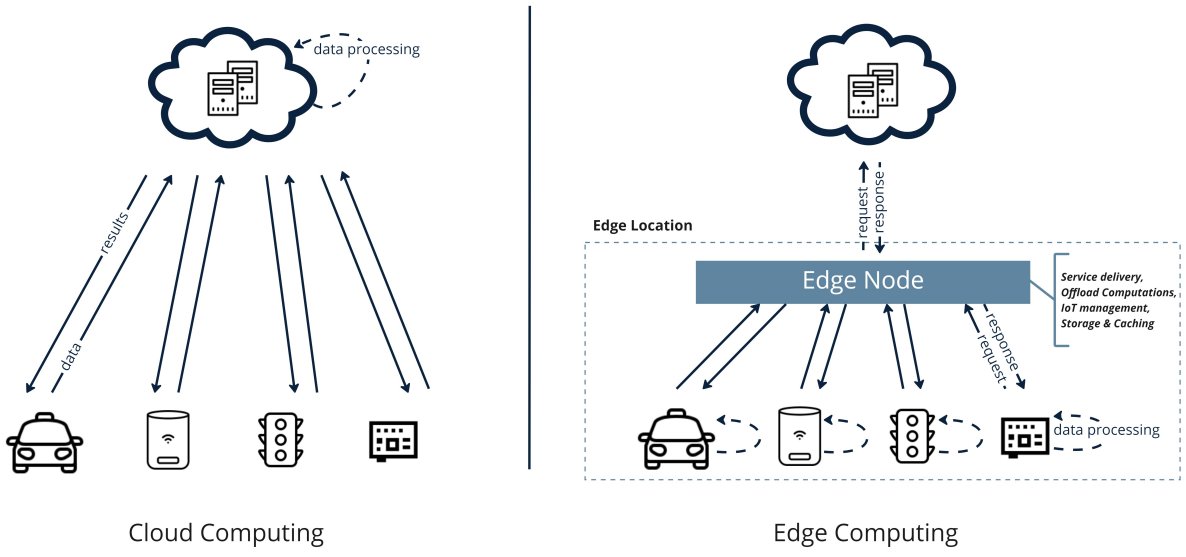

Figure 2.11: Edge computing V.S Cloud computing

2.3. Tiny Machine Learning and Quantization

2.3.1. tinyML

`tinyML` is a term that refers to a subfield of machine learning that focuses on the development of machine learning technologies and applications for resource-constrained devices. According to the `tinyML Foundation`[1]:

> *"Tiny machine learning is broadly defined as a fast growing field of machine learning technologies and applications including hardware, algorithms and software capable of performing on-device sensor data analytics at extremely low power, typically in the mW range and below, and hence enabling a variety of always-on use-cases and targeting battery operated devices."*

The development of this field is a direct consequence of the evolution of edge computing and its advantages, applied to one of the most computationally intensive fields. A great illustration of this trend is `TensorFlow Lite`[2], i.e., the lite version of the `TensorFlow` library, which is designed to optimize deep learning models for mobile and edge devices such as microcontrollers. A common practice to lighten models is to use quantization, as discussed in Section 2.3.2.

2.3.2. Quantization

Quantization has its origin in signal processing and consists in approximating a continuous signal by the values of a discrete set. For software, quantization consists in approximating floating values by integers or by other floating values with less precision. Execution is then lighter, which is particularly interesting for systems with limited resources. The key point is to understand what is quantized, when, and how. For instance, with `Tensorflow Lite`, the weights of the neural network are post-training quantized. This means that a model is first trained on a powerful machine, and the adjusted weights are then processed to make the model lighter for deployment.

Within the streaming machine learning framework, the traditional training and production phases are merged, thus the edge device is responsible for both training and testing. In this thesis, quantization is performed on `tinyMOA`, by reducing the precision of 64-bit float variables in the source code and compiling a new library. This makes quite a difference with the example of `TensorFlow Lite` where only the production phase is quantized.

[1] `https://www.tinyml.org/`
[2] `https://www.tensorflow.org/lite/microcontrollers`

2.4. Related Work

Several works were conducted around the topics of interest of this thesis. The article [4] is of particular interest, as it focuses on the execution of MOA on a Raspberry Pi, in a configuration using Kafka[3] and Parsl[4]. Parsl is a python library that allows to parallelize codes so that Apps execute concurrently while respecting data dependencies. It is used in this case to dynamically deploy MOA algorithms on the available computational resources, including the Raspberry Pi. Kafka is a message broker that is used on the same device as the Parsl Executor to support stream management. The data used is the Census dataset [32], which aims at predicting whether a person earns more than $50,000 per year. The comparison of MOA performance between Raspberry Pi and high-performance computing resources is detailed, in the form of a series of what-if scenarios. The article also introduces a compact version of MOA named tinyMOA, which simply has fewer features than MOA, for a smaller software size. This paper shows interesting results, in this particular configuration, which seem to confirm the feasibility and the interest of the first part of this thesis, which also focuses on the execution of MOA on the Raspberry Pi.

Other articles present the execution of standard machine learning algorithms on the Raspberry Pi, or streaming machine learning ones for specific uses, such as intrusion detection. Article [33] for instance presents the execution of standard machine learning algorithms, using WEKA, on several Raspberry Pis, with sensor data. The trained models are frequently sent from the Raspberry Pis to a central server in the cloud, that averages them. Then, it returns the averaged model to the Raspberry Pis, and the process is repeated. Article [34] presents a lightweight detection system for cyber attacks, based on the Hoeffding Tree algorithm. This other article [35] shows the execution of deep learning models for video analysis and object detection on the Raspberry Pi. Article [36] gets deeper into the utilization of streaming machine learning for subspace analysis, a task that consists in finding subspaces of the original space such that the corresponding projection has some special properties. A typical example is the Principal Component Analysis (PCA) algorithm, which aims at finding the subspace that maximizes the variance of the projected data.

These articles, which are the closest ones to the scope of this thesis, show that the execution of streaming machine learning algorithms on the Raspberry Pi is the logical continuation of what has been done so far, where streaming machine learning has the ability to replace machine learning for more and more applications, and where edge devices replace high-performance computing resources.

[3] https://kafka.apache.org/
[4] https://parsl-project.org

3 | Objectives and Proposed Approach

In this chapter are presented both the problem to be solved and the adopted approach to reach a solution, with each step clearly formulated. For each one, the imagined experiments, their components, and the selected configurations are detailed. Thus, this chapter presents in a comprehensive manner all the work that has been done for this thesis. Then, the algorithms, scripts, and leveraging technologies are detailed in Chapter 4 while the results are exposed in Chapter 5.

3.1. Problem Statement

This thesis is structured around two main axes. The first one is the study of the Raspberry Pi's suitability to execute streaming machine learning algorithms. The purpose of the suitability analysis is to determine:

1. whether it is possible to run `MOA` with reasonable performance on the Raspberry Pi,
2. how far the performance of `MOA` on the Raspberry Pi is from the performance obtained with a desktop computer.

The second axis is the study of an innovative improvement possibility of the results obtained with `MOA` on the Raspberry Pi. This improvement possibility consists in quantizing the `tinyMOA` software, proposed as a compact alternative to `MOA` in the article [4], as explained in Section 1.1, to further lighten its execution and thus propose a version adapted to embedded systems. The improvement study aims at determining:

1. whether it is possible and how to compile a quantized version of `tinyMOA`,
2. whether there is a significant gain in execution performance with this quantized version and to what extent the quantization impacts accuracy in return.

3.2. Approach Overview

The adopted approach to address the issues outlined in the previous section consists of five steps.

- **Suitability Analysis** (3 steps)

  The first step is to study, choose and implement a virtual version of the Raspberry Pi that will be suitable to run `MOA` and estimate the performance of a physical Raspberry Pi board. This step allows the assessment of the feasibility of the project, and once done, implements a suitable working environment to estimate what would be the performance of `MOA` on a physical Raspberry Pi board. For this purpose, this step also includes the execution of a set of benchmarks. These benchmarks were previously executed on physical boards whose results are available in the public domain so that it is possible to evaluate the performance differences between the virtual and the physical versions and thus calibrate the virtual environment against the physical one. The second step consists in running a series of experiments with `MOA` on the Raspberry Pi emulators. These experiments allow to confirm or deny the potential suitability of the Raspberry Pi to run `MOA`. After confirmation, the third step begins. A physical Raspberry Pi board is used to run these experiments, as well as a desktop computer, to finally compare their results.

- **Improvement study** (2 steps)

  The fourth step consists in implementing a quantized version of `tinyMOA`, named `tinyMoa-lite`, and running a series of experiments on the Raspberry Pi to compare this version and the original one. The fifth and final step consists in determining if there exists a compromise between the original version and the quantized one which would offer a perfectly similar execution to the original version. This quasi-quantized version is a variant of the quantized version with the minimum possible number of 64-bit variables such that it generates the very same solutions as the original version. Indeed, since both the training and testing phases are quantized, the obtained solutions between the two versions may differ.

3.3. Raspberry Pi Simulation

There exist different versions of the Raspberry Pi, which is now developed for more than 10 years. The ones that could offer a suitable working environment to run `MOA` are the `Raspberry Pi 3b` (or `3b+`) and the `Raspberry Pi 4b`. The `Raspberry Pi 2b`, although

	RPi 2b	RPi 3b	RPi 3b+	RPi 4b
Processor	Cortex-A7	Cortex-A53	Cortex-A53	Cortex-A72
Processor specs.	ARMv7 900MHz	ARMv8 1.2GHz	ARMv8 1.4GHz	ARMv8 1.5GHz
CPU	Quad-Core	Quad-Core	Quad-Core	Quad-Core
Thread per core	1	1	1	1
Instruction length	32 bits	64 bits	64 bits	64 bits
RAM	1GB	1GB	1GB	1,2,4 or 8GB

Table 3.1: Comparative table of the Raspberry Pi models.

older, can also be considered as an option, with lower performance but lower cost. As virtualization allows to try out several versions at no extra cost, all three versions, `2b`, `3b+`, and `4b`, are virtualized. The `Raspberry Pi 2b` features an ARMv7 900MHz processor, 1GB of RAM, and four USB ports. The `Raspberry Pi 3b` offers an ARMv8 1.2GHz processor, 1GB of RAM, and 4 USB ports. The intermediate version between the `3b` and `4b`, the `3b+`, offers an ARMv8 1.4GHz processor, 1GB of RAM and 4 USB ports. Its architecture is similar to the `3b` with a slightly better processor. This thesis presents the results with the `3b+` version rather than the `3b`. The `Raspberry Pi 4b` offers an ARMv8 1.5GHz processor, 1GB of RAM, and 4 USB ports. The ARMv8 architecture is a major evolution of the ARMv7 architecture since it allows 64-bit instructions to be executed instead of 32-bit ones. Table 3.1 recapitulates these differences.

To virtualize a Raspberry Pi, several solutions exist. In particular, two families of solutions were considered: simulators and emulators.

3.3.1. Simulators and Emulators

Simulators are solutions that offer a simulation of the electronic circuits of the Raspberry Pi. This allows the user to run and test programs that make use of components such as `GPIO` (`General Purpose Input Output`) ports or other specific modules. They are mainly interfaces that only allow users to execute code snippets of certain languages, like Python or NodeJS. Simulators are therefore interesting solutions for testing ad hoc programs, but they do not provide the user any access to a terminal and do not allow to estimate the performance of the Raspberry Pi as only the circuits are simulated.

Emulators are more complete but heavier solutions, which propose to virtualize a set of

physical components, and thus replicate the execution of the desired system. To emulate a complete Raspberry Pi, especially from a non-ARM system, both the software and the hardware must be virtualized. This includes emulating the ARM processor, memory, peripherals, and operating system. It is possible to run any ARM-compatible operating system on top of the virtualized hardware, but there exists a dedicated operating system to the Raspberry Pi, named `Raspberry Pi OS` (formerly `Raspbian`). Emulation is the chosen solution for the sequel of the thesis, as it allows to perfectly replicate the execution of the physical board. More specifically, `QEMU` [37] was used. This solution offers great customization of the emulation, proposes almost all the common hardware components, and gives direct access to a terminal, which offers the user a total freedom.

QEMU

`QEMU` is an open-source emulator that allows to emulate a set of components, and thus replicate the execution of the desired system. It also offers specific support for emulating the `Raspberry Pi 2b`, `3b`, and `3b+`, which gives access to a set of specific components such as `GPIO` ports.

Raspberry Pi 2b Emulation

For the `Raspberry Pi 2b`, `QEMU v2.9.0` was used, with specific support for the `Raspberry Pi 2b` board. This emulation implements a `Broadcom 2709` quad-core cortex-A7 CPU at 900MHz, 512MB of RAM, and the `Raspbian OS Jessie` (2016-05-27 release) operating system. The operating system is installed on a virtual SD card.

Raspberry Pi 3b+ Emulation

For the `Raspberry Pi 3b+`, `QEMU v6.2.0` was used, with specific support for the `Raspberry Pi 3b+` board. This emulation implements a `Broadcom 2710` quad-core cortex-A53 CPU at 1.4GHz, 1GB of RAM, and the `Raspberry Pi OS Bullseye ARM64` operating system (2022-01-28 release).

Raspberry Pi 4b Emulation

For the `Raspberry Pi 4b`, `QEMU v6.2.0` was used, but no specific support for this board was available at this time. The emulation is supported by `Virtio` with the components of the `Raspberry Pi 4b`. `Virtio` is a `Linux` kernel programming interface useful for virtual machines. It allows virtual devices to be managed as physical devices directly within the `Linux` kernel. This emulation implements a `Broadcom 2711` quad-core cortex-A72 CPU at 1.5GHz, 1GB of RAM, and the `Raspberry Pi OS Bullseye ARM64` operating system

	RPi 2b	RPi 3b+	RPi4b
Emulated processor	Cortex-A7	Cortex-A53	Cortex-A72
Processor specs.	ARMv7 900MHz	ARMv8 1.4GHz	ARMv8 1.5GHz
QEMU version	v2.9.0	v6.2.0	v6.2.0
Specific board support	Yes	Yes	No
OS	Raspbian Jessie	Raspberry Pi OS Bullseye ARM64	Raspberry Pi OS Bullseye ARM64
Instruction length	32 bits	64 bits	64 bits
RAM	1GB	1GB	4GB

Table 3.2: Comparative table of the Raspberry Pi emulations.

(2022-01-28 release). Table 3.2 recapitulates the different emulations implemented.

3.3.2. Benchmarks

After implementing the different emulations, different benchmarks were used to evaluate their performance and compare them to their physical counterparts, as a kind of calibration. This step is crucial before running the series of experiments with `MOA`, as it will allow to estimate the potential performance of the physical boards with the emulations. Benchmarks are programs that measure the performance of a system. They are often used to compare several systems and they usually consist of a series of operations that are repeated a large number of times. The main performance dimensions are the execution time, the resource consumption, or the number of computations performed. Within the context of this thesis, two performance dimensions were evaluated for the emulators: the execution throughput and the memory consumption. To evaluate them, parallel executions must be distinguished from sequential ones.

It was necessary to run benchmarks whose performance on the Raspberry Pi were available in the public domain. Thus, the benchmarks used come from the publication [38], where the compiled files are provided along with the results obtained on a couple of Raspberry Pi models: the `Raspberry Pi 3b+` and the `Raspberry Pi 4b`. Consequently, the emulation of the `Raspberry Pi 2b` was not used for the following, and the `Raspberry Pi 2b` solution was then definitively abandoned. Only the 64-bit benchmarks were executed, although the 32-bit versions were also available, as `MOA` is run with 64-bit instructions.

Single Thread Benchmarks CPU

CPU benchmarks focus on the device execution throughput. It measures the number of computations per execution time ratio. Sequential executions make use of a single thread and consist of operations that need to be executed sequentially, i.e., with no parallelism, and that are repeated a large number of times. The benchmarks used are the following:

- **Whetstone**: a benchmark that executes basic computational loops (e.g., cos, exp) and evaluates performance on floating-point numbers.
- **Dhrystone**: a benchmark that executes basic computation loops (e.g., addition, multiplication) and evaluates the performance on integers.
- **Linpack**: a benchmark that executes matrix calculations by solving linear systems. It evaluates the performance for scientific applications.
- **FFT**: a benchmark that performs Fourier transform calculations. It evaluates the performance on floating point numbers, single and double precision.
- **Livermore**: a benchmark that runs physics and chemistry calculations. It evaluates the performance on floating point numbers, single and double precision.

Multiple Threads Benchmarks CPU

These benchmarks make use of multiple threads as they consist in executing operations with a certain degree of parallelism. The benchmarks used are the following:

- **MP Whetstone**: a benchmark that runs 8 test functions at the same time with dedicated variables. The performance measurement is based on the last thread to finish. Common data can only be accessed by one thread at a time. The process is repeated with 1,2,4 and 8 threads.
- **MP Dhrystone 2**: a benchmark that runs several copies of `Dhrystone` at the same time with dedicated data lists but many common variables.
- **MP Linpack NEON**: linear algebra routines with 128-bit `NEON` registers, which are registers adapted to vector calculations, in unrolled loops with four-dimensional differential equations.
- **MP MFLOPS**: a benchmark that performs calculations on float variables, in cache and in memory. Several operations are performed on each word, and each thread supports the same calculation, from different segments of memory.

Single Thread Memory Benchmarks

Memory benchmarks focus on how well the device reads data in cache and memory. Results are often expressed in MB/sec. Similarly, sequential and parallelized executions have been distinguished for memory benchmarks. The sequential single-thread memory benchmarks are as follows:

- **BusSpeed**: a read-only benchmark that reads word after word, with different increments.
- **MemorySpeed**: performs computations on lists in memory and in cache, and measures reading rates.
- **NeonSpeed**: similar to the `MemorySpeed` benchmark, with `NEON` registers and directives.

Multiple Threads Memory Benchmarks

The parallelized memory benchmarks are the following ones:

- **MP BusSpeed**: a read-only benchmark that reads integers word by word, each thread on different memory segments.
- **MP RandMem**: a benchmark that reads and writes integers randomly in memory. Each thread starts at a different location and then accesses the memory randomly.

3.4. Emulator Experiments with MOA

Once the emulators were implemented, and their performance calibrated against the physical boards, a series of experiments with `MOA` was performed. To conduct the experiments, `MOA` was used, as well as its compact version `tinyMOA`. `MOA` offers two interfaces: a graphical interface and a command line interface. With `tinyMOA`, the `.jar` file is lighter, 900 KB against 2.8 MB for `MOA`, which represents a significant gain an embedded use, with limited memory.

3.4.1. Datasets

Within the streaming machine learning framework, a central element remains the data stream. Two types of data streams can be used to conduct experiments, those synthetically generated and real datasets.

Synthetic Datasets

Synthetically generated datasets are generated by algorithms. They use pseudo-random number generators, and other mathematical functions to generate data with certain properties. `MOA` directly provides a set of data generators, which can be parameterized. The following synthetic generators were used for the experiments run on the couple of emulators:

- **RandomTreeGenerator** : generates a classification problem via a randomly generated decision tree. It constructs a decision tree by choosing attributes at random to split, and assigning a random class label to each leaf. Once the tree is built, new instances are generated by assigning uniformly distributed values to attributes, and then traversing the tree to find the class label This generator comes from the publication [16].
- **RandomRBFGenerator** : generates a random radial basis function stream. While the `RandomTreeGenerator` generates a classification problem adapted to decision trees, the `RandomRBFGenerator` generates an alternate complex concept type that is not easily learned by decision trees. A fixed number of random centroids are generated. Each center has a random position, a single standard deviation, class label and weight. New instances are generated by selecting a center at random so that the probability of selecting a it is proportional to its weight. A random direction is chosen to offset the attribute values from the central point. The length of the displacement is randomly drawn from a gaussian distribution with the standard deviation of the center. The class label is determined by the centroid.
- **RandomRBFGeneratorDrift** : same as the `RandomRBFGenerator`, but the concept drift is directly introduced via the generator by moving the centroids with constant speed.
- **SEAGenerator** : generates `SEA` concepts functions [39]. The points of the dataset are divided into four blocks with different concepts. That is why this generator is used for the concept drift based experimentss in the sequel of this thesis, as it is easily possible to change and parametrize the concept of the generated data.
- **AgrawalGenerator** : generates one of the ten predefined loan functions as introduced by Agrawal et al. [40]. This generator is also used in the sequel of the thesis, along with the `SEAGenerator`, to configure experiments that implement concept drifts as it is easy to switch between different functions by parametrizing it.
- **LEDGenerator** : generates a problem of predicting the digit displayed on a seven-

segment LED display where each attribute has a 10% chance of being inverted. This implementation proposes 24 binary attributes, 17 of which are irrelevant. This generator comes from [41].

- **LEDGeneratorDrift** : same as the `LEDGenerator` with concept drift, directly introduced via the generator by changing the concept for certain attributes.
- **WaveFormGenerator** : generates a problem of predicting one of three waveform types [41]. Each waveform type is generated by a combination of three base waves. There are 21 numerical attributes, all of which include noise.
- **WaveFormGeneratorDrift** : same as the `WaveFormGenerator`, with concept drift, directly introduced via the generator by changing the concept for certain attributes.
- **HyperplaneGenerator** : generates a problem of predicting class of rotating hyperplane [42]. This generator is useful to implement incremental drift thanks to its properties as explained in the dedicated Section 3.4.4.

Real Datasets

For the emulators' experiments, only one `ARFF` file dataset was used, which is the dataset `census.arff`. It contains information about the income of the inhabitants of the United States. Specifically, it is a binary classification dataset, which associated task consists in predicting whether an individual earns more or less than $50,000 per year. This dataset appears in the group of paper [4] experiments, where the results obtained on emulators are compared with the results of a physical board.

3.4.2. Algorithms

The algorithms used are the following:

- **HoeffdingTree (HT)**: incremental decision tree `Hoeffding Tree` as presented in Section 2.1.5. This implementation is directly adapted from paper [42]. In the current version of `MOA`, the `HoeffdingTree` algorithm is by default implemented with the `Naive Bayes Adaptive` algorithm at leaves, since it gives the best results. This `Naive Bayes Adaptive` prediction method monitors the error rate of `Majority Class` and `Naive Bayes` decisions in every leaf, and chooses to employ `Naive Bayes` decisions only where they have been more accurate in past cases. The decision method at leaves is tunable, and it is possible to select only `Majority Class` or only `Naive Bayes` in substitution to `Naive Bayes Adaptive`.

In this thesis, `HT (MC)` is the abbreviation for the `Hoeffding Tree` model with the `Majority Class` decision method at leaves.

- **HoeffdingAdaptiveTree (HAT)**: incremental decision tree `Hoeffding Adaptive Tree`, a variant of the `Hoeffding Tree`, with `ADWIN` concept drift detectors at each node for increased adaptation capacity, also presented in Section 2.1.5. This implementation is directly adapted from paper [17]. It uses the `HoeffdingTree` implementation presented above as a base.
- **NaiveBayes (NB)**: the incremental `Naive Bayes` algorithm, as presented in Section 2.1.5.
- **OzaBag (OB)** : the `Oza Bagging` algorithm, as presented in Section 2.1.5. It implements an online bagging method, as described in this paper [28], from N. Oza and S. Russell. For all experiments, 10 `HoeffdingTree`, as described above, were used as base learners.
- **OzaBagAdwin (OBA)** : a variant of `OzaBag`, as described above, with `ADWIN` concept drift detectors for each base learner, so that in case of concept drift, the worst performing base learner is replaced by a new one. This implementation is directly adapted from paper [43].
- **LeveragingBag (LB)** : the `Leveraging Bagging` algorithm, as presented in Section 2.1.5. It implements an online bagging method, with more randomization than `OzaBag` and is directly adapted from this paper [30]. As for `OzaBag`, for all experiments, 10 `HoeffdingTree` were used as base learners.

3.4.3. Evaluation Methods

As presented in Section 2.1.4, several performance evaluation methods exist within the streaming machine learning framework. For these experiments, the following evaluation methods were used:

- **EvaluatePrequential**: the method presented in Section 2.1.4, which consists in first evaluating the model and then updating it for each new incoming instance. With `MOA`, it is possible to use either a sliding window with a tunable size or a fading factor. The implementation is directly adapted from paper [13]. Then, the frequency for reporting the results is also tunable.
- **EvaluateInterleavedTestThenTrain**: the method presented in Section 2.1.4, which is the same as the `Prequential Evaluation`, without any sliding window

or fading factor.

- **EvaluatePeriodicHoldOutTest**: the method presented in Section 2.1.4, which consists in evaluating the model on a hold-out set at a certain frequency. With `MOA`, the frequency can be tuned as well as the size of the hold-out set.

3.4.4. Concept Drift Types

As presented in Section 2.1.3, an important possible concern for streaming machine learning models are concept drift phenomenon. It consists of a change of the underlying distribution of the data, which can occur in different ways. For these experiments, concept drifts are also artificially generated with the following types:

- **Abrupt drift**: the underlying distribution changes abruptly, meaning that the window needed for the data to totally shift is really small (a few instances).
- **Gradual drift**: the underlying distribution changes gradually, meaning that the window needed for the data to totally shift may be more or less large (a few hundreds to thousands of instances). In this window, the data might come from the old distribution or from the new one, the proportion of each gradually changing from [1,0] to [0,1].
- **Incremental drift**: the underlying distribution changes incrementally, meaning that the data changes from one distribution to another smoothly in a continuous way. To do so, the incremental concept drift directly occurs within the data generator, which must be the `Hyperplane Generator`, by varying the values of the hyperplane weights as the data stream is generated. These weights change its relative orientation and position. The `Hyperplane Generator` is chosen for these properties that allow to easily generate incremental concept drifts.

3.4.5. Experiments

The experiments conducted on the emulators can be grouped into two groups. The first group includes the experiments taken from the `MOA` official tutorials. The second group includes the experiments presented in the paper [4], which were performed on a `Raspberry Pi 4b`, with 2GB of RAM, in the special configuration using `Kafka` and `Parsl` previously presented in Section 2.4. Although the execution conditions are different, comparing the results detailed in this paper with those of the emulators ensures that the orders of magnitude in the difference between the virtual and the physical performance, estimated when calibrating the emulators against the benchmarks, are consistent.

	#inst.	algo.	evaluation	generator
Ex. 1	1000000	HT, NB	Interl. Test Train (freq.=10000)	RandomTree
Ex. 2	1000000	HT	Interl. Test Train (freq.=10000), Prequential (freq.=10000, Nwin=1000), Hold-Out (freq.=10000, Ntest=1000)	RandomTree
Ex. 3	1000000	HT, NB	Interl. Test Train (freq.=10000)	RandomRBF
Ex. 4	1000000	HAT, OBA, HT (MC)	Interl. Test Train (freq.=10000)	RandomRBF
Ex. 6	1000000	HT, HAT, OBA	Prequential (freq.=10000, Nwin=1000)	RandomRBF

Table 3.3: MOA Tutorial 1 experiments.

MOA Tutorials

`MOA` tutorials are examples of `MOA` usage, each focusing on different aspects. Two tutorials were followed, each presenting several exercises, leading to a total of 13 experiments. The first tutorial is a basic tutorial, presenting the basic functionalities of `MOA`. Table 3.3 summarizes these experiments, with $freq.$ being the frequency of the evaluation, $Nwin$ the size of the sliding window, and $Ntest$ the size of the hold-out set.

The second tutorial includes experiments about concept drifts, as recapitulated in Table 3.4, with w being the width of the concept drift, i.e., the number of instances to shift from a distribution to another, and k the position of the concept drift, i.e., the number of instances where the concept drift is centered. Each experiment uses the `Interleaved Test Then Train` evaluation method with a frequency $freq. = 1000$.

Paper Experiments

These experiments come from paper [4]. As previously mentioned, the objective is to ensure that the benchmarks results are consistent and allow to consider the calibration of the emulators against the physical boards as reliable. Table 3.5 details them, where `Hoeffding Option Tree` is a regular `Hoeffding Tree` with additional option nodes that allow several tests to be applied, leading to multiple `Hoeffding Trees` as separate paths. `ASHT` or `Adaptive-Size Hoeffding Tree` is derived from the `Hoeffding Tree` algorithm with a maximum number of split nodes or size, such that if the number of split nodes of the `ASHT` is higher than the maximum value, it deletes some nodes to reduce its size. The

	#inst.	algo.	concept drift	generator
Ex. 3.1	100000	NB	1 abrupt drift (w=1, k=50000)	SEA
Ex. 3.2	100000	NB	1 gradual drift (w=20000, k=50000)	RandomTree
Ex. 4.1	100000	LB	3 abrupt drifts (w1,w2,w3=1, k1=25000, k2=50000, k3=75000)	SEA
Ex. 4.2	100000	LB	2 gradual drifts (w1,w3=10000, k1=25000, k3=75000), & 1 abrupt drift (w2=1, k2=50000)	RandomRBF
Ex. 5	100000	NB	Incremental drift	RandomRBF
Ex. 7.1	100000	HT	1 abrupt drift (w=1, k=50000)	SEA
Ex. 7.2	1000000	HT	2 gradual drifts (w1=200000, w2=50000, k1=350000, k2=600000)	SEA
Ex. 7.3	1000000	HT	3 abrupt drifts (w1,w2,w3=1, k1=250000, k2=500000, k3=750000)	Agrawal

Table 3.4: MOA tutorial 5 experiments.

`Random Hoeffding Tree` is simply a random decision tree.

3.4.6. Performance Dimensions

The selected performance dimension is the execution throughput. This measure is independent of the results of the model, and only evaluates the ability of the device to execute it.

The throughput measures the ratio between the number of instances processed and the execution time. Note, however, that it must be averaged or aggregated carefully. Depending on whether the number of instances or the execution time varies between each experiment, it may be necessary to use the arithmetic or harmonic mean. Generally, we can define the throughput as in Eq. 3.1.

$$throughput = \frac{nbr\ of\ instances}{time} \tag{3.1}$$

3.5. Raspberry Pi Experiments with MOA

After running the experiments with the couple of emulators, it appeared that the results were good enough to try out `MOA` on physical board and that the `Raspberry Pi 4b` was outperforming the `Raspberry Pi 3b+` in a way that justifies its extra cost. The exper-

	#inst.	algos.	evaluation	data source
Exp 1	10000000	Hoeffding Tree	Interl. Test Train (freq.=100000)	WaveForm
Exp 2	100000	Hoeffding Tree, Hoeffding Option Tree, Hoeffding Adaptive Tree, ASHT, Random Hoeffding Tree	None	WaveForm
Exp 3	100000	Hoeffding Tree	None	WaveForm, SEA, LED, Hyperplane, RandomTree, census.arff
Exp 4	2000000	Hoeffding Tree	None	census.arff

Table 3.5: Paper [4] Experiments.

iments previously described were then run on a physical board, as well as on a desktop computer, to compare their performance. This step ends the suitability analysis of the Raspberry Pi to run `MOA`. The results obtained along with all the results of the thesis are presented in Chapter 5. They tend to show that it is possible to use the `Raspberry Pi 4b` to run `MOA`, with performance lower than those of a desktop computer, but good enough for a certain number of applications.

Raspberry Pi Configuration

The `Raspberry Pi 4b` used is a version with 4GB of RAM, with a 64-bit `Raspberry Pi OS Bullseye ARM` operating system. Its other components were previously described in Section 3.3.1.

Desktop Configuration

The desktop computer used is a server in the cloud, with 6 CPU cores, 16GB of RAM, and a 64-bit `Ubuntu 20.04 LTS (Focal Fossa)` operating system. The chosen provider is `UpCloud`. With such a configuration, the aim was to set up a high-performance computing configuration, with performance in the order of magnitude of a basic cloud computing solution.

Table 3.6 summarizes the features of the two devices.

	RPi4b	Desktop
CPU	Cortex-A72 1.5GHZ	AMD EPYC 7542 3.4GHz
Nbr of cores	4	6
RAM	4GB	16GB
OS	Raspberry Pi OS Bullseye ARM	Ubuntu 20.04 LTS (Focal Fossa)

Table 3.6: Comparative table of the Raspberry Pi and Desktop used.

3.6. tinyMOA Quantization

At this stage, the suitability analysis is completed. The second axis of development of this thesis consists of the development of a possible improvement of the execution performance of `MOA` on Raspberry Pi, via the quantization of the `tinyMOA` software, which gave birth to `tinyMOA-lite`. Once the quantization done, a series of experiments were conducted to compare the performance of `tinyMOA-lite` to those of `tinyMOA`, still on the Raspberry Pi. Contrary to the experiments carried out on emulators, it was necessary for this part to obtain robust enough results, which could demonstrate the relevance of the quantization of `tinyMOA`, with relatively strong statistical support. For this purpose, a significant number of experiments were conducted, featuring different configurations, and a large number of repetitions for each one, to counterbalance differences due to the operating system, the system load, or other factors possibly inducing noise.

3.6.1. Quantization

As mentioned in Section 2.3.2, within the streaming machine learning framework, the traditional training and production phases are merged, and thus the edge device is responsible for both training and testing. Consequently, the quantization was not applied to the trained weights as for `tf-lite`, but to the whole `tinyMOA` software source code. The source code being in Java, the quantization is performed by transforming all `double` variables into `float` variables.

3.6.2. Datasets

Similarly to the experiments carried out on emulators, these experiments were conducted on synthetic datasets provided by `MOA`, and real datasets, taken from this repository [44].

Synthetic Datasets

The data generators used are the same as those presented in Section 3.4.1.

Real Datasets

To ensure the relevance of the results obtained with synthetically generated datasets, the experiments were also conducted on real datasets, stored as `ARFF` files. These datasets present different data structures and are different one from another, in their characteristics such as the number of features, their length, or their feature types, which have a strong impact on the execution of streaming machine learning models. These datasets are the following:

- `airlines.arff`: 7 features, 539 383 instances, 1 target of 2 classes. The task consists in predicting whether a flight will be delayed or not.
- `covtypeNorm.arff`: 54 features, 581012 instances, 1 target of 7 classes. It contains the forest cover type for 30x30 meter celles obetained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data.
- `elecNormNew.arff`: 8 features, 45312 instances, 1 target of 2 classes. The data is collected from the Austrian Electricity Market. The task consists in predicting whether the price of electricity will increase or decrease, relative to a moving average of the last 24 hours. Each sample refers to a period of 30 minutes.
- `GMSC.arff`: 10 features, 150000 instances, 1 target of 2 classes. This data comes from the `Give Me Some Credit` Kaggle competition. The task consists in predicting whether a loan will be paid back or not.
- `kdd99.arff`: 41 features, 4898431 instances, 1 target of 23 classes. This data comes from the `KDD Cup 1999` competition. The task consists in predicting the type of cyber attack, among 23 different types.
- `spam_corpus.arff`: 39916 features, 9324 instances, 1 target of 2 classes. The task consists in predicting whether an email is a spam or not.

3.6.3. Algorithms

The algorithms used are the same as those presented in Section 3.4.2, namely `HoeffdingTree`, `HoeffdingAdaptiveTree`, `NaiveBayes`, `OzaBag`, `OzaBagAdwin`, and `LeveragingBag`. 10 `HoeffdingTree` base learners were used for `OzaBag`, `OzaBagAdwin`, and `LeveragingBag`.

Algos.	HT, HAT, NB, OB, OBA, LB
Generators	Agrawal, SEA, RBF (Drift), LED (Drift), Random Tree, WaveForm (Drift)
#inst.	500k per experiment
Seeds	10 per key [algorithm, generator, `MOA` version]
Repetitions	30 per key [algorithm, générateur, `MOA` version, seed]

Table 3.7: Group 1 experiments post quantization.

3.6.4. Experiments

The experiments were conducted in three groups. The first group, `Group 1`, proposes configurations without extra concept drift in addition to those produced by data generators (e.g., with RBF Drift, LED Drift, or WaveForm Drift). The second group, `Group 2`, gathers configurations with hand-crafted concept drifts, including gradual and abrupt ones. Finally, `Group 3` includes all the experiments conducted on real datasets. This last group aims at confirming the consistency of the results of `Group 1` and `Group 2`.

Group 1

The whole `Group 1` includes 18 000 experiments (6 algorithms x 10 generators x 10 seeds x 30 repetitions per seed) with both `tinyMOA` and `tinyMOA-lite`, i.e., 36000 experiments in total, see Table 3.7. The objective is to compare, for each key [algorithm, generator], the performance of `tinyMOA` and `tinyMOA-lite`. Each key retrieves 600 experiments, 300 for `tinyMOA` and 300 for `tinyMOA-lite`.

Group 2

`Group 2` consists of 12600 experiments (6 algorithms x 2 generators x 6 concept drift types x 10 seeds x 15 repetitions per seed) for both `tinyMOA` and `tinyMOA-lite`, i.e., 21600 experiments in total. In this case, the objective is to compare, for each key [algorithm, generator, concept drift type] the results obtained with `tinyMOA` and `tinyMOA-lite`. Each key retrieves 300 experiments, 150 for `tinyMOA` and 150 for `tinyMOA-lite`.

Algos.	HT, HAT, NB, OB, OBA, LB
Generator	SEA, Agrawal
Concept drift types	`[1AD]` 1 abrupt drift (k=250000, w=1) `[3AD]` 3 abrupt drifts (w1,w2,w3=1, k1=125000, k2=250000, k3=375000) `[MAD]` Multiple abrupt drifts (w1,w2,w3=1, k1=125000, k2=250000, k3=375000) `[1GD]` 1 gradual drift (w=100000, k=250000) `[2GD]` 2 gradual drifts (w1=50000, w2=25000, k1=175000, k2=300000) `[MMD]` Multiple mixed drifts (w1=50000, w2=1, w3=50000, k1=125000, k2=250000, k3=375000)
#inst.	500k per experimentation
Seeds	10 per key [algorithm, generator, concept drift type, `MOA` version]
Repetitions	15 per key [algorithm, generator, concept drift type, `MOA` version, seed]

Table 3.8: Group 2 Experiments post quantization.

Details are provided in Table 3.8, with w the width of the drift and k the position of the center of the drift. The difference between `[MAD]` and `[3AD]` as concept drift types, although the structure of the drifts is the same, resides in the order of the distributions used.

Group 3

`Group 3` includes experiments conducted on six real datasets, some with highly complex data structures composed of hundreds of features, and others with simple structures and larger numbers of data points. For each real dataset, the six usual algorithms are run, with a certain number of repetitions to limit noise in the results, see Table 3.9.

Seeds, Repetitions and Permutations

Seeds are numbers used by pseudo-random algorithms, themselves used by the data generators. They correspond to the input of the pseudo-random number generation function. Using multiple seeds for the same experiment configuration limits the bias related to data generation itself, which is emphasized when the generated data with the default seed leads to an execution that is not representative of the true performance of the model.

Repeating the same experiments multiple times also limits the bias, related this time to external factors, in particular those due to the operating system, e.g., the load of the system, memory availability, etc.

Dataset	Algos.	Repetitions
airlines.arff	HT HAT NB, OB OBA LB	30 tinyMOA and 30 tinyMOA-lite 15 tinyMOA, 15 tinyMOA-lite
covtypeNorm.arff	HT HAT NB, OB OBA LB	30 tinyMOA and 30 tinyMOA-lite 15 tinyMOA, 15 tinyMOA-lite
elecNormNew.arff	HT HAT NB OB OBA LB	30 tinyMOA and 30 tinyMOA-lite
GMSC.arff	HT HAT NB OB OBA LB	30 tinyMOA and 30 tinyMOA-lite
kdd99.arff	HT HAT NB OB OBA LB	15 tinyMOA and 15 tinyMOA-lite
spam_corpus.arff	HT HAT NB OB OBA LB	15 tinyMOA and 15 tinyMOA-lite

Table 3.9: Group 3 Experiments post quantization.

Between each repetition, cached data is flushed to limit the impact of previous executions. All permutations of order between tinyMOA and tinyMOA-lite are also implemented to limit these shared memory issues.

Recap

Two groups of experiments with synthetically generated data are carried out to compare the performance of tinyMOA and tinyMOA-lite. Group 1 includes experiments without added concept drift. Group 2 includes experiments with added concept drifts of different types. Finally, Group 3 consists of experiments on a bunch of six real datasets, to ensure the consistency of the previous results.

3.6.5. Outliers Detection

The first stage of data processing, after collecting the results, consists in removing outliers along the throughput dimension. Outliers are data points that probably do not belong to the population, and may distort further analyses. They can be caused by external factors to the algorithm, related to the execution environment. The aim is not simply to keep the average results but to keep all the results that probably belong, to some extent, to the population. In other words, an important aspect of this step is to carefully review the data so as not to delete any data that comes from the population. Several base methods were considered: N min and max, the Boxplot method, and the Z-score method.

N Min and Max

`N min and max` tackles the outliers problem by removing the N best and N worst results. This method is simple to implement but does not guarantee the removal of outliers or the preservation of results that probably belong to the population.

Boxplot Method

The `Boxplot` method consists in removing the values that are lower than $Q1 - 1.5 * IQR$ and higher than $Q3 + 1.5 * IQR$, where $Q1$ and $Q3$ are quartiles 1 (25%) and 3 (75%), and IQR is the interquartile range. This method is more robust than the previous one even though it still does not guarantee the removal of outliers nor the preservation of results that probably belong to the population of results. This method works well for data following a normal distribution but can be unsuitable for other distributions, for example, if there are several distinct clusters in the data. This method was often too restrictive by removing too many results (around 3%), mainly because part of the results were not gaussian-like distributed.

Z-Score Method

The `Z-score` method consists in removing the results that are not included in the interval around the mean, defined by :

$$[\bar{x} - 3 * \sigma, \bar{x} + 3 * \sigma]$$

where $\bar{x}$ is the mean, and σ the standard deviation. This method is based on the assumption that the data follows a normal distribution, in which case this interval includes 99.7% of the theoretical population.

Adopted Method

The finally adopted method to remove outliers is a mix of the `N min and max` and `Z-Score` methods, detailed in Algorithm 4.4. A `Shapiro-Wilk` statistical test is first executed to determine whether the data follows a normal distribution (null-hypothesis), defined by Eq. 3.2,

$$W = \frac{(\sum_{i=1}^{n} a_i x_i)^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{3.2}$$

where x_i is the i-th value, $\bar{x}$ the mean, and a_i a coefficient given by a table according to the sample size. W is then compared to a table of values which returns the thresholds at 5% to reject or not the null-hypothesis. The p-value can also be calculated with

	Algorithms
Hoeffding Trees	Hoeffding Tree, Hoeffding Adaptive Tree
Naive Bayes	Naive Bayes
Ensemble Methods	OzaBag, OzaBag Adwin, Leveraging Bagging

Table 3.10: Algorithm subgroups considered for aggregation.

dedicated algorithms (e.g., the `Royston` algorithm). Above a certain threshold, the data is considered as normally distributed and is filtered with the `Z-score` method to set parameter N for the final step. Then `N min and max` method is applied at the very end, with a default value for parameter N if the previous condition was not met, so that the couple of samples from `tinyMOA` and `tinyMOA-lite` have the same size, based on the one with the most detected outliers. This method leads to the removal of 0.5% of the data.

3.6.6. Performance Dimensions

Accuracy, throughput, and memory consumption were chosen as performance dimensions for these experiments. Some additional statistics also provide a more accurate view of the limitations of the results. The accuracy is a dimension of the model performance while the throughput and the memory consumption are meta dimensions that give insights into the execution itself.

Results are aggregated at different levels to provide a complete analysis at different levels of granularity. They can be aggregated by individual experiment configuration, by algorithm, or by groups of algorithms where algorithms are mainly grouped into three subgroups, as described in Table 3.10.

Accuracy and Statistics

Accuracy, as presented in Section 2.1.4, is the metric that calculates the proportion of correct predictions. Within the streaming machine learning framework, this measure must be updated incrementally. An incremental formula is given by Eq. 3.3, where y_i is the prediction of the i-th instance, and $accuracy_i$ is the accuracy after the i-th instance.

$$accuracy_n = accuracy_{n-1} + \frac{1}{n} \cdot (y_n - accuracy_{n-1}) \tag{3.3}$$

The accuracy aggregations are performed as follow:

- Extract for each experiment configuration the mean accuracy of `tinyMOA` and `tinyMOA-lite`. A configuration is identified by the key [generator, algorithm, seed] for `Group 1`, the key [generator, algorithm, concept drift type, seed] for `Group 2` and the key [dataset, algorithm] for `Group 3`.
- Calculate the desired comparative statistics, such as the average gain, the Pearson correlation coefficient, the mean absolute error, etc., for each pair of results.
- Aggregate gains and statistics with the desired granularity.

The aggregated results are therefore averages of ratios, not ratios of averages. Gains are first computed per key and then averaged. This choice was motivated by the desire to give each experiment configuration the same importance, independently from absolute values, and also to avoid compensations between results.

Individual key accuracy gain is calculated as stated in Eq. 3.4.

$$Gain_{accuracy} = \frac{\text{Accuracy}_{tinyMOA-lite}}{\text{Accuracy}_{tinyMOA}} - 1 \tag{3.4}$$

It gives an idea of the loss of precision induced by the quantization.

The first additional statistic is the `Pearson` correlation coefficient, which measures the linear correlation between two random variables. The formula is as in Eq. 3.5, where x_i and y_i are the i-th values of the two variables, and $\bar{x}$ and $\bar{y}$ their mean. The value of r is between -1 and 1, and indicates the linear correlation between the two variables.

$$r_{pearson} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{3.5}$$

The second one is the mean absolute error (`MAE`) and is calculated as in Eq. 3.6, where x_i and y_i are the i-th values of the two variables. The value of MAE is between 0 and 1, and indicates the average of the absolute error between the two variables.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i| \tag{3.6}$$

Finally are computed the average of the maximums absolute errors, which simply averages the largest absolute error in precision across all experiment configurations, and the maximum of the maximums absolute errors, which returns the largest absolute error across all experiment configurations.

Throughput and Statistics

For the throughput dimension, aggregations follow a similar pattern as for the accuracy, except that seeds are not considered individually. The process is as follows:

- Extract for each experiment configuration the throughputs of `tinyMOA-lite` and `tinyMOA`. A configuration is identified by the key [generator, algorithm] for `Group 1`, the key [generator, algorithm, concept drift type] for `Group 2`, and the key [dataset, algorithm] for `Group 3`. Each key retrieves the results of the corresponding repetitions and seeds.
- Apply the adopted method for the removal of the outliers.
- Calculate the desired statistics to compare both versions, such as the average quantization gain or the Welch's t-test p-value.
- Aggregate gains and statistics with the desired granularity.

This time, averages must be computed as harmonic means because throughputs are themselves ratio, where the numerator (number of instances) is common to all experiment repetitions and seeds for a configuration key. The harmonic mean is computed as in Eq. 3.7, where x_i is the i-th throughput value, and n the number of throughputs. The aggregated results are also averages of ratios.

$$\bar{x} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \tag{3.7}$$

Individual key throughput gain is calculated as follows in Eq. 3.8.

$$Gain = \frac{\text{Throughput}_{tinyMOA-lite}}{\text{Throughput}_{tinyMOA}} - 1 \tag{3.8}$$

Ideally, the gain is positive indicating a performance improvement by the quantized version.

In addition to the average gain are calculated:

- The minimum gain across the seeds of a configuration key [algorithm, generator] for `Group 1`, [algorithm, generator, concept drift type] for `Group 2` and [algorithm, dataset] for `Group3`.
- The maximum gain across the seeds of a configuration key.
- The Welch's t-test p-value, which is a statistical test with a null hypothesis that

states that two samples have the same mean. This variant of the Student's t-test allows to compare two samples with unequal variance. The objective is to ensure that the differences in performance between `tinyMOA` and `tinyMOA-lite` are statistically significant.

Memory Consumption and Statistics

To measure memory consumption, `MOA` provides in its execution report a measure in RAM.Hours which multiplies the execution time in nanoseconds and the memory consumption in MB. The results are aggregated in the same way as for throughputs, detailed in the previous Section 3.6.6, except that the arithmetic mean is initially used instead of the harmonic mean to average results across repetitions and seeds. The gain in RAM.Hours is calculated as in Eq. 3.9.

$$Gain = \frac{\text{RAM.Hours}_{tinyMOA-light}}{\text{RAM.Hours}_{tinyMOA}} - 1 \tag{3.9}$$

Ideally, the gain should be negative, meaning a gain in performance for the quantized version. In addition to the average gain are calculated:

- The minimum gain (the worst, i.e., the largest in absolute value) among the seeds of a configuration key.
- The maximum gain (the best, i.e. the smallest in absolute value) among the seeds of a configuration key.

3.7. tinyMOA Quasi-quantization

The last task of this thesis, also related to the improvement study and the quantization part, consists in proposing a quasi-quantized version of `tinyMOA`, which with a minimum of 64-bit variables proposes a perfectly similar execution to the non-quantized version. The objective is to keep the advantages of quantization, i.e., a lighter execution while generating the very same solution as the non-quantized version. This problem mainly concerns decision trees and adaptive models. As in streaming machine learning the execution is sequential, as soon as the quantized version makes a different decision (choice of the splitting variable, concept drift detection etc.) from the non-quantized version, the execution may diverge and the solution proposed by the quantized version, in the end, is no longer the same as the one proposed by the non-quantized version.

For adaptive algorithms, the impact of the parameter `delta` of `ADWIN` was a possible devel-

opment axis. The idea is to compare the results of adaptive models for different values of `delta` and to see if it is possible to obtain more similar results between the quantized version and the original version with the optimal `delta`. In a second phase, a quasi-quantized version of `tinyMOA` was compiled, after backtracking all causes of differences between the quantized and the non-quantized version, for models using `HoeffdingTree` and/or `ADWIN`. The quasi-quantized version, with a minimum of 64-bit variables, must propose a perfectly identical execution to the non-quantized version for the algorithms subject to differences in the solution structure, i.e., `HoeffdingTree`, `HoeffdingAdaptiveTree`, `OzaBag`, `OzaBagAdwin` and `LeveragingBag`.

3.7.1. Delta Tuning

For adaptive models, the results in terms of accuracy are affected by the number of detections triggered by the concept drift detector, `ADWIN` in this case. Indeed, adaptive models proceed to heavy modifications of their structure when a concept drift is detected. Streaming machine learning model execution is sequential, thus a modification results in a different execution from the non-quantized version. The `delta` parameter of `ADWIN` is a parameter controlling the concept drift detection sensitivity.

```
1  private boolean blnCutexpression(int n0, int n1, double u0, double u1,
2  double v0, double v1, double absvalue, double delta) {
3      int n = getWidth();
4      double dd = Math.log(2 * Math.log(n) / delta); // High delta => small dd
5      double v = getVariance();
6      double m = ((double) 1 / ((n0 - mintMinWinLength + 1))) +
7      ((double) 1 / ((n1 - mintMinWinLength + 1)));
8      double epsilon = Math.sqrt(2 * m * v * dd) + (double) 2 / 3 * dd * m;
9      // Small dd => small epsilon
10     boolean result = (Math.abs(absvalue) > epsilon);
11     return result; // Small epsilon => more results being True
12 }
```

Listing 3.1: Code snippet of the ADWIN concept drift detector.

As shown in Listing 3.1, the higher the value of `delta`, the more sensitive the concept drift detector. Starting from this observation, the idea is to compare the results obtained with the quantized version and the original version of `tinyMOA`, tuned with the optimal `delta` for each experiment. The optimality criterion is the mean accuracy, and the optimal `delta` value is obtained by a grid search, where the values of `delta` are tested on an interval of values with a certain step.

Algos.	HT, HAT, NB, OB, OBA, LB
Generators	Agrawal, SEA, RBF, LED, Random Tree, WaveForm
#inst.	100k per experiment
Seeds	5 per key [algorithm, generator, `MOA` version]
Repetitions	5 per key [algorithm, generator, MOA version, seed]

Table 3.11: Experiments without added concept drift after quasi-quantization.

3.7.2. Quasi-quantization

For the quasi-quantization phase, the main goal is to minimize the use of 64-bit variables. To determine the causes of differences between the quantized and the non-quantized version, an example with `LeveragingBag` was analyzed, and the causes of differences were backtracked using static lists and counters, useful for observing the execution flow in detail. It turned out that in order to get two identical executions, different modifications of the source code were necessary. An important number of 64 bits variables had to be reintroduced. In summary, the following modifications were necessary:

- the `LeveragingBag` class, for calculations related to the *Poisson* distribution, used for online bagging,
- of the `HoeffdingTree` class, for all the computations related to the split decisions and to the choices at `Naive Bayes Adaptive` leaves,
- of the `ADWIN` class, for computations related to concept drift detection,
- of the `NaiveBayes` class, for computations related to numerical attributes using Gaussian distributions.

Quasi-quantization was thus only possible at the cost of many 64-bit variables, resulting in a new compiled file `tinyMOA-quasi-lite`. Experiments were conducted to assess whether some of the benefits of quantization were still valid. Two groups of experiments were conducted, following the same principle as previously. One group focuses on experiments without added concept drift, which are detailed in Table 3.11, and the other group focuses on experiments with added concept drift, detailed in Table 3.12. Each experiment is run with `tinyMOA` and `tinyMOA-quasi-lite`. w is the width of the drift and k the position of the center of the drift.

Algos.	HT, HAT, NB, OB, OBA, LB
Generator	SEA, Agrawal
Concept drift types	`[1AD]` 1 abrupt drift (k=250000, w=1) `[MAD]` Multiple abrupt drifts (w1,w2,w3=1, k1=125000, k2=250000, k3=375000) `[1GD]` 1 gradual drift (w=100000, k=250000) `[MMD]` Multiple mixed drifts (w1=50000, w2=1, w3=50000, k1=125000, k2=250000, k3=375000)
#inst.	100k per experimentation
Seeds	5 per key [algorithm, generator, concept drift type, `MOA` version]
Repetitions	5 per key [algorithm, generator, concept drift type, `MOA` version, seed]

Table 3.12: Experiments with added concept drift after quasi-quantization.

4 | Implementation

This chapter focuses on the main scripts, algorithms, and leveraging technologies of this thesis. It follows the same order as the previous chapter 3. The first section focuses on the Raspberry Pi simulation part, including the emulators themselves and the conducted experiments using the benchmarks and `MOA`. The second section focuses on the quantization of `tinyMOA`, including the conducted experiments, the outliers detection methods and the results aggregation strategies. Finally, the third and last section focuses on the quasi-quantization part, including delta fine-tuning and a couple of important scripts for the extraction of results. Only the main scripts or algorithms are presented for each section.

4.1. Raspberry Pi Simulation Implementation

4.1.1. Emulators

The first task consisted in implementing the emulators of the `Raspberry Pi 2b`, `Raspberry Pi 3b+`, and `Raspberry Pi 4b`, using `QEMU` as leveraging technology. To automate the deployment of such emulators, `bash` scripts are used along with `Docker` containers. `Docker` containers isolate the emulators from the host system along with their dependencies, to ensure that the environment is the same at each run. This solution was adopted since the `Raspberry Pi 2b` emulator leverages `QEMU v2.9.0` while the `Raspberry Pi 3b+` and `Raspberry Pi 4b` emulators leverage `QEMU v6.2.0`, and each version of `QEMU` requires a different set of dependencies. `Docker` containers provide a solution to this problem at a low cost in terms of performance.

Each emulator is associated with a `Dockerfile`, which contains the instructions to build the image of the corresponding container, as shown with the example of Listing 4.1, and a `bash` script to download the operating system, download the right version of `QEMU`, process the operating system `iso` file with `qemu-img`, and run `Docker`, as shown with the example of Listing 4.2. After the execution of each script, the corresponding emulator is ready for use, and the emulated terminal is directly accessible.

```
1  FROM ubuntu:20.04
2  ARG DEBIAN_FRONTEND=noninteractive
3
4  # Create useful directories
5  RUN mkdir /home/raspi3b
6  RUN mkdir /home/raspi3b/boot3b
7
8  # Copy in the container qemu, the operating system and the useful files for qemu
9  COPY ./src/raspi3b/image-raspi3b-resized-8G.img /home/raspi3b/
10 COPY ./src/raspi3b/qemu-6.2.0.tar.xz /home/raspi3b/
11 COPY ./src/raspi3b/boot3b/kernel8.img /home/raspi3b/boot3b/
12 COPY ./src/raspi3b/boot3b/bcm2710-rpi-3-b-plus.dtb /home/raspi3b/boot3b/
13
14 RUN apt-get update -y
15 RUN apt-get install xz-utils -y
16
17 # Unzip qemu
18 WORKDIR /home/raspi3b
19 RUN tar -xvJf ./qemu-6.2.0.tar.xz
20
21 # Install dependencies
22 RUN apt-get install -y gcc make ninja-build pkg-config libglib2.0-dev
23 RUN apt-get install -y libpixman-1-dev libcairo2-dev libpango1.0-dev
24 libjpeg8-dev libgif-dev;
25 RUN apt-get install -y python libfdt-dev;
26 RUN apt-get install -y libgtk2.0-dev;
27 RUN apt-get update -y && apt-get upgrade -y;
28
29 # Build and compile qemu
30 WORKDIR /home/raspi3b/qemu-6.2.0
31 RUN mkdir build
32 WORKDIR /home/raspi3b/qemu-6.2.0/build
33 RUN ../configure --target-list=aarch64-softmmu,arm-softmmu
34 RUN make -j `nproc`
35
36 WORKDIR /home/raspi3b
```

Listing 4.1: Dockerfile used for the Raspberry Pi 3b+ emulation.

```
if  [[ $1 = "-b" ]]; then # Build the docker image
    # Create useful directories
    mkdir src
    mkdir src/raspi3b

    # Download the Raspberry Pi operating system
    apt-get update -y
    apt-get install -y wget unzip
    wget "https://downloads.raspberrypi.org/raspios_arm64/images/ \
    raspios_arm64-2022-01-28/2022-01-28-raspios-bullseye-arm64.zip" -P ./src/raspi3b
    cd ./src/raspi3b

    # Unzip the operating system
    unzip 2022-01-28-raspios-bullseye-arm64.zip
    IMAGE=`find ./ -name '2*.img'`

    # Mount the operating system and extract the kernel and the device tree
    losetup -f --show -P $IMAGE > loop.txt
    LOOPDIR=$(cat loop.txt)
    LOOPDIRMODIF=$LOOPDIR"p1"
    mkdir /mnt/rpi
    mount $LOOPDIRMODIF /mnt/rpi
    mkdir ./boot3b
    cp -r /mnt/rpi/* ./boot3b
    umount /mnt/rpi
    losetup -d $LOOPDIR

    # Resize the operating system with qemu-img
    apt-get install -y qemu-utils
    cp $IMAGE ./image-raspi3b-resized-8G.img
    qemu-img resize "image-raspi3b-resized-8G.img" 8G
    wget https://download.qemu.org/qemu-6.2.0.tar.xz
    cd ../../
    export DOCKER_BUILDKIT=1

    # Build the docker image
    docker build -t raspi3b -f Dockerfile-raspi3b .
    docker run -p 2222:2222 --name raspi3b -it -d raspi3b

else # Just start the container
    docker start raspi3b;
fi

# Execute the QEMU command within the container to start the emulator
docker exec -it raspi3b /bin/bash -c \
"./qemu-6.2.0/build/aarch64-softmmu/qemu-system-aarch64 -m 1024 -M raspi3b \
-cpu cortex-a53 -smp 4 -kernel ./boot3b/kernel8.img
-dtb ./boot3b/bcm2710-rpi-3-b-plus.dtb \
-drive file=./image-raspi3b-resized-8G.img,if=sd,format=raw \
-append 'rw earlycon=pl011,0x3f201000 console=ttyAMA0 loglevel=8 \
root=/dev/mmcblk0p2 fsck.repair=yes net.ifnames=0 rootwait memtest=1' \
-device usb-net,netdev=net0 -netdev user,id=net0,ipv4=on,hostfwd=tcp::2222-:22 \
-nographic"
```

Listing 4.2: Deployment bash script for the Raspberry Pi 3b+ emulator.

4.1.2. Experiments with MOA

The experiments conducted with `MOA` on the emulators come from the official `MOA` tutorials and the publication [4]. Thus, there are two groups of experiments, and three distinct parts in the implementation: tutorial 1, tutorial 5, and the paper experiments. For each part, two `bash` scripts are used. One for the installation of `MOA` along with the calls to the different experiments. This script is the one executed by the user and an example is given in Listing 4.3. The other script includes the experiments themselves, implemented with `MOA` (or `tinyMOA`) commands. An example is shown in Listing 4.4.

```
while getopts i: option
do
    case "${option}"
        in
        i)install=${OPTARG};; # Installation option
    esac
done

if  [[ $install = "full" ]]; then
  mkdir Tutorials1
  # Download MOA installation script from the Github repository
  wget "https://raw.githubusercontent.com/franckdeturchedura/tesina-SML-raspberryPI/ \
  main/2_MOA/scripts/installMOA.sh" -O installMOA.sh
  apt-get install -y dos2unix
  dos2unix installMOA.sh
  bash install-MOA.sh
  # Download experiments execution script from the Github repository
  wget "https://raw.githubusercontent.com/franckdeturchedura/tesina-SML-raspberryPI/ \
  main/2_MOA/scripts/Tutorials1/exercices.sh" -O Tutorials1/exercices.sh

elif  [[ $install = "jre" ]]; then # Just install the JRE
  apt-get update
  apt-get install -y default-jre

else
    echo "Execution within cloned repository";
fi

# Execute the 5 experiments of Tutorial 1
cd Tutorials1
apt-get install -y dos2unix
dos2unix exercices.sh
read -p "Enter your device name : " DEVICENAME
echo $DEVICENAME | bash ./exercices.sh -n 1
echo $DEVICENAME | bash ./exercices.sh -n 2
echo $DEVICENAME | bash ./exercices.sh -n 3
echo $DEVICENAME | bash ./exercices.sh -n 4
echo $DEVICENAME | bash ./exercices.sh -n 6
```

Listing 4.3: Bash script for MOA installation and experiments calls for Tutorial 1.

```
# Create all the necessary directories and variables.
cd ../src
MOADIR='moa'
cd $MOADIR
MOAJAR='./moa.jar' # or ./tinyMOA.jar
SIZEOFFAG=`find ./ -type f -name 'sizeofag*.jar'`
read -p "Enter your device name : " DEVICENAME
BASEDIR='../../Tutorials1'
mkdir $BASEDIR/results
mkdir $BASEDIR/results/$DEVICENAME

while getopts n: option
do
    case "${option}"
        in
        n)number=${OPTARG};; # Number of the experiment to execute
    esac
done

if  [[ $number = 1 ]]; then # Execute the corresponding MOA commands
    java -cp $MOAJAR -javaagent:$SIZEOFFAG moa.DoTask \
    "EvaluateInterleavedTestThenTrain -l trees.HoeffdingTree \
    -s generators.RandomTreeGenerator \
    -i 1000000 -f 10000" > $BASEDIR/results/$DEVICENAME/ex1_HoeffdingTree.csv
    # Results are directly saved as a csv file within the results directory.
    java -cp $MOAJAR -javaagent:$SIZEOFFAG moa.DoTask \
    "EvaluateInterleavedTestThenTrain -l bayes.NaiveBayes \
    -s generators.RandomTreeGenerator \
    -i 1000000 -f 10000" > $BASEDIR/results/$DEVICENAME/ex1_NaiveBayes.csv

elif  [[ $number = 2 ]]; then
    java -cp $MOAJAR -javaagent:$SIZEOFFAG moa.DoTask \
    "EvaluateInterleavedTestThenTrain -l trees.HoeffdingTree \
    -s generators.RandomTreeGenerator \
    -i 1000000 -f 10000" > $BASEDIR/results/$DEVICENAME/ex2_IntervealedTestThenTrain.csv

    # ... and so on until the end of the tutorial, i.e., 5 experiments

else
    echo "nothing to execute";
fi
```

Listing 4.4: Bash script with Tutorial 1 experiments.

The experiments conducted, after this part on the physical `Raspberry Pi 4b`, are the same as the ones conducted on the emulators. Consequently, the same `bash` scripts were used.

4.2. Quantization Implementation

In the quantization part, many components were implemented because it was important to obtain statistically robust results to confirm or not the potential gain provided by the quantization. That is why in addition to the experiments, tools such as outliers detection methods and aggregation strategies were implemented.

4.2.1. Experiments

The first step was to run the `MOA` experiments, with a more complex procedure than the one used for the emulators. Each experiment is repeated multiple times, with different seeds for synthetic generators, and different permutations between `tinyMOA` and `tinyMOA-lite` to avoid any conflict or optimization between runs. There are three groups of experiments, each one associated with its own execution flow.

The three groups of experiments have each two `bash` scripts associated, used in the same way as for the emulators. A script is executed by the user, it ensures that the environment allows to run `MOA` and calls the other script that contains the different `tinyMOA` (or `tinyMOA-lite`) commands.

`Group 1` is the one that focuses on experiments without added concept drifts. The execution pseudocode is detailed in Algorithm 4.1. The algorithms and generators abbreviations used (lines 1-3) refer to the same concepts as described in Chapter 3. The function *createResultPath* (line 13) creates the relative path where results are stored as `csv` files. The path is composed of the components of the configuration, i.e., the algorithm, the generator, the seed, and the repetition counter value. The function *InterlTestTrain* (line 14) refers to the streaming machine learning execution function, that takes as input the model to be trained, the generator, the number of instances to be processed, the evaluation frequency evaluation, and the `tinyMOA` version to use between the quantized and the non-quantized one. The evaluation method is `InterleavedTestThenTrain`.

Algorithm 4.1 Pseudocode of Group 1 experiments execution

```
1: algos ← {HT, HAT, NB, OB, OBA, LB}
2: generators ← {RandomTree, RBF, RBFDrift, LED, LEDDrift,
   WaveForm, WaveFormDrift, Agrawal, Hyperplane, SEA}
3: seeds ← {1456985, 2547893, 3547812, 4865791, 5478963,
   628941, 7854226, 8541277, 9854753, 9658741}
4: permutations ← {moa_lite, moa, moa, moa_lite, moa_lite, moa}
5: for each gen ∈ generators do
6:     for each seed ∈ seeds do
7:         for each algo ∈ algos do
8:             cnt ← 0
9:             while i < loop do
10:                for each moaType ∈ permutations do
11:                    Drop caches
12:                    gen ← set seed to gen
13:                    resultPath ← createResultPath(algo, gen, seed, cnt)
14:                    result ← InterlTestTrain(algo, gen, inst, freq, moaType)
15:                    Write result to csv file resultPath
16:                    i ← i + 1
17:                end for
18:            end while
19:        end for
20:    end for
21: end for
```

`Group 2` is the one that focuses on experiments with hand-crafted concept drifts. The execution pseudocode is provided in Algorithm 4.2. This time there are only two generators, and an additional loop to switch between the different concept drift types. The functions are the same as the ones described for `Group 1`.

Algorithm 4.2 Pseudocode of Group 2 experiments execution

```
1: algos ← {HT, HAT, NB, OB, OBA, LB}
2: generators ← {Agrawal, SEA}
3: seeds ← {1456985, 2547893, 3547812, 4865791, 5478963,
   628941, 7854226, 8541277, 9854753, 9658741}
4: conceptDrifts ← {1AD, 1GD, 2GD, 3AD, MAD, MMD}
5: permutations ← {moa_lite, moa, moa, moa_lite, moa_lite, moa}
6: for each gen ∈ generators do
7:     for each seed ∈ seeds do
8:         for each algo ∈ algos do
9:             for each cd ∈ conceptDrifts do
10:                cnt ← 0
11:                while i < loop do
12:                    for each moaType ∈ permutations do
13:                        Drop caches
14:                        gen ← set seed to gen
15:                        resultPath ← createResultPath(algo, gen, seed, cd, cnt)
16:                        result ← InterlTestTrain(algo, gen, cd, inst, freq, moaType)
17:                        Write result to csv file resultPath
18:                        i ← i + 1
19:                    end for
20:                end while
21:            end for
22:        end for
23:    end for
24: end for
```

`Group 3` focuses on experiments with real datasets. The execution pseudocode is detailed in Algorithm 4.3. The principle is the same as for the experiments of `Group 1`, but generators are replaced with real datasets. The functions used are still the same and the datasets (line 2) are the one introduced in Chapter 3.

Algorithm 4.3 Pseudocode of Group 3 experiments execution

```
1: algos ← {HT, HAT, NB, OB, OBA, LB}
2: datasets ← {airlines, covTypeNom, elecNormNew, GMSC, kdd99, spam_corpus}
3: permutations ← {moa_lite, moa, moa, moa_lite, moa_lite, moa}
4: for each dataset ∈ datasets do
5:     for each algo ∈ algos do
6:         cnt ← 0
7:         while i < loop do
8:             for each moaType ∈ permutations do
9:                 Drop caches
10:                resultPath ← createResultPath(algo, dataset, cnt)
11:                result ← InterleavedTestTrain(algo, dataset, inst, freq, moaType)
12:                Write result to csv file resultPath
13:                i ← i + 1
14:            end for
15:        end while
16:    end for
17: end for
```

4.2.2. Results Processing

Outliers Detection

Once the results were collected, the next step was to process and analyze them. To remove outliers, a mix between two methods was used, as shown in Algorithm 4.4. To summarize, the `Z-Score` method, presented in Algorithm 4.5, is used when a statistical test shows evidence that the sample follows a normal distribution. In this case, all points that are more than 3 standard deviations away from the mean are considered outliers. Otherwise, a default number of outliers n is assigned for the last step. Finally, the `n min and max` method, see Algorithm 4.6, is used to make both the quantized and non-quantized samples have the same size, based on the one with the most detected outliers. The $configurations$ are the different combinations of hyperparameters that define the experiments. Each $configuration$ is a key [algorithm, generator] for `Group 1`, [algorithm, generator, concept drift type] for `Group 2`, and [algorithm, dataset] for `Group 3`. The $retrieveResults$ function retrieves all the results for a given $configuration$ and a given version of `tinyMOA` (either `tinyMOA` or `tinyMOA-lite`). The number of retrieved results depends on the number of repetitions and seeds for a given $configuration$ and `tinyMOA` version. $results_{tinyMOA}$ and $results_{tinyMOA-lite}$ are sets of throughputs, which is the

considered dimension for outliers removal. The function $testShapiroWilk$ returns the p-value of the statistical `Shapiro-Wilk`test, which null-hypothesis states that the sample follows a normal distribution. The threshold value 1e−5 was chosen empirically as the value that rejects for sure only the non-gaussian distributed samples for these experiments.

Algorithm 4.4 Pseudocode of the outliers detection method

function REMOVEOUTLIERS($configurations$,n)
  $r \leftarrow n$
  for each $configuration \in configurations$ **do**
    $results_{tinyMOA} \leftarrow retrieveResults(configuration, tinyMOA)$
    $results_{tinyMOA-lite} \leftarrow retrieveResults(configuration, tinyMOA - lite)$
    if $testShapiroWilk(results_{tinyMOA}) > 1e{-}5$ **then**
      $r_{tinyMOA} \leftarrow |\text{ZSCORE}(results_{tinyMOA})|$
    else
      $r_{tinyMOA} \leftarrow r$
    end if
    if $testShapiroWilk(results_{tinyMOA-lite}) > 1e{-}5$ **then**
      $r_{tinyMOA-lite} \leftarrow |\text{ZSCORE}(results_{tinyMOA-lite})|$
    else
      $r_{tinyMOA-lite} \leftarrow r$
    end if
    $r \leftarrow max(r_{tinyMOA}, r_{tinyMOA-lite})$
    $outliers_{tinyMOA} \leftarrow \text{NMINMAX}(results_{tinyMOA}, r_{tinyMOA})$
    $outliers_{tinyMOA-lite} \leftarrow \text{NMINMAX}(results_{tinyMOA-lite}, r_{tinyMOA-lite})$
    Remove $outliers_{tinyMOA}$ from $results_{tinyMOA}$
    Remove $outliers_{tinyMOA-lite}$ from $results_{tinyMOA-lite}$
  end for
  Return $results_{tinyMOA}$, $results_{tinyMOA-lite}$
end function

In Algorithm 4.5, which details the `Z-Score` algorithm, the function `mean` computes the mean of the sample and the function `std` computes the standard deviation of the sample. In the `N min and max` Algorithm 4.6, the function $min(results, n)$ returns the value of the n-th smallest value of the sample. Similarly, the function $max(results, n)$ returns the value of the n-th largest value of the sample.

Algorithm 4.5 Pseudocode of the Z-score outliers detection method

```
1: function zScore(results)
2:     threshold_inf ← mean(results) − 3 ∗ std(results)
3:     threshold_sup ← mean(results) + 3 ∗ std(results)
4:     outliers ← ∅
5:     for each result ∈ results do
6:         if result < threshold_inf or result > threshold_sup then
7:             outliers ← outliers ∪ result
8:         end if
9:     end for
10:    Return outliers
11: end function
```

Algorithm 4.6 Pseudocode of the N min and max outliers detection method

```
1: function nMinMax(results,n)
2:     threshold_inf ← min(results, n)
3:     threshold_sup ← max(results, n)
4:     outliers ← ∅
5:     for each result ∈ results do
6:         if result ≤ threshold_inf or result ≥ threshold_sup then
7:             outliers ← outliers ∪ result
8:         end if
9:     end for
10:    Return outliers
11: end function
```

Aggregation Strategies

To aggregate the results, considering the three chosen performance dimensions (accuracy, memory, and throughput), from each individual repetition to the overall performance, specific strategies were set up. *granularity* defines which experiment configurations must be aggregated, such that:
$granularity \in \{All, HoeffdingTrees, NaiveBayes, EnsembleMethods\}$. Details about configurations for each dimension are provided in Section 3.4.6. Algorithm 4.7 shows the accuracy aggregation strategy. $retrieveConfigurations$ retrieves all the configurations for a certain *granularity*, while $retrieveExperiments$ retrieves all the experiments associated with a configuration, which cardinality depends on the number of seeds and/or repetitions.

Algorithm 4.7 Pseudocode of the accuracy aggregation strategy

1: $meanAccuracies_{tinyMOA} \leftarrow \emptyset$
2: $meanAccuracies_{tinyMOA-lite} \leftarrow \emptyset$
3: $meanAccuracyGains \leftarrow \emptyset$
4: $configurations \leftarrow retrieveConfigurations(granularity)$
5: **for each** $configuration \in configurations$ **do**
6: $experiments \leftarrow retrieveExperiments(configuration)$
7: **for each** $exp \in experiments$ **do**
8: $meanAccuracies_{tinyMOA} \leftarrow exp.meanAccuracy_{tinyMOA}$
9: $meanAccuracies_{tinyMOA-lite} \leftarrow exp.meanAccuracy_{tinyMOA-lite}$
10: **end for**
11: $gain \leftarrow \frac{Mean(meanAccuracies_{tinyMOA-lite})}{Mean(throughputs_{tinyMOA})}$
12: $meanAccuracyGains \leftarrow meanAccuracyGains \cup gain$
13: **end for**
14: $aggregatedResult \leftarrow Mean(meanAccuracyGains)$

Algorithm 4.8 shows the throughput aggregation strategy. The last performance dimension, memory consumption in RAM.Hours, uses this same algorithm, with arithmetic means instead of harmonic means.

Algorithm 4.8 Pseudocode of the throughput aggregation strategy

1: $throughputs_{tinyMOA} \leftarrow \emptyset$
2: $throughputs_{tinyMOA-lite} \leftarrow \emptyset$
3: $throughputGains \leftarrow \emptyset$
4: $configurations \leftarrow retrieveConfigurations(granularity)$
5: **for each** $configuration \in configurations$ **do**
6: $experiments \leftarrow retrieveExperiments(configuration)$
7: **for each** $exp \in experiments$ **do**
8: $throughputs_{tinyMOA} \leftarrow exp.throughput_{tinyMOA}$
9: $throughputs_{tinyMOA-lite} \leftarrow exp.throughput_{tinyMOA-lite}$
10: **end for**
11: $gain \leftarrow \frac{HarmonicMean(throughputs_{tinyMOA-lite})}{HarmonicMean(throughputs_{tinyMOA})}$
12: $throughputGains \leftarrow throughputGains \cup gain$
13: **end for**
14: $aggregatedResult \leftarrow Mean(throughputGains)$

4.3. Quasi-quantization Implementation

For the quasi-quantization part, the first step was to fine-tune the *delta* hyperparameter for `ADWIN`, to compare the results between the quantized and non-quantized versions for the optimal delta. The grid search function takes as input the experiment configurations for which the optimal *delta* is to be found and returns the corresponding list of optimal values. Algorithm 4.9 details it. The criterion considered is the mean accuracy and a configuration is simply the key [algorithm, generator, seed] for `Group 1`, [algorithm, generator, concept drift type, seed] for `Group 2`, and [dataset, algorithm] for `Group 3`. *runExp* runs the experiment for a given configuration and *delta* with either `tinyMOA` or `tinyMOA-lite`. It returns the associated mean accuracy, which is the optimality criterion. Obviously, this part only concerns configurations that make use of adaptive models with `ADWIN`.

Algorithm 4.9 Pseudocode of the delta grid search

function DELTAGRIDSEARCH($configurations$)
  $deltas \leftarrow \{0.00001, 0.00005, ..., 1\}$
  $deltas^*_{tinyMOA} \leftarrow \emptyset$
  $deltas^*_{tinyMOA-lite} \leftarrow \emptyset$
  for each $conf \in configurations$ **do**
    $results_{conf,tinyMOA} \leftarrow \emptyset$
    $results_{conf,tinyMOA-lite} \leftarrow \emptyset$
    for each $delta \in deltas$ **do**
      $result_{tinyMOA,delta} \leftarrow runExp(conf, delta, tinyMOA)$
      $result_{tinyMOA-lite,delta} \leftarrow runExp(conf, delta, tinyMOA - lite)$
    end for
    $delta^*_{conf,tinyMOA} = argmax_{delta \in deltas}(results_{conf,tinyMOA})$
    $delta^*_{conf,tinyMOA-lite} = argmax_{delta \in deltas}(results_{conf,tinyMOA-lite})$
    $deltas^*_{tinyMOA} \leftarrow deltas^*_{tinyMOA} \cup delta^*_{conf,tinyMOA}$
    $deltas^*_{tinyMOA-lite} \leftarrow deltas^*_{tinyMOA-lite} \cup delta^*_{conf,tinyMOA-lite}$
  end for
  Return $deltas^*_{tinyMOA}$, $deltas^*_{tinyMOA-lite}$
end function

In a second part, a quasi-quantized version of `tinyMOA` was compiled. For this purpose, the `tinyMOA-lite` source code was modified thanks to static lists and counters which were used to backtrack the causes of the differences in the execution of the quantized and

non-quantized version. Some important code snippets are the ones that allow to extract these counters and lists from the text file results. They are presented in Listing 4.5 and Listing 4.6.

```
def get_counter(resultPath, counterName):
    counter = 0
    with open(resultPath, 'r') as file:
        for line in file:
            if line.startswith('counterName'):
                counter = int(line.split(' ')[-1])
    return counter
```

Listing 4.5: Pseudocode of the static counters extractor.

```
def get_list_from_string(list_string):
    res_list = []
    ind_start = [i for i,ind in enumerate(list_string) if ind == '['][0]+1
    ind_end = [i for i,ind in enumerate(list_string) if ind == ']'][0]
    res_list_string = list_string[ind_start:ind_end]
    res_list_string = res_list_string.split(' ')
    try:
        for elem in res_list_string:
            #delete all ',' from elem
            elem = elem.replace(',','')
            res_list.append(float(elem))
    except:
        print('Failed to find the list.')
return res_list
```

Listing 4.6: Pseudocode of the static lists extractor.

5 | Results

This chapter presents the results of all the different experiments presented in Chapter 3. They follow the same order and correspond to the described configurations. To keep the thesis readable, only the main results or some representative ones are presented here.

5.1. Raspberry Pi Simulation Results

5.1.1. Benchmarks Results

First will be presented the results of the benchmarks executed on the emulators, as detailed in Section 3.3.2. The benchmarks are divided between memory and CPU performance evaluation, and between single thread or multi threads execution. Each benchmark is composed of one or several compiled files (or operations). The emulators of the `Raspberry Pi 3b+` and the `Raspberry Pi 4b` were used, as the benchmarks results of the physical devices are presented in paper [38]. The objective is to compare the emulators' results with the corresponding physical devices results, to estimate and calibrate the emulators' performance. For each benchmark, two main results are presented:

- the first interesting one is the ratio between the emulator and its corresponding physical device performance, both for the `Raspberry Pi 3b +` and the `Raspberry Pi 4b`. The aggregation procedure across operations for a benchmark is as described in Eq. 5.1, where N_{op} is the number of operations of *benchmark*, $Emulator_{op,rpi}$ is the performance result of the *rpi* emulator for the operation *op* of *benchmark* and $Physical_{op,rpi}$ is the performance result of the physical device *rpi* for the operation *op* of *benchmark*.

$$\begin{cases} \forall benchmark \in \{Whetstone, Dhrystone, ...\} \\ \forall operation \in benchmark \\ \forall rpi \in \{Rpi3b+, Rpi4b\} \\ Ratio_{benchmark,rpi} = \frac{1}{N_{op}} \times \sum_{op \in benchmark} \frac{Emulator_{op,rpi}}{Physical_{op,rpi}} \end{cases} \tag{5.1}$$

- the second interesting one is the performance ratio of the `Raspberry Pi 3b +` with the `Raspberry Pi 4b`, to determine whether the `Raspberry Pi 4b` benefits are worth its extra cost on the one hand, and to make sure that the emulators' results are consistent with physical ones on the other hand. The aggregation procedure across operations for a benchmark is as described in Eq. 5.2, where N_{op} is the number of operations of *benchmark*, $RPi3b+_{op,env}$ is the performance result of the *env* for the operation *op* of *benchmark* and $RPi4b_{op,env}$ is the performance result of the *env* for the operation *op* of *benchmark*..

$$\begin{cases} \forall benchmark \in \{Whetstone, Dhrystone, ...\} \\ \forall operation \in benchmark \\ \forall env \in \{Emulator, Physical\} \\ Ratio_{benchmark,env} = \frac{1}{N_{op}} \times \sum_{op \in benchmark} \frac{RPi3b+_{op,env}}{RPi4b_{op,env}} \end{cases} \tag{5.2}$$

All the mentioned physical results on `Physical Raspberry Pi 3b+` and `Physical Raspberry Pi 4b` come from paper [38]. Detailed benchmarks results are included in Appendix A to keep this chapter compact.

Single Thread CPU Benchmarks

Single thread CPU benchmarks include sequentially executed operations. Absolute results, before ratio computations, are expressed in MFLOPS (Millions of Floating Point Operations Per Second) or MIPS (Millions of Instructions Per Second), see Appendix A.

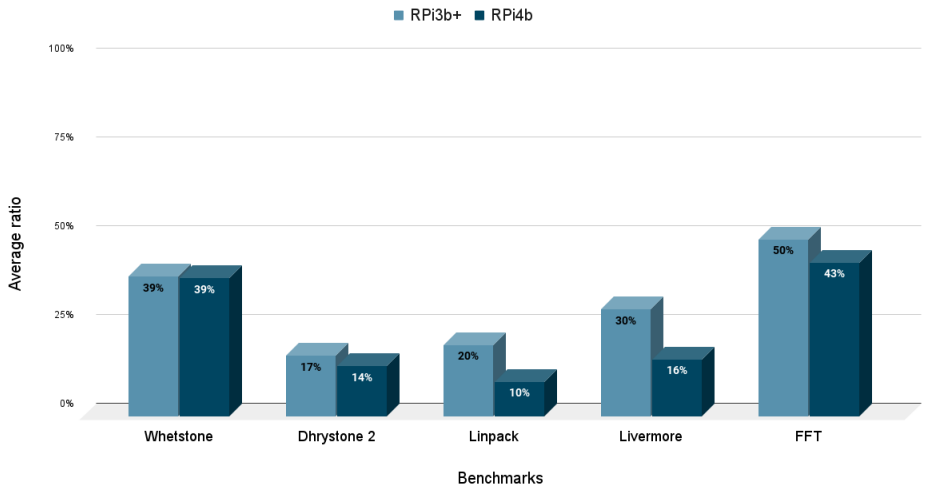

Figure 5.1: Performance ratio emulation/physical for single thread CPU benchmarks.

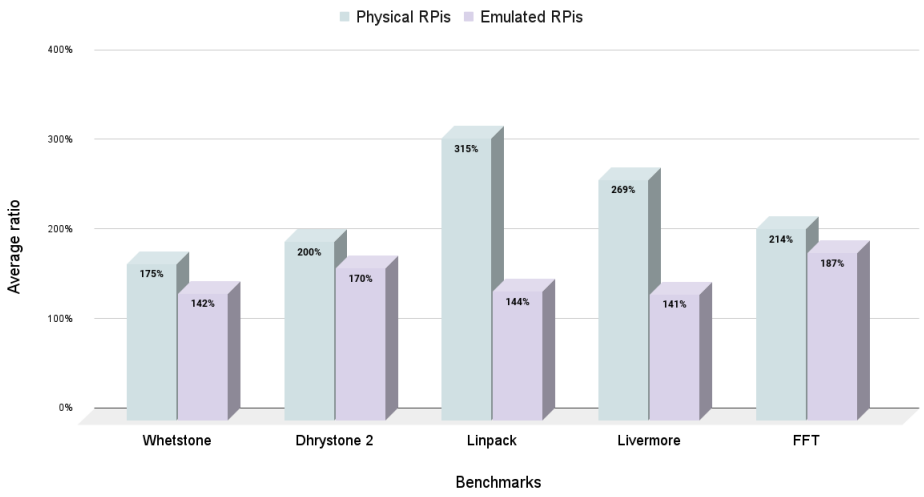

Figure 5.2: Performance ratio RPi3b+/RPi4b for single thread CPU benchmarks.

Figure 5.1 follows the principle of equation 5.1 and presents the ratio between physical and virtual environments both for the `Raspberry Pi 3b +` and the `Raspberry Pi 4b`. The `Raspberry Pi 3b+` emulator shows performance between 20% and 50% of those of the physical device, with an average of 33.2%. The `Raspberry Pi 4b` emulator shows performance between 10% and 45% of the physical device, with an average of 24.4%.

Figure 5.2 follows the principle of equation 5.2 and presents the performance ratio between the `Raspberry Pi 3b +` and the `Raspberry Pi 4b` both for the physical and virtual environments. Both environments show a ratio between 1.5 and 3 in favor of the `Raspberry Pi 4b`, with an average of 2.35 for the physical environment and 1.57 for the virtual one. These results demonstrate a certain consistency of the emulators' performance with respect to the physical boards' ones, which is a good sign that they preserve the differences and orders of magnitude between their target devices.

Multiple Threads CPU Benchmarks

Multiple threads CPU benchmarks operations are executed with a certain degree of parallelism, over multiple threads. Absolute results are still expressed in MFLOPS or MIPS, before being used for ratio, as it concerns CPU performance.

Figure 5.3 follows the principle of equation 5.1. The `Raspberry Pi 3b +` emulator shows results between 15% and 35% of the physical device, with an average of 20.8%. The `Raspberry Pi 4b` emulator shows results between 10% and 45% of the physical device, with an average of 21.8%, which is close to the single thread CPU benchmarks performance (with 24.4%).

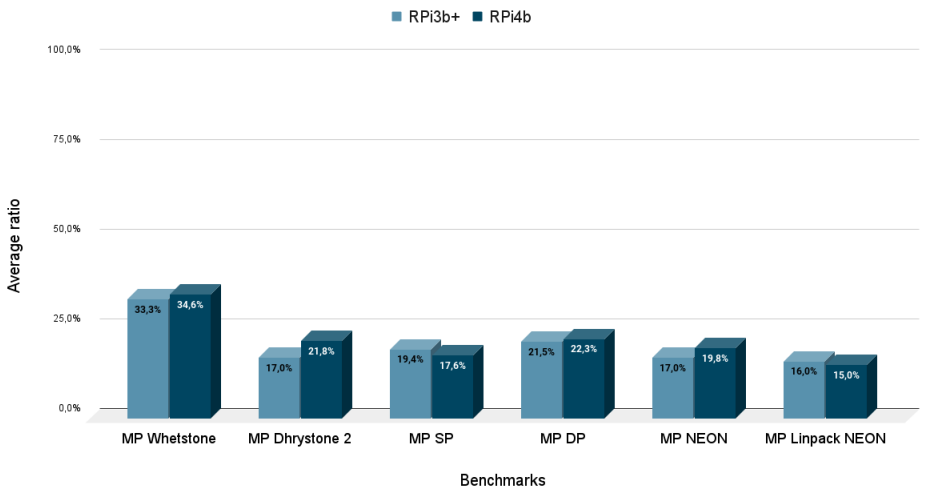

Figure 5.3: Performance ratio emulation/physical for single thread CPU benchmarks.

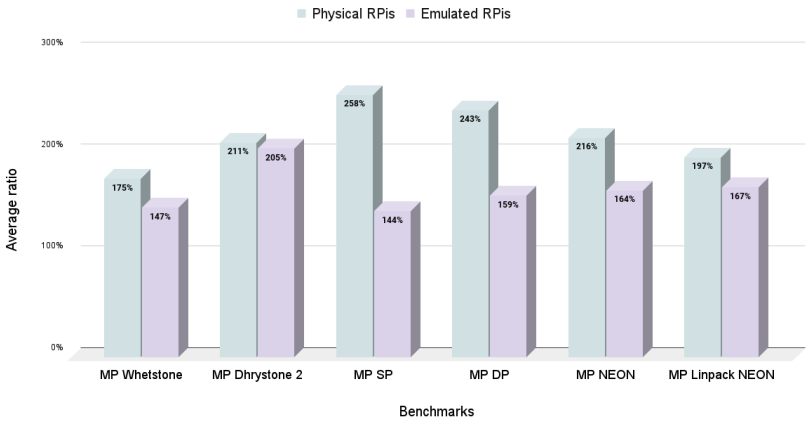

Figure 5.4: Performance ratio emulation/physical for single thread CPU benchmarks.

Figure 5.4 follows the principle of equation 5.2. The obtained ratios are between 1.4 and 2.5 in favor of the `Raspberry Pi 4b`, for both the physical and emulated environments, with 2.17 on average for the physical environment and 1.64 for the virtual one. Once again, the results show a great consistency between both environments.

Single Thread Memory Benchmarks

Memory benchmarks aim at measuring memory performance, in particular the capacity of the device to access and read data in cache and memory. The absolute results of the different operations, before being used to compute ratios, are expressed in MB/s. The single thread memory benchmarks include operations where the data must be read sequentially in cache and memory.

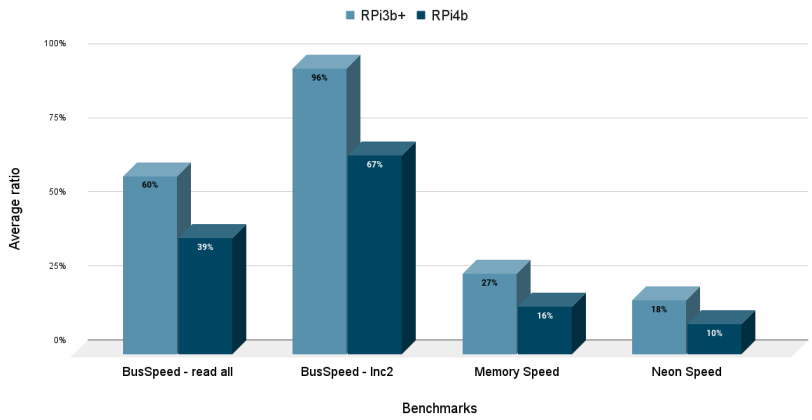

Figure 5.5: Performance ratio emulation/physical for single thread Memory benchmarks.

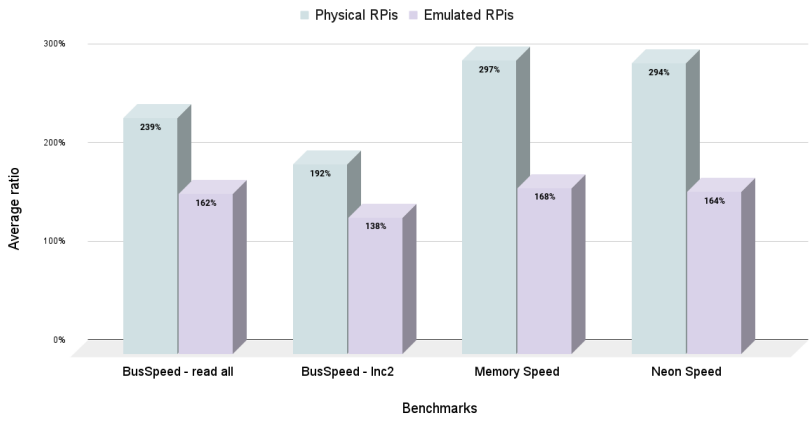

Figure 5.6: Performance ratio emulation/physical for single thread Memory benchmarks.

Figure 5.5 follows the principle of equation 5.1. The `Raspberry Pi 3b +` emulator shows results between 25% and 100% of the physical device, with an average of 50.3%. The `Raspberry Pi 4b` emulator shows results between 10% and 70% of the physical device, with an average of 33%.

Figure 5.6 follows the principle of equation 5.2. The obtained ratios are comprised between 1.5 and 3 in favor of the `Raspberry Pi 4b`, with an average of 2.56 for the physical environment and 1.58 for the virtual one.

Multiple Threads Memory Benchmarks

Multiple threads memory benchmarks consist of operations executed with a certain degree of parallelism, where the ability of the device to access and read data from different segments of the memory is evaluated.

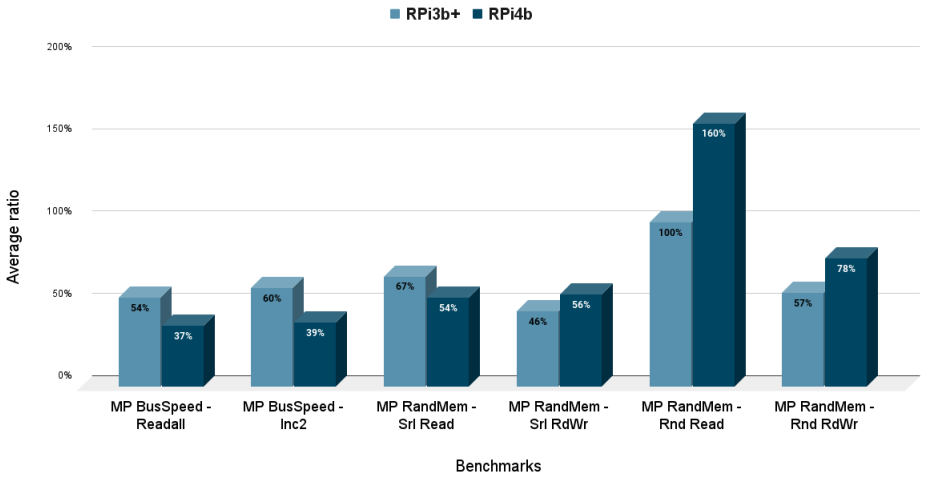

Figure 5.7: Performance ratio emulation/physical for multiple thread Memory benchmarks.

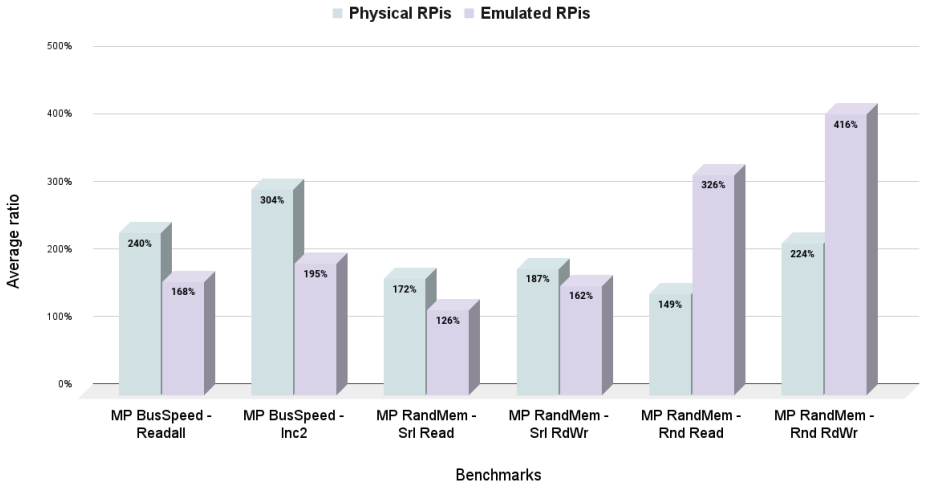

Figure 5.8: Performance ratio emulation/physical for multiple thread Memory benchmarks.

Figure 5.7 follows the principle of equation 5.1. The `Raspberry Pi 3b +` shows results between 45% and 100% of the physical device, with an average of 64%. The emulator of the `Raspberry Pi 4b` shows results between 35% and 160% of the physical device, with an average of 70.8%.

Figure 5.8 follows the principle of equation 5.2. The results show ratios between 1.2 and 4 in favor of the `Raspberry Pi 4b`, for both the physical and emulated environments, with an average of 2.13 for the physical environment and 2.32 for the virtual one.

Recap

The Raspberry Pi 3b+ emulator produces CPU performance around 27% of the physical device, and the emulator of the Raspberry Pi 4b shows CPU performance around 23% of the physical device. Consequently, we can roughly estimate, when it comes to executing MOA on the emulators, that the corresponding physical performance is approximately 3 to 5 times better. Concerning memory performance, although they are less interesting regarding the MOA experiments conducted in this thesis as data is mainly synthetically generated on the fly, the Raspberry Pi 3b+ emulator shows performance around 55% of the physical device, and the Raspberry Pi 4b emulator shows performance around 50% of the physical device.

Another important aspect that follows these results is the way the Raspberry Pi 4b outperforms the Raspberry Pi 3b+, around 2.2 faster on average for CPU performance and 1.6 times for memory performance. It gives a first strong insight that the benefits of choosing the Raspberry Pi 4b are worth its extra cost in comparison to the Raspberry Pi 3b+. This gain in performance is relatively consistent between the physical and virtual environments, which indicates that the emulators behave similarly to the physical devices to a certain extent, with a loss factor. This ends the first step of the suitability analysis as it gives an overview of the emulators' performance.

5.1.2. Emulators Results with MOA

After the emulators were evaluated and calibrated with the benchmarks, the time came to conduct the prepared bunch of experiments with MOA on these emulators. There are two groups of experiments, including the group implementing Tutorial 1 and Tutorial 5, and the group implementing the paper [4] experiments. While the results of the first group show the emulators' performance in absolute values, the results of the second group are particularly interesting as the paper [4] also details those on Raspberry Pi 4b, admittedly with the special configuration described in Section 2.4, but still providing a relatively precise overview of the physical board performance. Each group of experiments was run with MOA and tinyMOA to also evaluate the potential performance gain of the compact version.

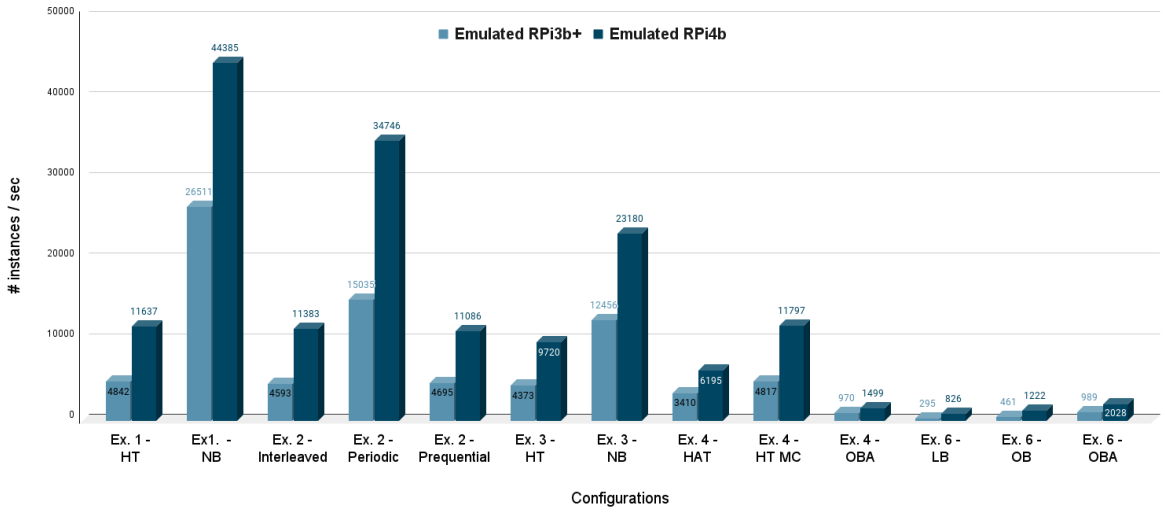

Figure 5.9: Emulators throughputs for Tutorial 1 experiments.

Tutorial 1

This section's results come from the experiments detailed in Table 3.3. Figure 5.9 includes the throughputs obtained with the `Raspberry Pi 3b+` and `Raspberry Pi 4b` emulators. These absolute values confirm that Raspberry Pi devices are able to run `MOA` with reasonable performance, i.e., that satisfy the requirements for certain applications. The `Raspberry Pi 3b+` emulator runs `MOA` with throughputs between 300 instances/sec and 25k instances/sec, depending on the model, the evaluation method or the generator. The `Raspberry Pi 4b` emulator runs `MOA` with throughputs between 800 instances/sec and 45k instances/sec. Ensemble methods perform worse than decision trees or naive Bayes, which is consistent as ensemble methods are more computationally intensive.

As expected, the `Raspberry Pi 4b` emulator performs better. Figure 5.10 demonstrates that the `Raspberry Pi 4b` emulator performs between 1.5 and 2.8 times faster, in terms of throughputs than the `Raspberry Pi 3b+` emulator. This confirms very well the results obtained with the CPU benchmarks.

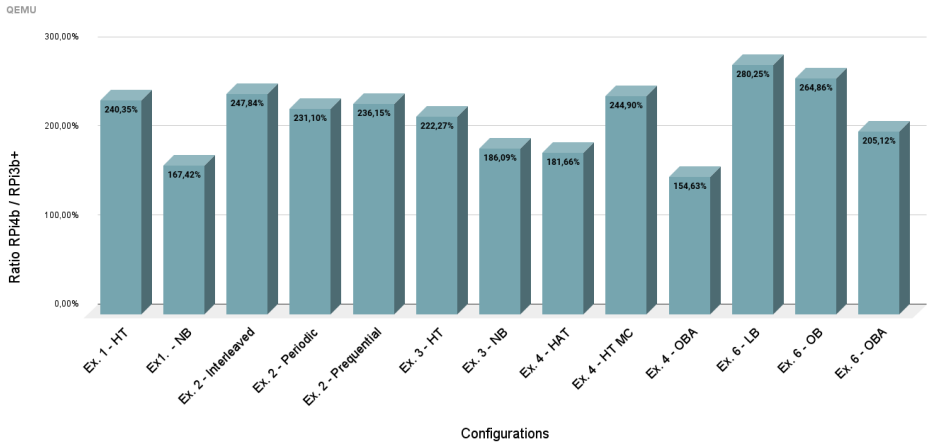

Figure 5.10: Comparison of the RPi3b+ and RPi4b emulators throughputs for the Tutorial 1 experiments.

Tutorial 5

This section's results come from the experiments detailed in Table 3.4. Figure 5.11 includes the throughputs obtained with the `Raspberry Pi 3b+` and `Raspberry Pi 4b` emulators. Once again, the `Raspberry Pi 4b` performs better than the `Raspberry Pi 3b+`. The `Raspberry Pi 3b+` emulator runs `MOA` with throughputs between 100 instances/sec and 10k instances/sec, depending on the configuration.

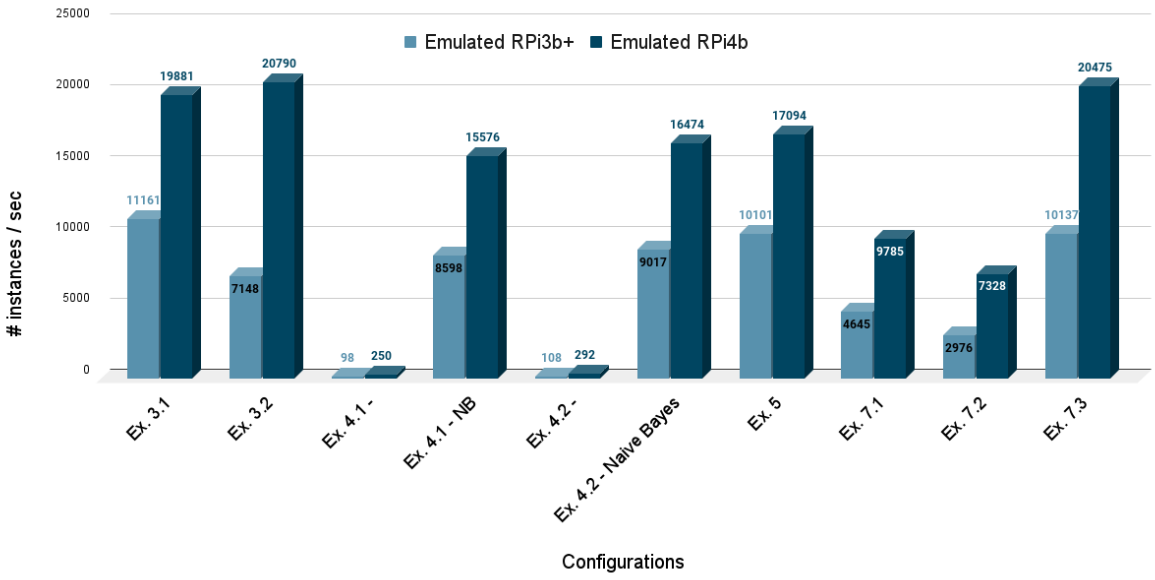

Figure 5.11: Emulators throughputs for the Tutorial 5 experiments.

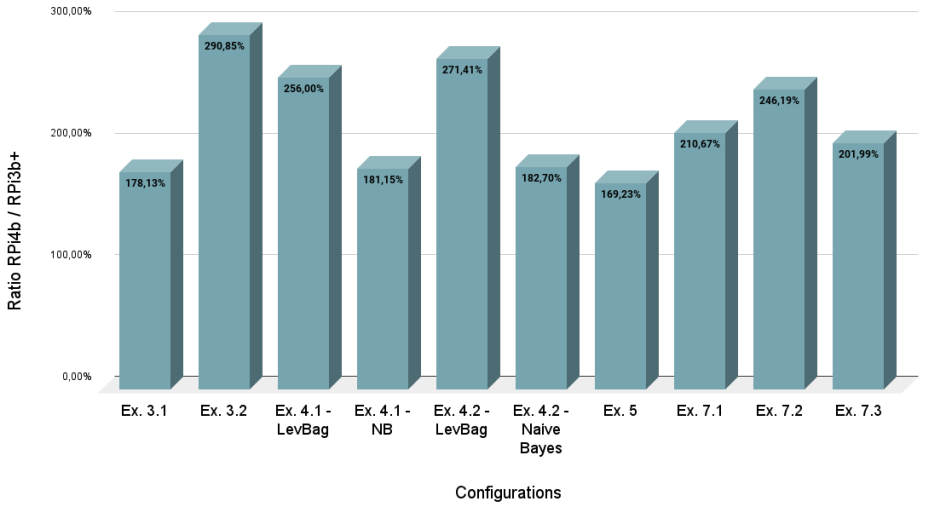

Figure 5.12: Comparison of the RPi3b+ and RPi4b emulators throughputs for the Tutorial 5 experiments.

The `Raspberry Pi 4b` emulator runs `MOA` with throughputs between 250 instances/sec and 20k instances/sec. Observation made on ensemble methods for Tutorial 1 also applies in this case.

Figure 5.12 shows that the `Raspberry Pi 4b` emulator performs between 1.8 and 2.9 times better, in terms of throughput than the `Raspberry Pi 3b+` emulator. This also confirms the results obtained with the CPU benchmarks.

TinyMOA

The idea was to try out the `tinyMOA` implementation with Tutorial 1 and 5 experiments and compare the results with the ones obtained with `MOA`. For embedded use, such a compact version is expected to provide gains in performance, as it is lighter than the original version, and the limited resources aspect of the Raspberry Pi should amplify this phenomenon.

Figure 5.13 shows the emulators' throughputs with both `tinyMOA` and `MOA`. The aggregation across subgroups, as defined in Table 3.10, is performed to keep the figure readable and the comparison easier. Significant improvements are observed by using `tinyMOA`, especially for the `Raspberry Pi 3b+` emulator, which is consistent with the fact that it embeds only 1GB of RAM, while the `Raspberry Pi 4b` emulation embeds 4GB of RAM.

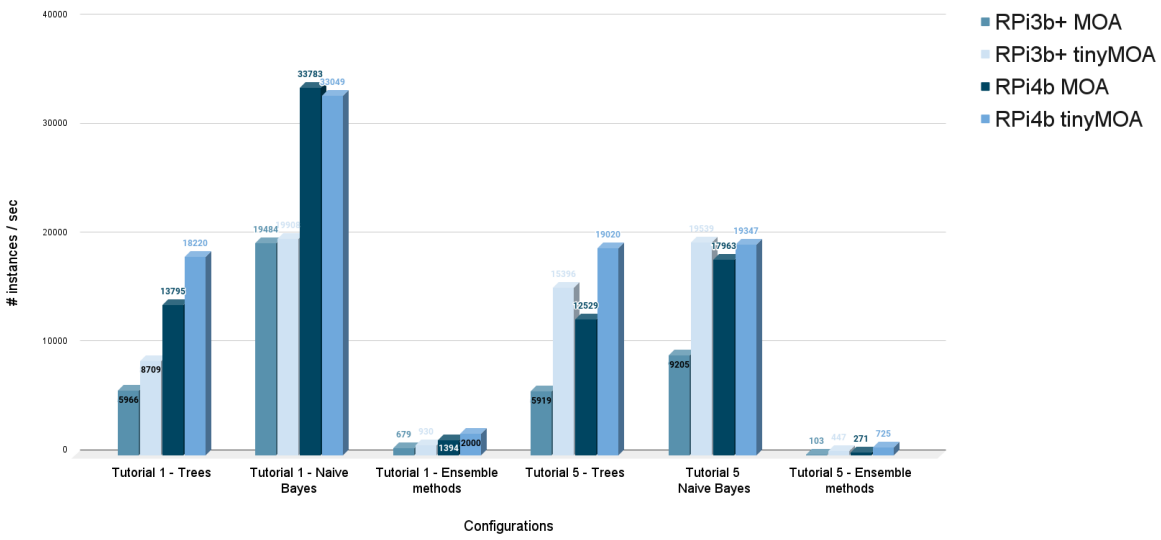

Figure 5.13: Emulators throughputs for experiments ran with MOA and tinyMOA.

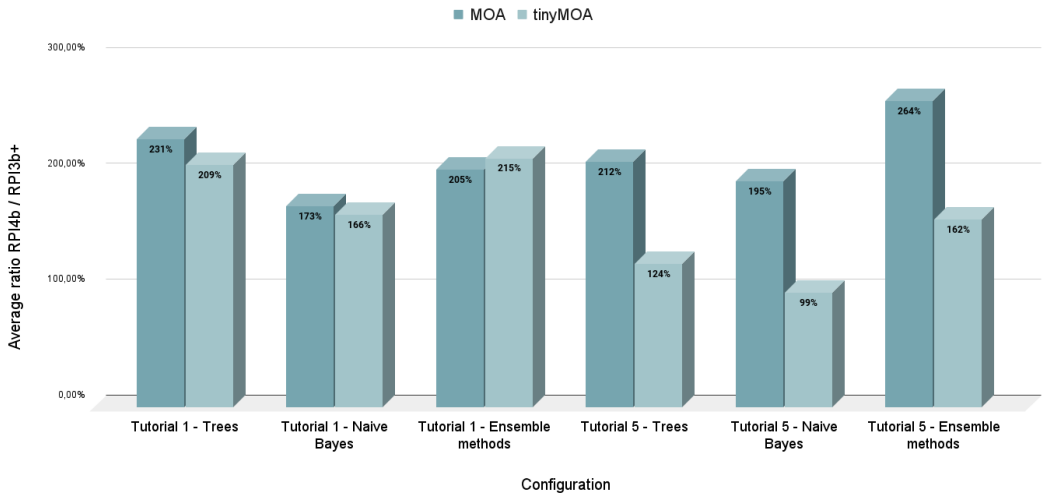

Figure 5.14: Comparison of the RPi3b+ and RPi4b emulators with tinyMOA.

And indeed, Figure 5.14 shows that the `Raspberry Pi 4b` emulator benefits less, around 1.6 times faster, from the use of `tinyMOA` than the `Raspberry Pi 3b+`, around 2.3 times faster on average. Consequently, `tinyMOA` reduced the gap between both emulators, although the `Raspberry Pi 4b` still provides significant gain with respect to the `Raspberry Pi 3b+`.

Paper Experiments

Paper [4] experiments results have the big advantage to be accompanied by those of the physical `Raspberry Pi 4b`. Even though `Kafka` and `Parsl` were used to manage the execution, it gives a good insight into the performance of the `Raspberry Pi 4b`. Only `Exp 3` results are presented here. The other experiments tend to produce the same conclusions and are included in Appendix B.

Exp 3 results, see Figure 5.16, 5.15 show that the `Raspberry Pi 4b` emulator reaches between 20% and 48%, with an average of 32% of the `Raspberry Pi 4b` physical board performance. By selecting the synthetically generated datasets, this interval is reduced to [20%,38%], which is very consistent with the results of the CPU benchmarks. By selecting the only real dataset, stored as an `arff` file, the same emulator reaches 48% of the physical performance, which is also consistent with both the CPU and memory benchmarks as data is read in memory in this case. Memory benchmarks gave a performance of 50% on average, with a maximum of 100%, of the physical board, while CPU benchmarks were sticking around 23%, within an interval of [10%, 45%].

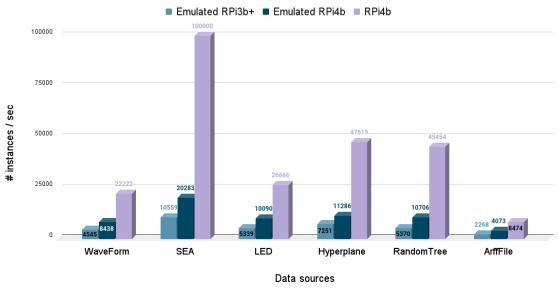

Figure 5.15: Emulators throughputs for Exp 3.

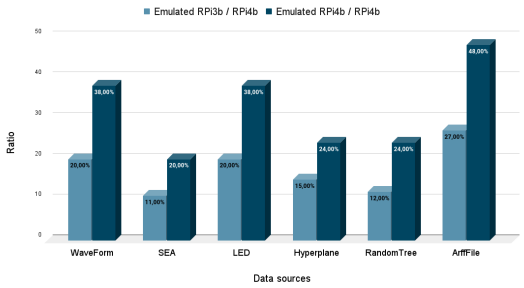

Figure 5.16: Comparison of emulators and RPi4b performance on Exp 3.

5.1.3. Simulation Conclusions

Tutorial 1 and 5 experiments results confirm that it is possible to run `MOA` on Raspberry Pi devices. The `Raspberry Pi 4b` is the best candidate, with a very significant gain, on average 2.2 times better than the `Raspberry Pi 3b+` for CPU performance and 1.6 times better for memory performance. The `tinyMOA` version also provides a significant gain in terms of performance, especially for the `Raspberry Pi 3b+`, which is consistent with the fact that it embeds less memory. The paper [4] experiments results tend to confirm the validity of the benchmarks results used for calibration with the benchmarks. All these observations finally confirm the feasibility of the project, i.e., the ability of the Raspberry Pi to run `MOA`, likely with reasonable performance. This ends the second step of the first axis of the project, the suitability analysis.

5.2. Raspberry Pi Results with MOA

Once the simulation part confirmed the ability of the Raspberry Pi to run `MOA`, a physical `Raspberry Pi 4b` was used to run Tutorial 1 and 5 experiments. This last step of the suitability analysis aims at validating the simulation results and providing absolute values to describe and give an overview of the physical board performance with the main streaming machine learning algorithms implemented in `MOA`. In addition, the same experiments were conducted on a Desktop computer, as described in Section 3.5, to evaluate the difference in performance between a high-performance computing resource and the `Raspberry Pi 4b`.

5.2.1. Tutorial 1

With Tutorial 1, the `Raspberry Pi 4b` delivers throughputs between 3500 and 250k instances/sec, while the Desktop delivers throughputs between 11000 and 770k instances/sec, see Figure 5.17. More in detail, ensemble methods perform the worst, with a throughput arithmetic mean of 5000 instances/sec for the Raspberry Pi and 21000 instances/sec for the Desktop. Naive Bayes performs the best with a throughput arithmetic mean of 194000 instances/sec for the Raspberry Pi and 576000 instances/sec for the Desktop. Finally, the `Hoeffding Trees` subgroup reaches 47000 instances/sec for the Raspberry Pi and 222000 instances/sec for the Desktop.

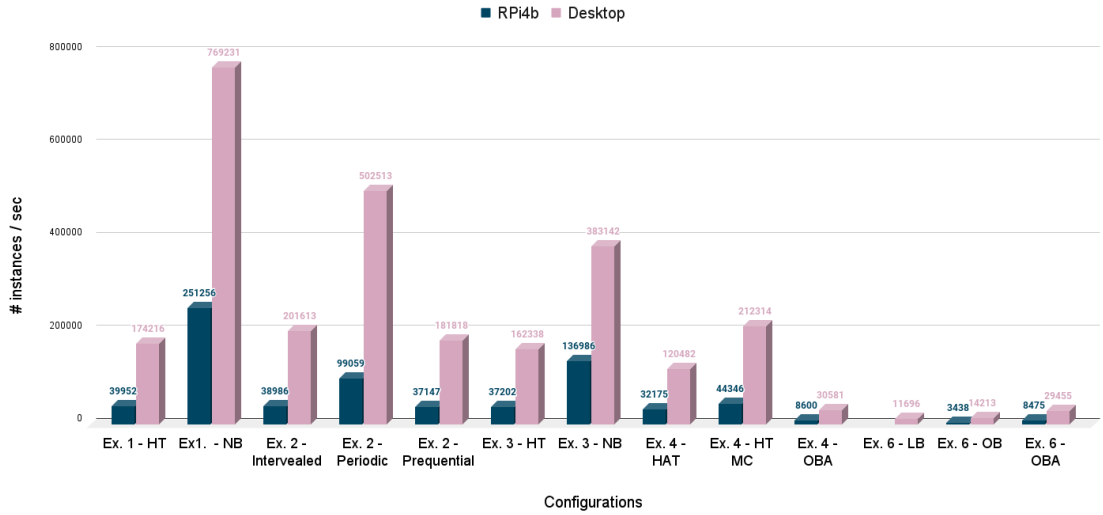

Figure 5.17: RPi4b and Desktop throughputs for Tutorial 1 experiments.

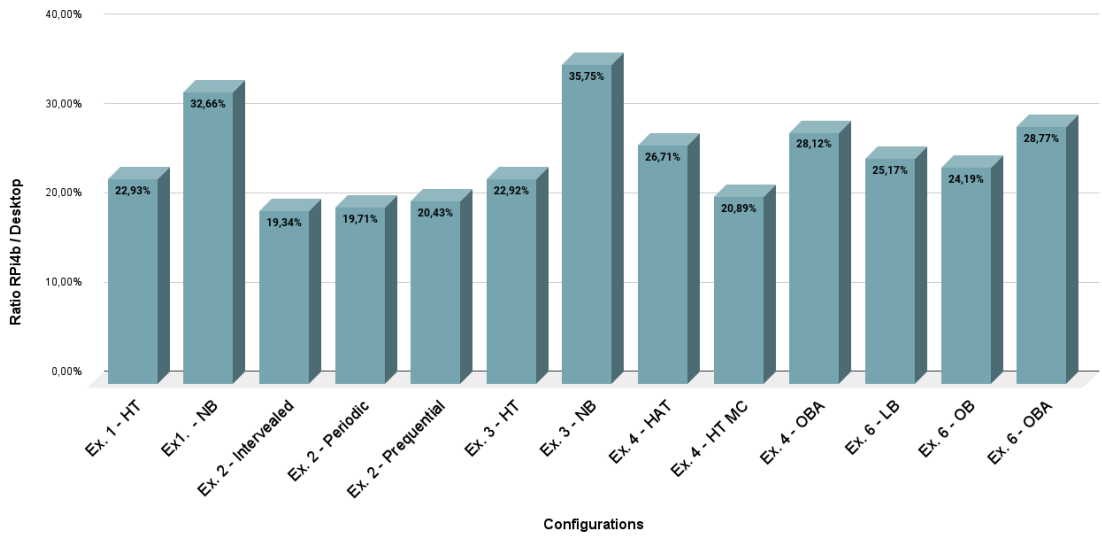

Figure 5.18: Comparison of the RPi4b and the Desktop for Tutorial 1 experiments.

The `Raspberry Pi 4b` is on average around 23.7% of the Desktop throughput, see Figure 5.18, which is a good result considering the difference in cost and size between the two platforms. With such results, a large panel of edge applications may be considered to be executed on the Raspberry Pi, within the throughputs limits presented before.

5.2.2. Tutorial 5

With Tutorial 5, the `Raspberry Pi 4b` delivers throughputs between 800 and 100k instances/sec, while the Desktop delivers throughputs between 5000 and 500k instances/sec, see Figure 5.19. More in detail, ensemble methods perform the worst, with a throughput arithmetic mean of 942 instances/sec for the Raspberry Pi and 5400 instances/sec for the Desktop. Naive Bayes performs the best with a throughput arithmetic mean of 86000 instances/sec for the Raspberry Pi and 413000 instances/sec for the Desktop. Finally, the `Hoeffding Trees` subgroup reaches 51000 instances/sec for the Raspberry Pi and 213000 instances/sec for the Desktop.

The `Raspberry Pi 4b` is on average around 21.1% of the Desktop throughput, see Figure 5.20, which is consistent with the observations made in Tutorial 1.

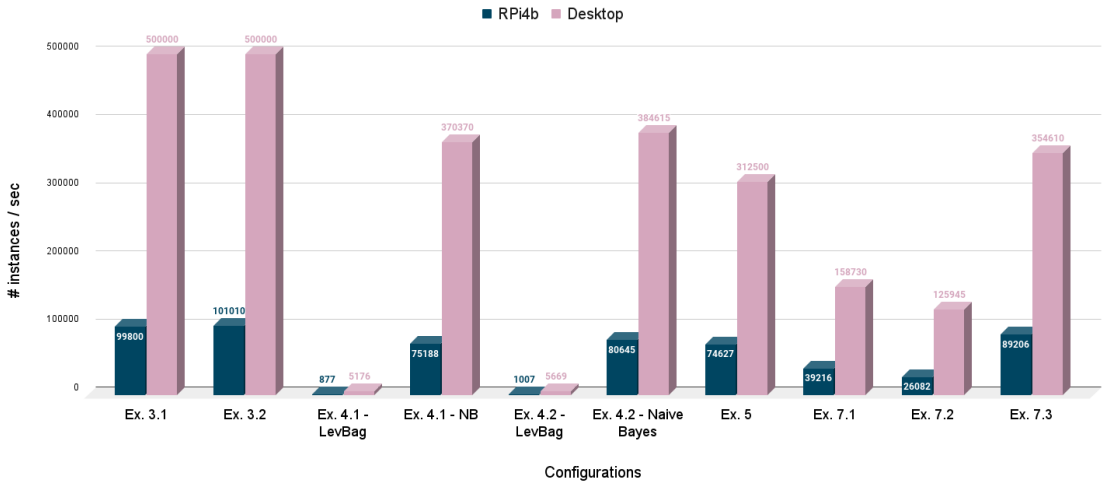

Figure 5.19: RPi4b and Desktop throughputs for Tutorial 5 experiments.

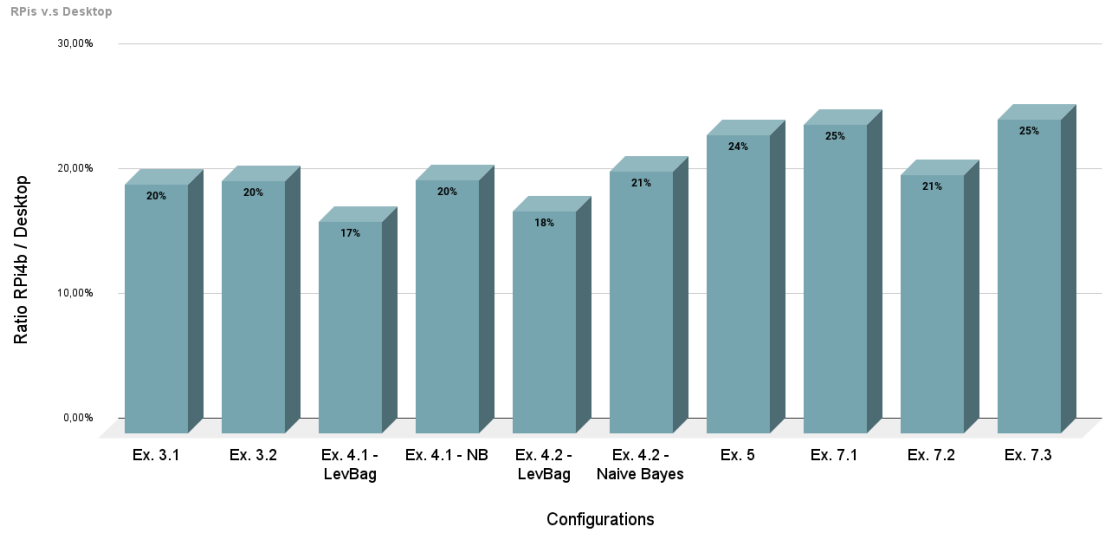

Figure 5.20: Comparison of the RPi4b and a Desktop for Tutorial 5 experiments.

5.2.3. Suitability Analysis Conclusions

This step ends the suitability analysis of the thesis, which was the first main axis. The simulation part's first step with benchmarks confirmed the feasibility of the project to run streaming machine learning algorithms on a Raspberry Pi device. The second step, with `MOA` on emulators, gave insights into how well it would perform on physical device, along with strong indications that the `Raspberry Pi 4b` is the best choice. Finally the experiments conducted on a physical `Raspberry Pi 4b` confirmed the suitability of the Raspberry Pi to run streaming machine learning algorithms, with throughputs around 22% of those of a high-performance computing resource, which is quite encouraging. The next axis focuses on a way to improve `MOA` performance on `Raspberry Pi 4b`.

5.3. Quantization Results

The quantization results follow the same order as the experiments presented in Section 3.6.

5.3.1. Overview

The first results are presented in Table 5.1. This overview is the highest level of aggregation. It shows the gain in terms of throughput, accuracy, and RAM.Hours for all the experiments of Group 1 and Group 2. Group 3 results are presented independently in Section 5.3.4, as they are used to ensure that the results obtained with synthetically generated data are consistent with a bunch of real datasets. Aggregations were performed

Accuracy	RAM.Hours	Throughput
-0.006%	-5.94%	+6.1%

Table 5.1: Quantization gains overview.

Overall	Ensemble Methods	Naive Bayes	Hoeffding Trees
12k inst/sec	6.8k inst/sec	175k inst/sec	38k inst/sec

Table 5.2: Quantization throughputs overview.

as described in Algorithm 4.7 and Algorithm 4.8, in particular harmonic mean were used for throughputs as they share the same numerator. In this way, averages correspond to the throughput required to process the sum of all instances within the overall time. Throughputs aggregations results are detailed in Table 5.2.

On average, the quantized version of `tinyMOA` is 6.1% faster in terms of throughput, which is a significant gain. The accuracy is slightly worse than the original version with a 0.0006% average loss, but it seems that the difference is bearable for most applications.

5.3.2. Results Group 1

For `Group 1`, as detailed in Tables 5.3 and 5.4, the quantized version of `tinyMOA` is 7.5% faster in terms of throughput, which is a significant gain. The accuracy is also slightly worse than the original version. More in detail, no subgroup show very significant differences in terms of accuracy. Even the `Ensemble Methods` subgroup, which is the more prone to differences as models are more complex, with two of the three models which are adaptive (`OzaBagAdwin` and `LeveragingBag`), shows a max difference among all experiments at any point of 1.37%. All experiment configurations [algorithm, generator], 60 in total, show a significant gain in throughput, according to the Welch's t-test over their seeds and repetitions with a 95% confidence level. The throughput gain for each subgroup is significant and no configuration has a negative throughput gain. This means that the

Accuracy	RAM.Hours	Throughput
-0.004%	-7.46%	+7.5%

Table 5.3: Quantization gains overview Group 1.

	Hoeffding Trees	Naive Bayes	Ensemble Methods
Accuracy Gain	+0.5%	-0.02%	-1.2%
Pearson	99.5%	100%	99.9%
MAE	0.04%	0.0007%	0.03%
Mean max diff	0.09%	0.003%	0.09%
Max max diff	2.67%	0.02%	1.37%
RAM.Hours Gain	-7.73%	-6.7%	-7.52%
Min gain	-0.8%	-2.4%	-3.3%
Max gain	-14.8%	-10.8%	-12.3%
Throughput Gain	+9.95%	+7.2%	+8.2%
Min gain	+1.1%	+2.5%	+3.37%
Max gain	+13.4%	+12.1%	+%14.1
Mean p-value	0.1%	9.8e−66	8.6e−20
Significant (95%)	20/20	10/10	30/30

Table 5.4: Quantization results Group 1.

quantized version always performs better than the original version on the experiments of `Group 1`.

5.3.3. Results Group 2

For `Group 2`, as detailed in Tables 5.5 and 5.6, the quantized version of `tinyMOA` is on average 5% faster in terms of throughput, which is also a significant gain. This time the loss in accuracy is higher than in `Group 1`, which is consistent with the fact that these experiments include added concept drifts, which make the differences in execution between the quantized and non-quantized less stable. Once again, the `Ensemble Methods` subgroup is the most prone to differences in terms of accuracy, with a max difference among all experiments at any point of 15.2% and an average loss of 2.7%. A max difference of 30.9% can be noted for the `Hoeffding Trees` subgroup, which is not representative of the global behavior of the subgroup. This value comes from a difference at a given point

Accuracy	RAM.Hours	Throughput
-0.8%	-4.7%	+5%

Table 5.5: Quantization gains overview Group 2.

	Hoeffding Trees	Naive Bayes	Ensemble Methods
Accuracy Gain	+2.7%	-0.01%	-2.7%
Pearson	97.8%	100%	97.9%
MAE	0.56%	0.016%	0.51%
Mean max diff	2.72%	0.12%	2.94%
Max max diff	30.9%	0.5%	15.2%
RAM.Hours Gain	-3.19%	-5.26%	-5.84%
Min gain	+0.8%	-3.9%	-2.8%
Max gain	-6.5%	-6.2%	-7%
Throughput Gain	+3.54%	+5.6%	+6.2%
Min gain	+1.5%	+4.1%	+2.9%
Max gain	+5.4%	+6.6%	+7.5%
Mean p-value	0.2%	1.4e−32	3.8e−13
Significant (95%)	24/24	12/12	36/36

Table 5.6: Quantization gains overview Group 2.

after an abrupt drift, where one of the two versions responds faster than the original version. This is typically the kind of difference stressed by the delta tuning part of the quasi-quantization process 5.4.1. All experiment configurations [algorithm, generator, concept drift type], 72 in total, show a significant gain in terms of throughput, according to the Welch's t-test over their seeds and repetitions with a 95% confidence level.

5.3.4. Results Group 3

Finally, the results for the experiments conducted on real datasets (`Group 3`) (see Table 5.7) confirm the results obtained on synthetic data, with significant gains for the quantized version. Depending on the data structure, with more or fewer features and different structure complexity, the quantization process leads to throughput gains between 2.1% and 13.9%. The accuracy loss is between 0% and 9.4%, which is also consistent with the previous results, especially if the `spam_corpus` dataset is not considered, in which case the accuracy loss is comprised between 0% and 0.3%. The `spam_corpus` is a very special dataset, with a very high number of features (39916) which makes the `Hoeffding Trees`, as well as `Naive Bayes`, very inefficient and makes a difference in the split choice between the quantized and original versions very likely to happen, even with a small difference in precision in the information gain.

	Accuracy Gain	RAM.Hours Gain	Throughput Gain
airlines.arff	-0.04%	-2.7%	+3.9%
covtypeNorm.arff	+0%	-1.6%	+4.2%
elecNormNew.arff	-0.1%	-4.5%	+2.1%
GMSC.arff	-0.1%	-4.5%	+2.1%
kdd99.arff	-0.3%	-8.8%	+12.1%
spam-corpus.arff	-9.4%	-18.8%	+13.9%

Table 5.7: Quantization gains overview Group 3.

5.4. Quasi-quantization Results

5.4.1. Delta Tuning

For the delta tuning task, the objective was to observe the effect of tuning the *delta* hyperparameter of `ADWIN`, used by adaptive models such as `HoeffdingAdaptiveTree`, `OzaBagAdwin`, or `LeveragingBag`. The quantization has for side effect of possibly creating differences between the original model and the quantized model during the training phase. Adaptive models are particularly affected as concept drift detection may cause major changes in the model structure, such as a full branch replacement or a full base learner replacement in the case of ensemble methods. An example of an interesting configuration ([LeveragingBag, SEAGenerator,1AD].) is shown in Figure 5.21, where the two versions of `tinyMOA` react differently to an abrupt drift.

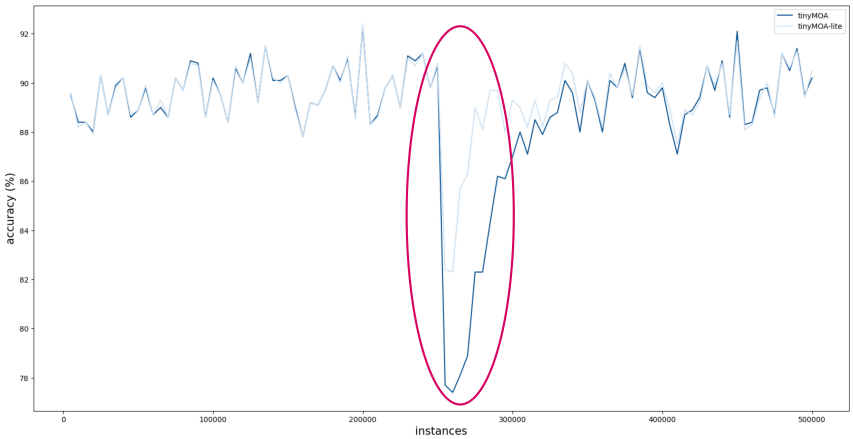

Figure 5.21: Example of an interesting configuration.

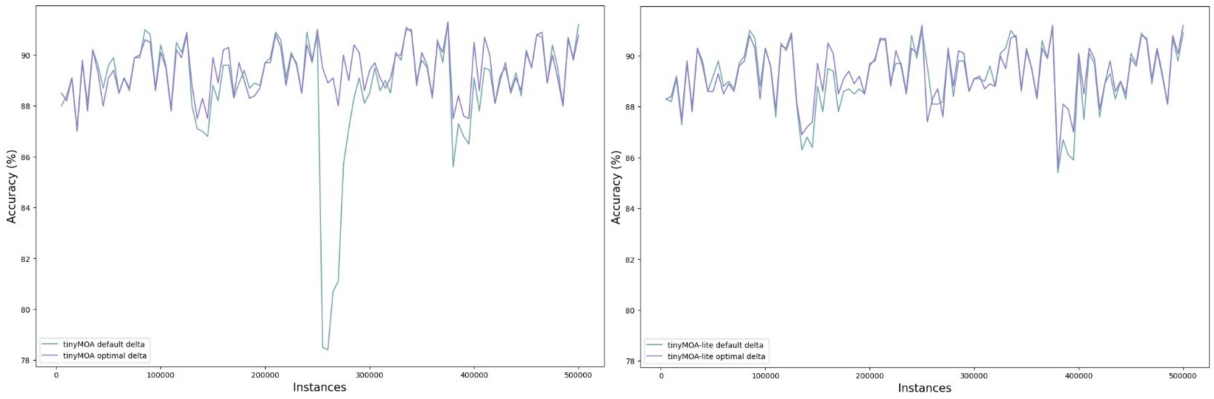

Figure 5.22: Comparison of the accuracy with default and optimal delta for tinyMOA and tinyMOA-lite.

Figure 5.22 shows that the optimal delta for both `tinyMOA` and `tinyMOA-lite` generates different results than the default delta. The criterion used to select the best delta is the mean accuracy.

By selecting the optimal delta, the two versions generate more similar results than with the default delta. In the example shown in Figure 5.23, the mean absolute error changes from 0.65% to 0.27%. However, this does not make the two executions perfectly similar, which was the initial objective, as the differences may come from different causes than the concept drift detector. This is the topic of the next and last part, where differences are treated directly by reintegrating 64-bit variables in the source code.

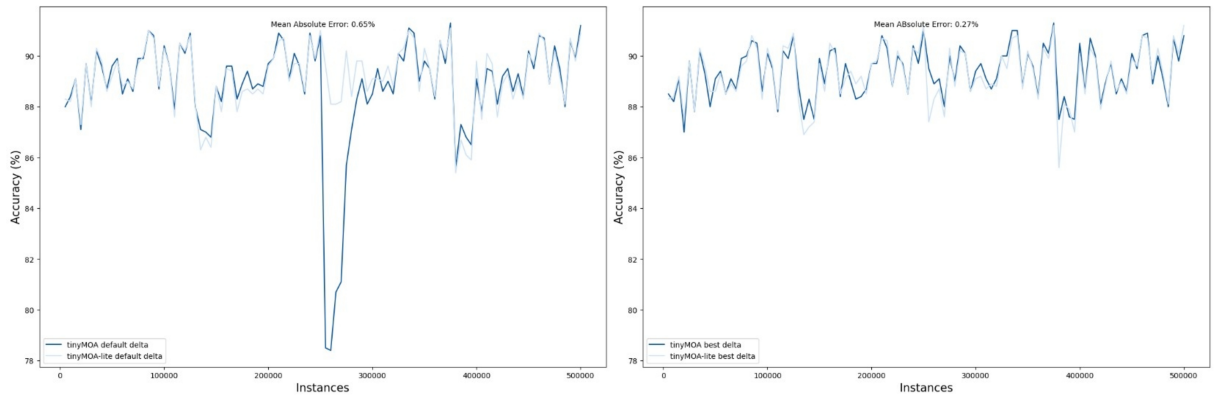

Figure 5.23: Comparison of the accuracy of tinyMOA and tinyMOA-lite with default and optimal delta.

	Accuracy Gain	RAM.Hours Gain	Throughput Gain
Overall	-0.07%	+2.1%	-2.1%
No concept drift	-0.2%	+2.5%	-2.4%
Concept drift	+0%	+1.8%	-1.8%

Table 5.8: Quasi-quantization gains overview.

5.4.2. Quasi-quantization Results

As explained in Section 3.7.2, the quasi-quantization was applied at the cost of many 64-bit variables. In addition to these variables, additional mathematical operations were required to keep some results in ranges that can be represented by 32-bit variables, for instance with the power function. The consequence of these modifications is the loss of all the quantization benefits, and more generally a loss of performance even in comparison with the non-quantized version, as shown in Table 5.8. The presented results are the gain of the quasi-quantized version against the non-quantized one.

It may be noticed that accuracy differences seem worst than the ones obtained with basic quantization. The reason is that across quantization results, accuracy means can compensate as in certain cases the quantized version execution leads to better mean accuracy, while the quasi-quantized version proposes the same concept drift detections and node splits as the original version and accuracy only suffers the lack of precision of the remaining 32-bit variables.

These results lead to the conclusion that quasi-quantization is not a valid option as it requires too many modifications to the quantized version, and even degrades the results of the original version.

6 | Conclusions and Future Developments

The ambition of this thesis was to explore the possibilities offered by the combination of two domains, streaming machine learning, and embedded systems, which are bound, if they are not already, to take a major place in the development of new intelligent systems. The first axis of work, the suitability analysis of the Raspberry Pi to run streaming machine learning algorithms, was carried out in three steps. First, Raspberry Pi emulators were implemented to determine the feasibility of the project, and a set of benchmarks was executed to calibrate them against the corresponding physical boards. The series of experiments with `MOA` carried out confirmed the ability of the Raspberry Pi to run these algorithms, to a reasonable extent that could be suitable for a number of applications. In a second step, the same series of experiments were conducted on a physical Raspberry Pi 4b 4GB board, in order to obtain exploitable results of the board's performance and compare them to the performance of a desktop computer. The second axis of work consisted in studying the possibility of improving the execution performance previously obtained, by using the quantization process on `tinyMOA`. The results showed that quantization allows to improve execution performance with a relatively small loss of precision. The results of this part contribute at their scale to the state of the art, by proposing this new solution particularly advantageous for embedded systems. This second axis also included a study of a compromise between the quantized and original versions, with quasi-quantization, where the objective is to obtain a perfectly similar execution as the original version, with as less 64-bit variables as possible. It appeared that this solution loses all the benefits of quantization, and even more, gives worse performance than the original version. Consequently, a choice has to be made between the original version with its 64-bit precision, or more throughput with a quantized version with a bit less precision.

To go further, it would be interesting to quantize `tinyMOA` even more, using 32-bit or 16-bit integer variables for example. Experiments with streaming machine learning algorithms for unsupervised or regression tasks could also be implemented instead of the classification ones. Finally, a new and more general development axis would be to reproduce these

experiments with another embedded system, perhaps even more compact and less powerful than the Raspberry Pi, or within another streaming machine learning framework such as the `River` python library.

Bibliography

[1] Heitor Murilo Gomes, Jesse Read, Albert Bifet, Jean Paul Barddal, and Joao Gama. Machine learning for streaming data: state of the art, challenges, and opportunities. *ACM SIGKDD Explorations Newsletter*, 21:6–22, 11 2019. doi: 10.1145/3373464.3373470.

[2] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[3] Massive online analysis, 2022. URL `https://moa.cms.waikato.ac.nz/details/`. Accessed: 2022-10-27.

[4] I. Petri, I. Chirila, H. Gomes, A. Bifet et O. Rana. Resource-aware edge-based stream analytics. *IEEE Internet Computing*, pages 79 – 88, 2022.

[5] Raspberry pi, 2022. URL `https://en.wikipedia.org/wiki/Raspberry_Pi`. Accessed: 2022-10-27.

[6] Arm, 2022. URL `https://en.wikipedia.org/wiki/ARM_architecture_family`. Accessed: 2022-10-27.

[7] Raspberry pi 4b, 2022. URL `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`. Accessed: 2022-10-27.

[8] Leo Breiman. Bagging predictors. *Machine Learning*, 01 1996. doi: 10.1007/BF00058655.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[10] Alessio Bernardo and Emanuele Della Valle. An extensive study of c-smote, a continuous synthetic minority oversampling technique for evolving data streams. *Expert Systems with Applications*, 196:116630, 2022. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2022.116630.

[11] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. *SDM*, 2007.

[12] Emanuele Della Valle & Alessio Bernardo. Streaming machine learning, pack of slides 15b, 2021. URL `https://github.com/quantiaconsulting/streaming-machine-learning/blob/main/slides/2_CD.pdf`. DEIB - Politecnico di Milano, Accessed: 2022-10-27.

[13] João Gama, Raquel Sebastião, and Pedro Rodrigues. Issues in evaluation of stream learning algorithms. *Issues in evaluation of stream learnings algorithms*, pages 329–338, 01 2009. doi: 10.1145/1557019.1557060.

[14] João Vinagre, Alípio Mário Jorge, and Joao Gama. Online gradient boosting for incremental recommender systems. In *International Conference on Discovery Science*, pages 209–223. Springer, 2018.

[15] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, pages 1–15, 01 1986. doi: 10.1007/BF00116251.

[16] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2000.

[17] Albert Bifet and Ricard Gavalda. Adaptive learning from evolving data streams. In *International Symposium on Intelligent Data Analysis*, pages 249–260. Springer, 2009.

[18] David J Hand and Keming Yu. Idiot's bayes—not so stupid after all? *International statistical review*, 69(3):385–398, 2001.

[19] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. *CoRR*, abs/1302.4964, 2013.

[20] Emanuele Della Valle & Alessio Bernardo. Streaming machine learning, pack of slides 16a, 2021. URL `https://github.com/quantiaconsulting/streaming-machine-learning/blob/main/slides/3_Classification.pdf`. DEIB - Politecnico di Milano, Accessed: 2022-10-27.

[21] D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, aug 1999. doi: 10.1613/jair.614.

[22] Heitor Murilo Gomes, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet. A survey on ensemble learning for data stream classification. *ACM Comput. Surv.*, 50 (2), mar 2017. ISSN 0360-0300. doi: 10.1145/3054925.

[23] Emanuele Della Valle & Alessio Bernardo. Streaming machine learning, pack of slides 16b, 2021. URL `https://github.com/quantiaconsulting/streaming-machine-learning/blob/main/slides/4_Ensemble_Classification.pdf`. DEIB - Politecnico di Milano, Accessed: 2022-10-27.

[24] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168:022022, 02 2019. doi: 10.1088/1742-6596/1168/2/022022.

[25] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[26] Weikato MOA. Streaming random patches, 2020. URL `https://moa.cms.waikato.ac.nz/streaming-random-patches/`. Accessed: 2020-10-27.

[27] Heitor Murilo Gomes, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet. A survey on ensemble learning for data stream classification. *ACM Computing Surveys (CSUR)*, 50:1 – 36, 2017.

[28] Nikunj Oza and Stuart Russell. Online bagging and boosting. *Proceedings of Artificial Intelligence and Statistics*, 01 2001.

[29] Wikipedia. Poisson distribution. `https://en.wikipedia.org/wiki/Poisson_distribution`, 2022. Skbkekas, CC BY 3.0 https://creativecommons.org/licenses/by/3.0, via Wikimedia Commons, Accessed: 2022-10-27.

[30] Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. Leveraging bagging for evolving data streams. In José Luis Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 135–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15880-3.

[31] Wikipedia. Edge computing. `https://en.wikipedia.org/wiki/Edge_computing`, 2022. Accessed: 2022-10-27.

[32] Arthur Asuncion and David Newman. Uci machine learning repository, 2007.

[33] A. Kochar, B. S. Pattnaik et A. Kumar Panda. Machine learning applied in the edge-devices. *IEEE Internet Computing*, 2021.

[34] Soe, Yan Naung and Feng, Yaokai and Santosa, Paulus Insap and Hartanto, Rudy and Sakurai, Kouichi. Towards a lightweight detection system for cyber attacks in

the iot environment using corresponding features. *Electronics*, 9(1), 2020. ISSN 2079-9292. doi: 10.3390/electronics9010144. URL https://www.mdpi.com/2079-9292/9/1/144.

[35] M. H. Gauswami and K. R. Trivedi. Implementation of machine learning for gender detection using cnn on raspberry pi platform. *IEEE Xplore*, 2018.

[36] Alex Marchioni, Luciano Prono, Mauro Mangia, Fabio Pareschi, Riccardo Rovatti, and Gianluca Setti. Streaming Algorithms for Subspace Analysis: an Edge- and Hardware-oriented review. *TechRxiv*, 5 2021. doi: 10.36227/techrxiv.14694420.v1.

[37] Qemu, 2022. URL https://www.qemu.org/.
Accessed: 2022-10-27.

[38] Roy Longbottom. Raspberry pi 4b 64 bit benchmarks and stress tests. *ResearchGate*, 11 2019. doi: 10.13140/RG.2.2.33099.54562.

[39] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–382, 2001.

[40] R. Agrawal, T. Imielinski, and A. Swami. Database mining: a performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, 1993. doi: 10.1109/69.250074.

[41] Wei-Yin Loh. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23, 2011.

[42] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106, 2001.

[43] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. *Proceedings of the 7th SIAM International Conference on Data Mining*, 7, 04 2007. doi: 10.1137/1.9781611972771.42.

[44] Heitor M Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9):1469–1495, 2017.

A | Benchmarks

All benchmarks are detailed in paper [4], where they come from.

A.1. Single Thread CPU Benchmarks

Whetstone

	MWIPS	FP1	FP2	FP3	COS	EXP	FIXPT	EQUAL
Emulated RPi3b+	393	139	109	151.6	6.13	6	705	660
Physical RPi3b+	1071	383	403	328	20.9	12.4	1704	1357
Emulated RPi4b	547.9	256.7	167.5	208	8.52	8.51	721.8	892
Physical RPi4b	2269	522	534	398	54.8	39.8	2487	997

Table A.1: Whetstone benchmarks results in MFLOPS.

Dhrystone

	VAX
Emulated RPi3b+	675.6
Physical RPi3b+	4028
Emulated RPi4b	1150.57
Physical RPi4b	8176

Table A.2: Dhrystone benchmarks results in MIPS.

Linpack

	DP	SP	NEON SP
Emulated RPi3b+	83.57	109.62	116.19
Physical RPi3b+	396.6	562.1	604.2
Emulated RPi4b	112.7	161.8	174.67
Physical RPi4b	1059.9	1977.8	1968.6

Table A.3: Linpack benchmarks results in MIPS.

Livermore

	Max DP	Avg DP	Geomean DP	Harmean DP	Min DP
Emulated RPi3b+	171.2	88.2	81.5	74.8	29.5
Physical RPi3b+	737.7	319.4	284.7	250.6	91.6
Emulated RPi4b	264.6	126	115.3	104.9	42.1
Physical RPi4b	2490.5	892	730.3	603.3	212.4

Table A.4: Livermore benchmarks results in MFLOPS.

FFT

	FFT1 SP	FFT1 DP	FFT3 SP	FFT3 DP
Emulated RPi3b+	158.75	478.15	106.27	141.87
Physical RPi3b+	329.6	445.64	360.34	957.85
Emulated RPi4b	195.51	292.8	33.53	48.46
Physical RPi4b	253.71	462.44	233.73	261.28

Table A.5: FFT benchmarks results in MFLOPS.

A.2. Multiple Threads CPU Benchmarks

MP Whetstone

	MWIPS	FP1	FP2	FP3	COS	EXP	FIXPT	EQUAL
Emulated RPi3b+	389.4	172.5	174.7	122.9	6.4	5.43	649.3	521.7
Physical RPi3b+	1152	383	383	328	23.2	13	2721	1365
Emulated RPi4b	542.9	229.1	223.7	179.1	8.7	7.8	1142.7	819.9
Physical RPi4b	2395	536	538	397	60.8	39	4483	997

Table A.6: MP Whetstone benchmarks results for 1 thread in MFLOPS.

	MWIPS	FP1	FP2	FP3	COS	EXP	FIXPT	EQUAL
Emulated RPi3b+	787.8	325.2	315.1	266.8	12.3	11.86	1665.8	1055.3
Physical RPi3b+	2312	767	767	657	46.5	26	5461	2738
Emulated RPi4b	1035.1	496.8	468.5	397	15.6	15.15	2174.1	1596.9
Physical RPi4b	4784	1062	1079	794	121.2	77.9	8932	1990

Table A.7: MP Whetstone benchmarks results for 2 threads in MFLOPS.

	MWIPS	FP1	FP2	FP3	COS	EXP	FIXPT	EQUAL
Emulated RPi3b+	1290.3	553.9	443	489.4	20.4	18	2348.3	1837.6
Physical RPi3b+	4580	1506	1526	1304	92	51.6	10777	5449
Emulated RPi4b	1844.5	845.2	936.3	657.4	28.3	26.92	3469.9	2503.3
Physical RPi4b	9476	2125	2080	1568	240.8	155.3	17718	3962

Table A.8: MP Whetstone benchmarks results for 4 threads in MFLOPS.

	MWIPS	FP1	FP2	FP3	COS	EXP	FIXPT	EQUAL
Emulated RPi3b+	1230.2	592.9	609.1	435.6	19	17.8	3632.3	1982.7
Physical RPi3b+	4788	1815	1961	1382	95	53.3	13827	5811
Emulated RPi4b	1898.9	944.1	1028.3	679.2	28.6	28.52	4362.2	2907
Physical RPi4b	9834	2631	2744	1630	243.6	160.1	22265	4053

Table A.9: MP Whetstone benchmarks results for 8 threads in MFLOPS.

MP Dhrystone

	VAX 1 thread	VAX 2 threads	VAX 4 threads	VAX 8 threads
Emulated RPi3b+	704	1065	1281	1250
Physical RPi3b+	4207	6804	7401	7415
Emulated RPi4b	1445	1532	2089	2127
Physical RPi4b	8880	7828	8303	8314

Table A.10: MP Dhrystone benchmarks results in MIPS.

MP MFLOPS

MP MFLOPS Single Precision

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	124	120	116	138	143	150
Physical RPi3b+	792	806	373	1780	1783	1724
Emulated RPi4b	211	197	185	222	219	212
Physical RPi4b	2908	2854	459	5778	5734	5405

Table A.11: MP MFLOPS Single Precision benchmarks results for 1 thread in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	302	311	210	316	334	303
Physical RPi3b+	1482	1596	382	3542	3509	3380
Emulated RPi4b	368	353	363	414	391	377
Physical RPi4b	5700	5311	457	10935	11212	7968

Table A.12: MP MFLOPS Single Precision benchmarks results for 2 threads in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	398	364	347	476	465	432
Physical RPi3b+	2861	2742	429	5849	7013	5465
Emulated RPi4b	418	488	441	663	676	666
Physical RPi4b	10375	5588	490	10181	21842	7637

Table A.13: MP MFLOPS Single Precision benchmarks results for 4 threads in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	321	326	342	463	463	461
Physical RPi3b+	2770	2877	429	6434	6700	6101
Emulated RPi4b	427	615	554	628	705	692
Physical RPi4b	9675	8460	511	20128	20567	8568

Table A.14: MP MFLOPS Single Precision benchmarks results for 8 threads in MFLOPS.

MP MFLOPS Double Precision

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	91	77	65	136	132	120
Physical RPi3b+	415	386	209	1400	1403	1333
Emulated RPi4b	134	133	124	191	199	190
Physical RPi4b	1464	1386	225	3398	3386	3182

Table A.15: MP MFLOPS Double Precision benchmarks results for 1 thread in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	167	191	135	249	269	240
Physical RPi3b+	820	813	209	2804	2767	2597
Emulated RPi4b	229	231	212	367	352	339
Physical RPi4b	2837	2792	228	6720	6741	4547

Table A.16: MP MFLOPS Double Precision benchmarks results for 2 threads in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	191	201	134	358	377	354
Physical RPi3b+	1328	1323	212	5433	5340	2465
Emulated RPi4b	320	349	348	558	564	552
Physical RPi4b	5172	3414	251	10405	12762	4763

Table A.17: MP MFLOPS Double Precision benchmarks results for 4 threads in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	158	307	196	397	406	375
Physical RPi3b+	1343	1308	214	5090	5006	3280
Emulated RPi4b	389	386	300	651	529	524
Physical RPi4b	4774	4353	275	11506	12118	4865

Table A.18: MP MFLOPS Double Precision benchmarks results for 8 threads in MFLOPS.

MP MFLOPS NEON

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	139	162	114	152	141	139
Physical RPi3b+	830	823	406	2989	2986	2792
Emulated RPi4b	221	195	198	245	241	236
Physical RPi4b	331	3192	535	6442	6548	6198

Table A.19: MP MFLOPS NEON benchmarks results for 1 thread in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	308	227	181	262	264	293
Physical RPi3b+	1575	1498	414	5981	5872	5445
Emulated RPi4b	396	421	339	468	445	462
Physical RPi4b	4607	6186	552	13030	13012	8468

Table A.20: MP MFLOPS NEON benchmarks results for 2 threads in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	435	358	312	416	437	407
Physical RPi3b+	2217	2650	431	11661	11644	6061
Emulated RPi4b	557	523	499	797	791	710
Physical RPi4b	6279	5725	562	23798	24128	9374

Table A.21: MP MFLOPS NEON benchmarks results for 4 threads in MFLOPS.

	Op1 12.8KB	Op1 128KB	Op1 12,8MB	Op2 12.8KB	Op2 128KB	Op2 12.8MB
Emulated RPi3b+	339	390	352	435	442	387
Physical RPi3b+	2733	3197	437	10505	10637	6708
Emulated RPi4b	507	595	574	686	761	757
Physical RPi4b	7815	12044	486	22725	21712	9395

Table A.22: MP MFLOPS NEON benchmarks results for 8 threads in MFLOPS.

A.3. Single Thread Memory Benchmarks

BusSpeed

	16KB	32KB	64KB	128KB	256KB	512KB	1MB	4MB	16MB	64MB
Emulated RPi3b+	1554	1533	1546	2078	1929	1123	1345	1576	1631	1037
Physical RPi3b+	3870	3674	3602	3625	3622	2790	1636	1840	1642	1864
Emulated RPi4b	2504	2530	2552	2428	2573	2526	2465	2285	1997	2085
Physical RPi4b	8217	7507	7918	7874	7883	7716	5721	4111	4115	4036

Table A.23: BusSpeed (Increment = 1) benchmarks results in MB/sec.

	16KB	32KB	64KB	128KB	256KB	512KB	1MB	4MB	16MB	64MB
Emulated RPi3b+	2371	1892	2180	2209	2189	2011	1926	1639	848	1175
Physical RPi3b+	5089	4082	3350	3238	3164	2085	1070	881	977	989
Emulated RPi4b	2594	2813	2838	2732	2817	2673	2546	2446	1650	1557
Physical RPi4b	5891	5154	5163	5161	5265	5124	3283	2009	2071	2023

Table A.24: BusSpeed (Increment = 2) benchmarks results in MB/sec.

Memory Speed

	Op1 DP	Op1 SP	Op1 Int	Op2 DP	Op2 SP	Op2 Int	Op3 DP	Op3 SP	Op3 Int
Emul. RPi3b+	521	444.9	1107.6	787.7	713.9	1240.9	645.5	652.2	625
Phy. RPi3b+	2178.3	1862.6	2730.1	3586.5	2656.9	3177.7	3398.3	2985	2953.5
Emul. RPi4b	913.8	683.3	1845	1278.5	1092	1904.7	1184.3	1158.8	1165.5
Phys. RPi4b	9220.6	8480.5	7968.2	9226.2	8714.1	9115.36	7284.5	6489.6	5963

Table A.25: Memory Speed benchmarks results in MB/sec.

NEON Speed

	Op1 normal	Op1 NEON	Op2 normal	Op2 NEON	Op3 Float	Op3 Int
Emulated RPi3b+	264.8	335	467.8	681.5	415.6	845.5
Physical RPi3b+	1897.8	2949.2	2522.6	2815	3077.5	3113.1
Emulated RPi4b	419.3	574	613.5	1261.2	750.3	1344.7
Physical RPi4b	7396.7	8290	7061	7593.	8307.2	8429.8

Table A.26: NEON Speed benchmarks results in MB/sec.

A.4. Multiple Threads Memory Benchmarks

MP BusSpeed

	12.3KB 1T	12.3KB 2T	12.3KB 4T	12.3KB 8T	122KB 1T	122KB 2T
Emulated RPi3b+	1004	1904	2270	2081	1287	1547
Physical RPi3b+	1737	3361	6597	5195	1712	3387
Emulated RPi4b	1284	2472	2551	3016	1322	2438
Physical RPi4b	4221	7837	14794	13075	4152	8184

Table A.27: MPBusSpeed (Increment = 1) benchmarks results in MB/sec. - pt 1

	122KB 4T	122KB 8T	122MB 1T	122MB 2T	122MB 4T	122MB 8T
Emulated RPi3b+	2554	2809	911	557	1628	1341
Physical RPi3b+	4372	6514	1363	2182	2158	2051
Emulated RPi4b	4524	3027	1305	2324	2825	2952
Physical RPi4b	14617	13633	3616	6228	3789	3577

Table A.28: MPBusSpeed (Increment = 1) benchmarks results in MB/sec. - pt 2

	12.3KB 1T	12.3KB 2T	12.3KB 4T	12.3KB 8T	122KB 1T	122KB 2T
Emulated RPi3b+	937	2345	1936	1659	918	2559
Physical RPi3b+	1708	3303	6216	5350	1627	3268
Emulated RPi4b	1322	2384	2930	3032	1388	2276
Physical RPi4b	4115	7837	14794	13075	4152	8184

Table A.29: MPBusSpeed (Increment = 2) benchmarks results in MB/sec. - pt 1

	122KB 4T	122KB 8T	122MB 1T	122MB 2T	122MB 4T	122MB 8T
Emulated RPi3b+	2634	3377	462	696	733	1144
Physical RPi3b+	6427	6317	1016	946	975	1077
Emulated RPi4b	4255	4258	1165	1633	3054	3742
Physical RPi4b	14617	13633	3616	6228	3789	3577

Table A.30: MPBusSpeed (Increment = 2) benchmarks results in MB/sec. - pt 2

B | Paper Experiments Results

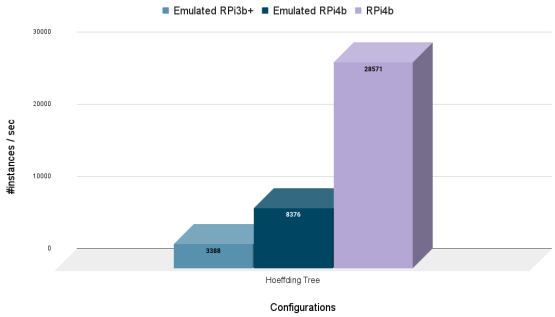

Figure B.1: Emulators throughputs for Exp 1.

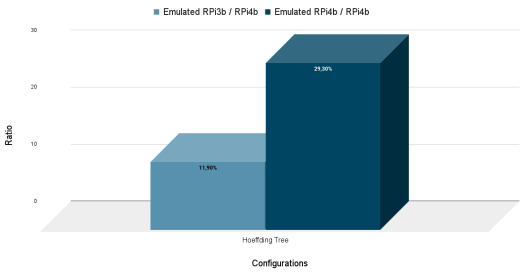

Figure B.2: Comparison of emulators and RPi4b performance on Exp 1.

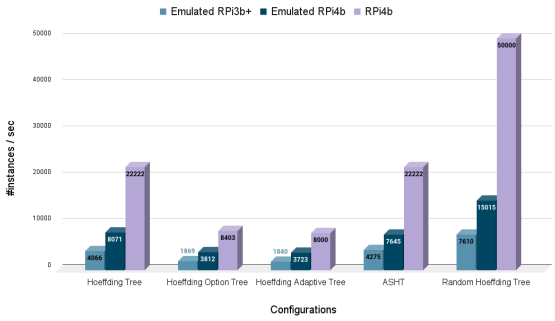

Figure B.3: Emulators throughputs for Paper Exp 2.

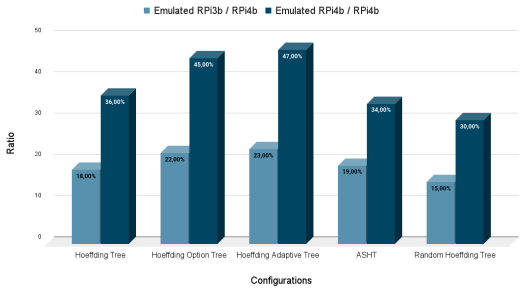

Figure B.4: Comparison of emulators and RPi4b performance on Exp 2.

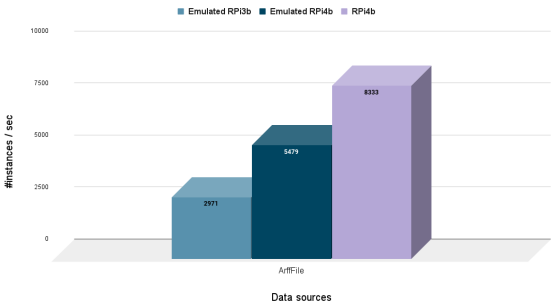

Figure B.5: Emulators throughputs for Paper Exp 4.

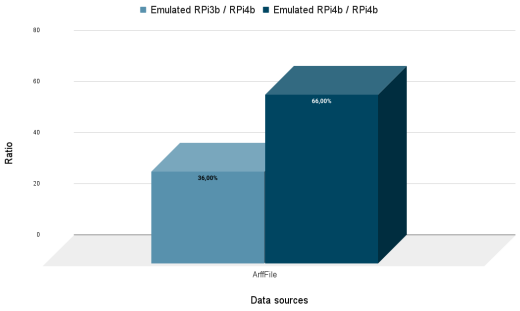

Figure B.6: Comparison of emulators and RPi4b performance on Exp 4.

List of Figures

List of Tables

Listings

List of Algorithms

Acknowledgements

I would like to express my sincere thanks to my thesis tutor at the Politecnico di Milano, Prof. Emanuele Della Valle, and his two research assistants and doctoral students Alessio Bernardo and Giacomo Ziffer, for this very great opportunity, their availability and their support throughout this work.

I also thank my ESIEE Paris thesis tutor, Prof. Jean-François Bercher, for his support, the transmission of his knowledge and his advice for over 2 years.

A huge thank you to my mother for her lifelong support, her patience, and all the sacrifices she did not hesitate to make to allow me to get to this point.

Finally, I would like to thank all those who accompanied me in the various trials of my journey from where they are.