LEARNING WORKLOAD BEHAVIOUR MODELS FROM MONITORED TIME-SERIES FOR RESOURCE ESTIMATION TOWARDS DATA CENTER OPTIMIZATION

DAVID BUCHACA PRATS

Dissertation submitted in partial fulfillment of the requirements for the Doctoral Degree in Computer Architecture

Universitat Politècnica de Catalunya - BarcelonaTECH

2020

David Buchaca Prats : *Learning workload behaviour models from monitored time-series for resource estimation towards data center optimization* .
Dissertation submitted in partial fulfillment of the requirements for the Doctoral Degree in Computer Architecture, 13th 2020.

SUPERVISORS:
David Carrera, Josep LLuís Berral

AFFILIATION:
Computer Architecture
Universitat Politècnica de Catalunya - BarcelonaTECH

LOCATION:
Barcelona

It would appear that we have reached the limits
of what it is possible to achieve with computer technology,
although one should be careful with such statements,
as they tend to sound pretty silly in 5 years.

John von Neumann

ABSTRACT

Modern Data Centers are complex systems that need management. As distributed computing systems grow, and workloads benefit from such computing environments, the management of such systems increases in complexity. The complexity of resource usage and power consumption on cloud-based applications makes the understanding of application behavior through expert examination difficult. The difficulty increases when applications are seen as "black boxes", where only external monitoring can be retrieved. Furthermore, given the different amount of scenarios and applications, automation is required. To deal with such complexity, Machine Learning methods become crucial to facilitate tasks that can be automatically learned from data.

This thesis demonstrates that learning algorithms allow relevant optimizations in Data Center environments, where applications are externally monitored and careful resource management is paramount to efficiently use computing resources. We propose to demonstrate this thesis in three areas that orbit around resource management in server environments.

Firstly, this thesis proposes an unsupervised learning technique to learn high level representations from workload traces. Such technique provides a fast methodology to characterize workloads as sequences of abstract phases. The learned phase representation is validated on a variety of datasets and used in an auto-scaling task where we show that it can be applied in a production environment, achieving better performance than other state-of-the-art techniques.

Secondly, this thesis proposes a neural architecture, based on Sequence-to-Sequence models, that provides the expected resource usage of applications sharing hardware resources. The proposed technique provides resource managers the ability to predict resource usage over time as well as the completion time of the running applications. The technique provides lower error predicting usage when compared with other popular Machine Learning methods.

Thirdly, this thesis proposes a technique for auto-tuning Big Data workloads from the available tunable parameters. The proposed technique gathers information from the logs of an application generating a feature descriptor that captures relevant information from the application to be tuned. Using this information we demonstrate that performance models can generalize up to a 34% better when compared with other state-of-the-art solutions. Moreover, the search time to find a suitable solution can be drastically reduced, with up to a 12x speedup and almost equal quality results as modern solutions.

These results prove that modern learning algorithms, with the right feature information, provide powerful techniques to manage resource allocation for applications running in cloud environments.

Contents

1 INTRODUCTION 1
1.1 Thesis context . . . 1
1.2 Motivations and research challenges . . . 2
1.3 Thesis Statement . . . 4
1.4 Thesis Contributions . . . 5
1.5 Document outline . . . 9
2 BACKGROUND 11
2.1 Conditional Restricted Boltzmann Machine . . . 11
2.2 Learnable vector representations . . . 15
2.3 Recurrent neural networks . . . 16
2.4 Bayesian Optimization . . . 19
3 RELATED WORK 21
3.1 Workload characterization and workload modelling . . . 21
3.2 Workload interference under co-scheduled scenarios . . . 23
3.3 Workload auto-tuning of Big Data workloads . . . 26
4 WORKLOAD PHASE CHARACTERIZATION THROUGH MACHINE LEARNING 29
4.1 Introduction . . . 29
4.2 Methodology . . . 30
4.3 Experiments: Evaluating Phase descriptor quality . . . 33
4.4 Experiments: Evaluating Phase descriptor in an auto-scaling task . . . 46
4.5 Conclusions . . . 50
5 SEQUENCE-TO-SEQUENCE MODELS FOR RESOURCE ESTIMATION OVER TIME UNDER CO-SCHEDULING. 53
5.1 Introduction . . . 53
5.2 Methodology . . . 56
5.3 Experiments . . . 59
5.4 Conclusions . . . 70
6 FAST TIME-TO-SOLUTION BIG DATA WORKLOAD AUTO-TUNING WITH REUSABLE PERFORMANCE MODELS 71
6.1 Introduction . . . 72
6.2 Methodology . . . 75
6.3 Experiments . . . 83
6.4 Conclusions . . . 94
7 CONCLUSION AND FUTURE WORK 95
7.1 Summary of the results . . . 95
7.2 Future Work . . . 97
BIBLIOGRAPHY 99

ACRONYMS

IoT Internet of Things

IT Information Technology

QoS Quality of Service

ML Machine Learning

SC Soft Computing

DL Deep Learning

NLP Natural Language Processing

RBM Restricted Boltzmann Machine

CRBM Conditional Restricted Boltzmann Machine

MLP Multi Layer Perceptron

RNN Recurrent Neural Network

BO Bayesian Optimization

EBM Energy-Based model

SGD Stochastic Gradient Descent

CD-k Contrastive Divergence

PCD-k Persistent Contrastive Divergence Algorithm

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

NN Neural Network

GP Gaussian Process

SVM Support Vector Machine

HMM Hidden Markov Model

ARIMA Autoregressive Integrated Moving Average

KNN K-Nearest Neighbor

SVD Singular Value Decomposition

CNN Convolutional Neural Network

SSW Square Sum Within clusters

OOM Out-of-Memory

SLA Service Level Agreements

HPC High Performance Computing

EoS End of Sequence

PC Percentage Completion

MAPE Mean Absolute Percentage Error

LR Linear Regression

SBO Simulated Bayesian Optimization

LOGO-CV Leave One Group Out Cross-Validation

MSE Mean Squared Error

NMT Neural Machine Translation

PCA Principal Component Analysis

List of Figures

Figure 2.1 Restricted Boltzmann Machine (RBM) diagrams: all connections between variables drawn (left), block based diagram (right). 11

Figure 2.2 CRBM diagram with a history length of n time steps. 14

Figure 2.3 The computation of $\boldsymbol{y}_t$ and $\boldsymbol{s}_t$ depends on $\boldsymbol{s}_{t-1}$ and $\boldsymbol{x}_t$. 17

Figure 4.1 Data pipeline schema showing how the resource monitoring data passes through the Conditional Restricted Boltzmann Machine (CRBM) and the clustering method. 33

Figure 4.2 Input trace behaviour for each phase value: $R1, \ldots, R5$ 38

Figure 4.3 Histograms of the normalized input features grouped by $R1, \ldots, R5$. 39

Figure 4.4 Two different workloads side by side. This example shows our phase descriptors do not exactly match the Hadoop phases. 41

Figure 4.5 Three different applications and the predicted phases. 43

Figure 4.6 Results for different random initialization of K-means. 45

Figure 4.7 Phase prediction on human body data. 45

Figure 4.8 Phase prediction via different models. 47

Figure 4.9 Example of DLaaS execution, with the provisioned CPU and Memory for each policy. 49

Figure 5.1 Input-output diagram of the proposed model 56

Figure 5.2 Example with two co-located applications that share resources only during a fraction of the execution. 64

Figure 5.3 Example where one application demands almost all memory available. . . 65

Figure 5.4 Example where both applications use all available memory at some point during the execution. 66

Figure 5.5 This figure shows the distribution of the Mean Absolute Percentage Error (MAPE) grouped into 4 categories. 68

Figure 5.6 Behaviour of End of Sequence (EoS) and Percentage Completion (PC) features for the workloads seen in Figures [5.2] [5.3] and [5.4]. 69

Figure 6.1 Classical optimization pipeline followed by a Bayesian Optimization process. 76

Figure 6.2 Classical optimization pipeline with a performance model (Oracle). 77

Figure 6.3 Proposed search pipeline for auto-tunning Big Data workloads. 78

Figure 6.4 Best solutions found by Simulated Bayesian Optimization (SBO)-1 and SBO-2. 92

Figure 6.5 Speedup with respect to the default configuration per application (the higher the better). An empty bar reflects no improvement over the default configuration. 92
Figure 6.6 Total search time, in minutes, to find a configuration for each method (the lower the better). The plot was cropped at 120 min to improve readability. . 93

List of Tables

Table 4.1 Example of data slice from the ALOJA dataset. For non-available values, −1 is used instead. 34
Table 4.2 CRBM training time, n_v is the number of hidden units. All models have the same delay, 50 time steps. 36
Table 4.3 Mean and standard deviation of the normalized traces under the different *regimes* given by the Hidden Markov Model (HMM). 39
Table 4.4 Accuracy results of the best alignment between true and predicted phases. 42
Table 4.5 Comparison of the prediction models. 47
Table 4.6 Average number of auto-scaling changes, number of Out-of-Memory (OOM) containers for each policy, and over-provisioning in *CPU_hour* and *KBytes_hour* 50
Table 5.1 Workload metrics recorded at each time step. 60
Table 5.2 Mean Absolute Percentage Error for each metric and model, evaluated in the test set 63
Table 5.3 MAPE errors made using the different stopping criteria. 68
Table 6.1 Task features. 80
Table 6.2 Applications used for the experiments. 84
Table 6.3 Search Space of the different parameters we tuned with the default values. The units in the Default Configuration correspond to the notation used by Spark. For example, 20g corresponds to 20 GB. 87
Table 6.4 Hyper-parameter search space. 89
Table 6.5 Training and validation Mean Squared Error (MSE) errors for different models. 90
Table 6.6 Test errors and improvement over the basic *sc* feature set. 90
Table 6.7 Time-to-solution for the different methods. The first column shows the improvement with respect to 20 runs of Bayesian Optimization. The second column the overall search time in minutes. 93

1

INTRODUCTION

1.1 THESIS CONTEXT

Internet services during the past two decades have grown in a steady step. The appearance of smartphones, Social Network platforms and other online communities has drastically increased the amount of data stored from users. The low manufacturing costs of single board computers have contributed to the expansion of the Internet of Things (IoT) which has populated the world with devices that continuously generate telemetric and sensor data.

The rise in data generation has changed the processing needs required by companies that want to extract knowledge or build products based on available data. Nowadays, big Information Technology (IT) companies offer computing and storage services which provide scalable hardware infrastructure capable to meet the current processing requirements of custormers. Therefore, efficient management of hardware and software has become a paramount concern to IT companies.

Service providers usually offer hardware resources subject to a Quality of Service (QoS) metric that ensures users a fair (and minimum) quality from the rented hardware resources. On the one hand, Cloud providers want to keep costs as low as possible, as long as the QoS is maintained. On the other hand, users want the most performance from the payed resources. As a consequence, the intelligent management of hardware and software becomes a key component of Data Centers to balance the cost and the quality provided to customers.

In order to increase the efficiency of hardware and software, automatic management of resources plays a key role in the control of many Cloud provider solutions. Nevertheless, the problem of mapping workloads on top of the hardware resources, with the goal of maximizing the performance of workloads as well as the utilization of resources, is a hard problem. A common approach has been to design heuristics that adapt to different contexts, providing solutions for a given workload mix and underlying infrastructure, but which cannot be easily generalized. When the workload mix is completely heterogeneous, the problem becomes even more challenging and needs to be automated.

The difficulties that engineers have to face in order to efficiently manage large-scale distributed computing systems have created the field of Autonomic Computing. This field now studies new methodologies to bring automation into the management of Data Centers. Some of the areas of research include: detecting and solving system failures, automating the deployment and scaling of services and applications on distributed systems and providing automatic configurations to efficiently run distributed software. In order to build such mechanisms, many of the solutions proposed in the area of Autonomic Computing use Machine Learning (ML) and Soft Computing (SC) techniques.

Machine Learning provides a methodology to build algorithms that learn from data in three types of scenarios. Supervised ML algorithms provide learnable functions that can make predictions of a vector of target variables from input observations. Unsupervised ML algorithms provide a mechanism to model the probability of a set of observations from a system or a mechanism to find interesting groups in the data. Reinforcement ML algorithms provide learnable mechanisms that reward agents depending on the actions they take in a given environment. These families of algorithms allow the creation of useful tools that can be automatically customised (and updated) to solve a particular task. In the Cloud environment, ML algorithms allow many useful applications to manage distributed systems. For example, automatic backup mechanisms of hard drives have been build using oracles that predict which drives are likely to fail. Besides, more efficient scaling mechanisms are also possible using ML models that can guess future resource demands, allowing the automatic hardware reconfiguration needed to maintain the QoS agreements of online services.

Soft Computing is an area of research that focuses on providing algorithms to tackle problems where imprecision, partial truth and uncertainty make classical exact solutions infeasible or impractical. This area of research studies, among many other topics, approximate search algorithms with the capability to find good solutions fast in very large search spaces. This type of search methods is crucial for several data centric problems such as workload placement and hyper parameter tuning of distributed applications.

1.2 MOTIVATIONS AND RESEARCH CHALLENGES

Data Centers and Cloud providers provide computing resources to users that submit jobs to a computing cluster. When a job is submitted a resource manager decides how long the job waits in a queue until it is scheduled and executed. The time spent in the queue, or wait time, depends on several factors including job priority, current load on the system, and availability of requested resources. Once the program is executed it uses a specific budget of assigned hardware resources which have to be either determined by the user or the hardware provider. Managing this mapping from software to hardware is not an easy

task since there are many factors that impact on the decision. Moreover, resource utilization during the lifetime of the job represents the actual useful work that has been performed.

In a typical production environment, many different jobs are submitted. These jobs can be characterized by features containing: number of processors requested, priority level, estimated runtime, parallel or distributed execution and specific I/O requirements. The total number of possible combinations of those features becomes easily unmanageable. In order to provide an adequate service for such variety of software some system administrators create different types of queues, each with a different priority level and available hardware resources.

Job scheduling and container autoscaling are relevant but difficult tasks because they might depend on a lot of variables. Both tasks are important because they are needed to efficiently use the hardware resources. Morover, they are difficult because they involve many topics, such as: job priority, expected execution time, resource access permission, resource availability and minimum hardware resources to reach a particular QoS are variables that can impact both scheduling and autoscaling decisions.

In this environment, workloads are usually seen as black boxes. Hardware providers do not have access to the source code of the applications that are remotely executed. Nevertheless, they have access to the metrics produced by the machines in their cluster. Such metrics, ofter referred to as execution traces, contain valuable information that can be stored and reused. Reusing historical executions to produce automatic decision mechanism has been a priority in Autonomic Computing area. The high variety of workloads and the temporal dimensionality of the data makes this a difficult task. This complexity brings us to the first research challenge:

- Research Challenge 1: We want to extract behavioural patterns from workload traces automatically. In particular, we would like to find simpler workload trace representations of running applications to faciliate automatic decision making.

When a cluster is under heavy load, the capability to provide a fair portion of the cluster as resources to each user is important. This capability can be provided by using the fair-share strategy, in which the scheduler collects historical data from previously executed jobs and uses the historical data to dynamically adjust the priority of the jobs in the queue, as well as any hardware resource sharing among workloads. Using historical data the scheduler can dynamically make priority changes to ensure that resources are fairly distributed among users. Nevertheless, to efficiently use the finite hardware resources available, resource sharing is needed. When co-executions occur, resource managers need to decide which applications can be executed sharing resources. In order to decide which applications can be co-scheduled resource managers need to estimate the impact of the decision. Some applications might interfere a lot (or even crash) while others can be executed seamlessly sharing resources. This brings us to our second research challenge:

- Research Challenge 2: We want to estimate the performance of applications sharing resources in order to select workloads that do not penalise each other in excess.

The previous two research challenges emphasize tasks that are based on managing software provided as a black box. This means that software is not tunable by the resource manager. In some scenarios, applications can be adapted through tunable software properties. Tunable properties allow software to be adjusted to different data sizes and hardware architectures. Big Data software that requires the scheduling of parallel jobs executed on top of frameworks such as Hadoop or Spark can be adapted through configuration files. These frameworks run parallel jobs that are made up of multiple subtasks. Each subtask is assigned to a unique compute node during execution. During execution nodes constantly communicate among themselves. The manner in which the subtasks are assigned to processors depends on the tunable properties of the software, the type of workload and the size of the data. In order to deal with the great variety of workload types, Big Data frameworks for parallel workloads, such as Hadoop or Spark, provide users hundreds of tunable parameters to adapt the wide variety of applications that benefit from massively parallel hardware resources. The huge number of possible parameter combinations makes this a hard task.

- Research Challenge 3: We want an efficient way to auto tune Big Data workloads that can improve the time-to-solution of current methodologies.

1.3 THESIS STATEMENT

Many of the current solutions for data centric problems proposed by the Autonomic Computing community are based on classical ML and SC techniques, which have some limitations. Nevertheless, during the past five years there has been a resurgence of connectionist models, already studied by the ML community since the 80s, that have materialised into new neural network models that provide previously unheard capabilities, creating the Deep Learning (DL) community. Most of these new learning methods have been applied to solve many computer vision and Natural Language Processing (NLP) tasks with impressive results. Nevertheless, few revolutionary applications of such methods can be found in the data centric community.

This thesis wants to demonstrate that **it is possible to characterize application behaviour, estimate workload colocation interference and efficiently asses the quality of software dependable parameters of applications seen as black boxes using learning algorithms.**

To prove this statement we develop solutions using ML and DL algorithms that improve state of the art solutions on the following problems: workload characterisation, workload resource estimation under co-scheduling and automatic hyper-parameter selection for distributed workloads.

The three problems studied in this thesis are paramount for the efficient use of hardware and software resources in data centric environments. Workload characterisation is important because it provides a building block to understand the hardware needs of an application and the different phases that might occur during the job execution. Workload resource estimation under co-scheduling environments is paramount to efficiently manage shared hardware resources. Hyper-parameter selection is crucial for the efficient use of distributed workloads based on technologies such as Spark, which offer great customisation capabilities at the expense of managing many tunable parameters.

1.4 THESIS CONTRIBUTIONS

The work presented in this thesis is based upon three contributions that revolve around improving data centric management techniques. Each contribution targets a particular research challenge. The three main contributions proposed in this thesis can be summarised as follows:

- C1: Create a novel methodology to learn a low dimensional descriptor used to characterize workload behaviour over time.
- C2: Propose a new technique to predict resource estimation over time in co-scheduling scenarios using a neural network architecture based on sequence-to-sequence models.
- C3: Provide a fast time-to-solution auto-tuning technique that improves current machine learning methodologies through a low level characterisation of Big Data workloads mined from logs of the application.

These three contributions, which can be considered partially independent of each other, target a more general goal: provide resource managers new capabilities to improve current Data Center decision making systems. The rest of this chapter provides an overview of each of the previous contributions.

1.4.1 *Workload phase characterization through machine learning*

The **first contribution** of this thesis is a study that aims to build a learning algorithm that can find different behaviour phases in workload traces. In Cloud environments applications are often provided as black-boxes. Existing literature has studied the behavior of applications

working on the assumption that different but recurrent behaviours, also known as phases, occur during the course of the execution. This phases can be used to describe an application (as a composition of phases) which can be used to schedule by means of decomposing application into phases instead of looking at the complete runtime. Many state of the art techniques propose invasive techniques that place phase markers in the applications source code, but this methodology provides more work to programmers and it not suitable for environments where applications are provided as black-boxes. To solve this issue we developed a pipeline based on a Conditional Restricted Boltzmann Machine (CRBM) that models slices of workload traces with the main goal to build a simpler representation for the workload traces that captures certain learned resource usage patterns. This new representation provides a lower dimensional descriptor of the workload trace that can be used to extract job execution phases over time.

Results showed that the proposed solution can find meaningful phases in the workload traces. Our methodology provides a mechanism to reduce the telemetry data from N features of the input to a single class label per time step. A thorough analysis of the application phases shows that different behaviours occur under different class labels, even if the provided labels are not easy to interpret, they provide a mechanism to drastically reduce the feature space which should help avoiding the curse of the dimensionality on any machine learning model that is applied on the data that is generated by our solution. The curse of the dimensionality states that, as the number of feature dimensions grow, the amount of data needed to make machine learning models generalize accurately grows exponentially. The goal of this contribution is to facilitate future scheduler policies and learning methods that can be build on top of the provided features. This work is inspired in previous attempts to use standard Restricted Boltzmann Machine (RBM) models to build features that facilitate learning over them, instead of using the original input features directly.

The work carried out in this contribution produced the following publication:

> [63] David Buchaca Prats, Josep Lluís Berral, and David Carrera. Automatic Generation of Workload Profiles Using Unsupervised Learning Pipelines. In *2018 IEEE Transactions on Network and Service Management (TSNM)*, pp. 142-155. doi: 10.1109/TNSM.2017.2786047.

After validating the phase mechanism in the publication [63], we have used the presented descriptor to build an auto-scaling mechanism for ML workloads. Our system uses a Multi Layer Perceptron (MLP) that takes as input a time-window of past phase ids and learns to predict the following phase values. We use this model to predict phase transitions from containers running different types of workloads. The MLP allows us to foresee sudden phase changes ahead of time, which allow us to build an auto-scaling policy, based on

the predictions of the MLP, to resize the container dynamically according to the phase predictions. This work has produced the following publication:

> David Buchaca Prats, Josep Lluís Berral, Chen Wang, Alaa Youssef. Proactive Container Auto-scaling for Cloud Native Machine Learning Services. In *2020 IEEE Cloud*

1.4.2 *Sequence-to-Sequence models for resource estimation over time under co-scheduling scenarios*

The **second contribution** of this thesis studies the use of Recurrent Neural Network (RNN) models to provide resource estimation over time when two applications are co-located. The motivation for this work is to be able to generate information that is currently not being provided to schedulers: how applications might interact over time.

Current ML techniques for resource estimation of co-located applications are based on extracting features from two applications and predicting a single point estimate. Given two applications $\boldsymbol{a_1}$,$\boldsymbol{a_2}$ and a model h, current state of the art works use machine learning models that provide as output a single vector $h(\boldsymbol{a_1}, \boldsymbol{a_2}) \in \mathbb{R}^n$. This vector captures information about the n estimated resources and contains the expected resource estimation for the application pair. Therefore, this type of modelling does not provide any information about how resources are required through time, making the scheduler unaware of possible phase change requirements during the execution of an application. To improve upon this setting we approach this problem differently. Our goal is to build a system capable to predict t vectors that estimate the resource usage demands over time. That is, we want a model h such that it produces $h(\boldsymbol{a_1}, \boldsymbol{a_2}) \in \mathbb{R}^n \times \mathbb{R}^t$. We adapt a Sequence-to-Sequence model with some peculiarities that arise in our use case. A relevant problem that arises when predicting resources over time is finding how many vectors the Sequence-to-Sequence model needs to generate. To do this a stopping criteria is needed. In order to build a stopping criteria to decide the length of the output signal, we investigated different mechanisms and proposed a continuous feature that provides better quality results than other State of the Art approaches. With this work, resource managers can receive extra information that is currently not generated which would provide more knowledge to better assign hardware resources.

Results showed that the proposed method can predict resource usage over time, even when the output trace is longer than the input traces because there is a high competition of resources. In such cases, classical machine learning regression techniques are not appropriate because the output length is not known a-priori. Therefore, standard regression techniques that cannot generate a sequence of arbitrary length, cannot be easily applied in this scenario. Our method is capable to generate an output trace of unbounded length that we need to stop using a heuristic.

The work carried out in this contribution has been published in:

[12] David Buchaca Prats, Joan Marcual, Josep LLuís Berral, and David Carrera. Sequence-to-sequence models for workload interference prediction on batch processing datacenters. In: *2020 Future Generation Computer Systems 13* , pp. 1-13. issn: 0167739X. doi: 10.1016/ j.future.2020.03.058.

1.4.3 *Fast time-to-solution big data workload auto-tuning with reusable machine learning models*

The **third contribution** of this dissertation studies how to build an auto-tuning system for Big Data Frameworks using reusable Machine Learning performance models. The motivation for this work is improve current auto-tuning systems providing a mechanism for reusing models across different workloads. Most state of the art solutions for tuning Big Data workloads are build using model based search techniques. Such solutions are build using perfomance models that receive as input a configuration of hyper-parameters and predict as output a variable that the user wants to optimize (such as the execution time of the application). Given a search algorithm and an oracle, a model based search method uses the predictions of the oracle in order to asses the quality of the examples provided by the search mechanism. One of the main difficulties with this type of systems is the fact that they are workload specific. In order to be able to find a configuration for a workload a model is trained specifically for that application and then it is used to perform the search. This involves a lot of time that is "wasted" during learning.

In order to improve the previously mentioned issues some model based search solutions include other information as input to the oracle. Information such as an application id or the dataset size that the program needs to process have been shown to improve results. This type of information allows oracles to be reused for multiple dataset sizes and for a variety of applications. Nevertheless, once a new application is presented to the system, a new feature has to be added and the model needs to be retrained for the new application. To avoid this issues we propose to build a feature descriptor based on low level features found in the logs of the application. The logs of the application contain information describing low level metrics as well as a description of how the computation is distributed in tasks. Using all this information we can build a vector that characterizes the workload and can be used as an input to the oracle, to condition the predictions on the type of workload that we aim to optimize. We test our methodology using Spark, which is one of the most popular Big Data frameworks.

Results show that, the feature descriptor we create enhances the learning algorithms during train and test predictions. Moreover, compared to other search based state of the

art solutions, such as Bayesian Optimization (BO), our methodology provides faster time-to-solution while using less resources with a similar quality in the final results.

The work carried out in this third contribution has produced the following accepted publication (still not published):

> David Buchaca Prats, Felipe Portella, Carlos Costa, Josep LLuís Berral. You Only Run Once: Spark Auto-tuning from a single run. Submitted to: *IEEE Transactions on Network and Service Management (TSNM)*

1.5 DOCUMENT OUTLINE

The remainder of this thesis is organized as follows:

Chapter 2 provides some Background in relevant areas and techniques relevant to this thesis: Conditional Restricted Boltzmann Machines, Recurrent Neural Networks, Sequence-to-Sequence and Bayesian Optimization.

Chapter 3 provides related work on the topics discussed in this dissertation: workload modelling, workload interference and resource estimation of co-scheduled applications and workload auto-tuning.

Chapter 4 provides a description and evaluation of the first contribution used to automatically generate a profile from an application resource trace using the hidden activations of a Conditional Restricted Boltzmann machine.

Chapter 5 provides a description and evaluation of the second contribution used to predict the resource usage of two co-located applications over time, as well as the overal execution time of the co-scheduled pair.

Chapter 6 provides a description and evaluation of the third contribution used to improve the time-to-solution of Big Data auto-tunning systems. This contribution is validated leveraging features mined from the SparkEventLog generated by Spark workloads.

Chapter 7 presents conclusions the overall results that link the relevance of the different contributions in the context of management of Data Centers. To finish the document, a brief discussion of possible future works is included.

2

BACKGROUND

2.1 CONDITIONAL RESTRICTED BOLTZMANN MACHINE

Restricted Boltzmann Machine

A RBM [28] is a generative artificial neural network that can learn a probability distribution from a set of input examples. An RBM is an example of an Energy-Based model (EBM) with a particular type of energy function and a specific formula for computing probabilities using the energy. As their name suggest, EBM models use an energy function to model the dependencies between variables in the model. This is done associating a scalar energy to each configuration of the variables in the model.

The RBM uses a set of visible variables $\mathbf{V}$ to present information to the model and a set of latent variables $\mathbf{H}$ to capture dependencies in the different input regions. Both $\mathbf{V}$ and $\mathbf{H}$ can have any number of variables which we will denote with n_v and n_h respectively. Figure 2.1 shows two diagrams of an RBM.

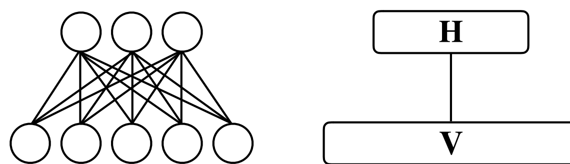


Figure 2.1: RBM diagrams: all connections between variables drawn (left), block based diagram (right).

The RBM is characterized by an energy function $E(\boldsymbol{v}, \boldsymbol{h})$ which measures the agreement between the visible and hidden vectors of a configuration pair $(\boldsymbol{v}, \boldsymbol{h})$. The energy function can be adjusted to match different characteristics of the input data. Equations (2.1a) and (2.1b) show the most common energy functions. In the case of binary input data most works use (2.1a). Nevertheless, for real valued data (2.1b) is preferred. Note that both equations are defined using the bias vector of the input variables $\boldsymbol{b}^v \in \mathbb{R}^{n_v}$, the bias vector of the latent variables $\boldsymbol{b}^h \in \mathbb{R}^{n_h}$ and the weight matrix $\boldsymbol{W} \in \mathbb{R}^{n_h \times n_v}$. With these three quantities we

define $\boldsymbol{\theta}_{RBM} := \{\boldsymbol{b}^v, \boldsymbol{b}^h, \boldsymbol{W}\}$ as the set of learnable parameters of the model. In the case of real valued data $\boldsymbol{\sigma}$ is usually left as a vector of ones as explained in [77].

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^\top \boldsymbol{b}^v - \boldsymbol{h}^\top \boldsymbol{b}^h - \boldsymbol{h}^\top \boldsymbol{W} \boldsymbol{v} \tag{2.1a}$$

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\frac{1}{2} \left\| \frac{\boldsymbol{v} - \boldsymbol{b}^v}{\boldsymbol{\sigma}} \right\|^2 - \boldsymbol{h}^\top \boldsymbol{b}^h - \boldsymbol{h}^\top \boldsymbol{W} \frac{\boldsymbol{v}}{\boldsymbol{\sigma}} \tag{2.1b}$$

Using the previous energy function the RBM defines the probability of a configuration pair $(\boldsymbol{v}, \boldsymbol{h})$ with the following formula:

$$P(\boldsymbol{v}, \boldsymbol{h}) = \frac{\exp\left(-E(\boldsymbol{v}, \boldsymbol{h})\right)}{\sum_{(\tilde{\boldsymbol{v}}, \tilde{\boldsymbol{h}})} \exp(-E(\tilde{\boldsymbol{v}}, \tilde{\boldsymbol{h}}))} = \frac{\exp\left(-E(\boldsymbol{v}, \boldsymbol{h})\right)}{Z} \tag{2.2}$$

where Z is a normalization constant. Note that marginalizing over the hidden units in Equation (2.2) we get the probability of a visible configuration. That is:

$$P(\boldsymbol{v}) = \sum_{h} P(\boldsymbol{v}, \boldsymbol{h}) \tag{2.3}$$

Training Restricted Boltzmann Machine

Learning using RBM models consist on finding a set of parameters $\{\boldsymbol{b}^v, \boldsymbol{b}^h, \boldsymbol{W}\}$ that provide high probability to the training data. Let us assume we have a training set $\boldsymbol{X}$ containing M examples. Learning consist on finding a good configuration of the model parameters that maximizes the probability of the data. Maximizing the probability of the training data can be represented as maximizing the likelihood of the M training examples, where the likelihood is a function that computes the product of the probabilities of all the training examples. Note that maximizing the likelihood is equivalent to minimizing the negative likelihood function. Therefore, learning can achieved using a Stochastic Gradient Descent (SGD) algorithm on the average negative log-likelihood of the data:

$$-\frac{1}{M} l(\boldsymbol{\theta}; \boldsymbol{X}) := -\frac{1}{M} \log \left(\prod_{m=1}^{M} P(\boldsymbol{x}^{(m)}; \boldsymbol{\theta}) \right) = -\frac{1}{M} \sum_{m=1}^{M} \log \left(P(\boldsymbol{x}^{(m)}; \boldsymbol{\theta}) \right) \tag{2.4}$$

The SGD algorithm starts with a random guess of all parameters which are iteratively updated using $\frac{\partial}{\partial \theta} \log(P(\boldsymbol{x}^{(m)}; \boldsymbol{\theta}))$, making small updates for each parameter θ. It can be

shown [29] that the expression of this derivative can be written as a subtraction of two terms:

$$\frac{\partial}{\partial\theta}\left(-P(\boldsymbol{x}^{(m)};\boldsymbol{\theta})\right) = \mathbb{E}_{\boldsymbol{h}}\left[\frac{\partial}{\partial\theta}E(\boldsymbol{x}^{(m)},\boldsymbol{h})\right] - \mathbb{E}_{\boldsymbol{x},\boldsymbol{h}}\left[\frac{\partial}{\partial\theta}E(\boldsymbol{x},\boldsymbol{h})\right] \tag{2.5}$$

The first term, usually referred to as the positive phase, can be computed efficiently. The second term, usually referred to as the negative phase, is an expectation over the visible and hidden units, which becomes intractable if the number of hidden or visible units is big. To overcome the computational complexity of computing the negative phase [28] proposed the Contrastive Divergence (CD-k) algorithm.

Contrastive Divergence

The CD-k algorithm [28] takes k steps of Gibbs Sampling to generate a sample $\tilde{\boldsymbol{x}}$. This process starts clamping a training example $\boldsymbol{x}$ and sampling hidden and visible units alternatively k times. This process generates $\boldsymbol{x}_{[1]}, \ldots, \boldsymbol{x}_{[k]}$ vectors, and the last one is defined to be the sample $\tilde{\boldsymbol{x}}$ that is used as a "representative" for the negative phase. We call it "representative" because it represents the expectation over all possible visible configurations. That is $\tilde{\boldsymbol{x}} := \boldsymbol{x}_{[k]}$.

Persistent Contrastive Divergence

The Persistent Contrastive Divergence Algorithm (PCD-k) algorithm [80] consist on a small modification of the CD-k algorithm. PCD-k takes k steps of Gibbs Sampling to generate $\tilde{\boldsymbol{x}}$, as CD-k does. Instead of starting the Gibbs Chain in a training example $\boldsymbol{x}$, the process reuses the last previously sampled visible vector $\boldsymbol{x}_{[k]}$ to start the sampling process. The reuse of the vector $\boldsymbol{x}_{[k]}$ gave rise to the name "persistent" because the chain "persists" over SGD updates. Note that this process generates a sequence of samples $\tilde{\boldsymbol{x}}_{[1]}, \ldots, \tilde{\boldsymbol{x}}_{[k]}$, which start from a visible clamped vector $\tilde{\boldsymbol{x}}$. After every SGD update the persistent visible vector is updated. That is $\tilde{\boldsymbol{x}} := \tilde{\boldsymbol{x}}_{[k]}$.

2.1.1 *Conditional Restricted Boltzmann Machine*

A CRBM [77] is a RBM with Gaussian visible units and binary hidden units with some extra connections used to model temporal dependencies. To model such dependencies, the CRBM keeps track of the n previous visible vectors, which are kept in a sliding window $\mathbf{His}^n$. We will call $\mathbf{His}^n$ the *history* of the CRBM. The CRBM was designed to learn sequential patterns such as human body motion [77] or pidgeon movement [93] over time.

The parameters of the CRBM are $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{A}, \boldsymbol{D}, \boldsymbol{c}, \boldsymbol{b}\}$, where $\boldsymbol{W}, \boldsymbol{A}, \boldsymbol{D}$ are matrices and $\boldsymbol{c}$, $\boldsymbol{b}$ are the vectors of biases for the visible and hidden units units, respectively. Let us assume we have input vectors defined with n_v features and we use n_h hidden units in the CRBM . Matrix $\boldsymbol{W} \in \mathbb{R}^{n_h \times n_v}$ models the connections between visible and hidden units, which is the equivalent to the $\boldsymbol{W}$ matrix from a standard RBM . Note that $\boldsymbol{A}$ and $\boldsymbol{D}$ were not present in an RBM . Matrix $\boldsymbol{A} \in \mathbb{R}^{n_v \times (n_v \cdot n)}$ is the mapping from the history to the visible units. Matrix $\boldsymbol{D} \in \mathbb{R}^{n_h \times (n_v \cdot n)}$ is the mapping from the history to the hidden units.

Let us consider a multidimensional time-series $\boldsymbol{v} = (\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \dots)$ where $\boldsymbol{v}_k \in \mathbb{R}^{n_v}$. The history for $\boldsymbol{v}$ at time t, denoted by $\mathbf{His}_t^n$, is defined as $(\boldsymbol{v}_{t-n}, \dots, \boldsymbol{v}_{t-1})$ and contains the previous n vectors from time $t-1$ to $t-n$. Note that the definition of $\mathbf{His}_t^n$ uses t to index the n vectors inside the history. Given $\mathbf{His}_t^n$ we can update $\mathbf{His}_{t+1}^n$ shifting the n vectors from $\mathbf{His}_t^n$ one unit to the left and adding a new vector in the last position. In other words, at time $t+1$, vector $\boldsymbol{v}_t$ is pushed into the history while observation $\boldsymbol{v}_{t-n}$ is popped out. Therefore $\mathbf{His}_{t+1}^n$ is $(\boldsymbol{v}_{t-n-1}, \dots, \boldsymbol{v}_t)$. Notice again that such a mechanism needs the first n observations of each time-series to have enough data to properly fill the history. Figure 2.2 shows a diagram of the CRBM.

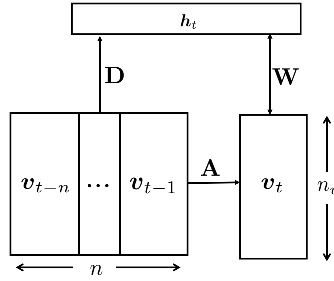


Figure 2.2: CRBM diagram with a history length of n time steps.

Given a vector $\boldsymbol{v}$, we define the hidden activation $\boldsymbol{h}$ as the sigmoid of the incoming signal from $\boldsymbol{v}$ and $\mathbf{His}^n$, weighted by $\boldsymbol{W}$ and $\boldsymbol{D}$, respectively, and adding the bias of hidden units. That is:

$$\boldsymbol{h}_{(n_h,1)} = \sigma\left(\boldsymbol{W}_{(n_h,n_v)} \cdot \boldsymbol{v}_{(n_v,1)} + \boldsymbol{D}_{(n_h,n_v\cdot n)} \cdot \mathbf{His}^n_{(n_v\cdot n,1)} + \boldsymbol{b}_{(n_h,1)}\right) \tag{2.6}$$

the subscripts in Eq. (2.6) indicate the dimensions of the different matrices and vectors. For brevity, we will express this without subscripts as

$$\boldsymbol{h} = \sigma\left(\boldsymbol{W} \cdot \boldsymbol{v} + \boldsymbol{D} \cdot \mathbf{His}^n + \boldsymbol{b}\right)$$

Note that $\boldsymbol{D}$ defines a function from $\mathbb{R}^{n_v \cdot n}$ to $\mathbb{R}^{n_h}$ and $\mathbf{His}^n$ is expressed as a column vector of length $n_v \cdot n$ instead of a matrix of shape (n_v, n).

Training a CRBM is similar to training an RBM . We can use a SGD iterative algorithm to find parameters that yield a low negative log-likelihood. A common method to find approximate gradients of the loss with respect to the parameters is the CD-k algorithm. More details about the fundamentals of CRBMs can be found in Taylor's work [77].

2.2 LEARNABLE VECTOR REPRESENTATIONS

Machine learning techniques that learn vector representations from raw data, sometimes referred to as embeddings, have been successfully applied for a wide variety of tasks. Some tasks include highly unrelated sources of data such as lattice-protein [83] or natural text [52]. Learning a vector representation is, in many cases, a preprocessing step to facilitate another task. For example, the vector representation learned by Word2Vec [52] has been used to map words to dense feature vectors to solve tasks such as Named Entity Recognition [38, 48].

We will briefly describe how Word2Vec works because it provides inspiration for learning representations in a domain that shares the sequential nature of the multi-dimensional time-series logs that we will use in Chapter 4.

Word2Vec uses a slight modification of a MLP with two hyper-parameters $D \in \mathbb{N}$ and $n \in \mathbb{N}$ that can be tuned by the user. The first parameter D is the size of the vector representation for the words, which corresponds to the number of neurons in the first hidden layer. The second parameter n corresponds to the size of the sliding window used to train the model. The objective of the learning task is to find a matrix $\boldsymbol{W} \in \mathbb{R}^{D \times V}$, where V is the size of the vocabulary. In order to train the model, the text is mapped into a tabular format using a sliding window over corpus that divides each sentence to a bunch of sub-sentences of n words. The set of all the sub-sentences of size n is the training data for Word2Vec. One approach to construct the train set consist on generating sub-sequences of n consecutive words. Let us consider a sequence $(w_1, w_2, w_3, w_4, w_5)$ where each w_i represents a word. Let us assume $n = 3$. In this example, we could generate two training examples $(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)})$ constructed as follows:

$$
\begin{aligned}
(\boldsymbol{x}^{(1)}, y^{(1)}) &= ((w_1, w_2, w_3), w_4) \\
(\boldsymbol{x}^{(2)}, y^{(2)}) &= ((w_2, w_3, w_4), w_5)
\end{aligned}
$$

In Word2Vec each word is mapped into an index, making the input of the model a fixed size vector of V components. Let $\mathrm{w2p}_V$ be a bijection that assigns each word in the vocabulary to a unique integer in $\{1, \ldots, V\}$. Given a word $w \in V$ the one-hot representation of w is defined as $v_w := (0, \ldots, \mathrm{w2p}_V(w), \ldots, 0) \in \mathrm{R}^V$. This is the word representation used as input to Word2Vec.

The learning process of Word2Vec consist on training the MLP to predict v_{w_t} given the input words $(v_{w_{t-n}}, \ldots, v_{w_{t-1}})$ across the training examples. After learning, the columns of

$\boldsymbol{W}$ become the vector representations of the V words in the vocabulary. Since each column in $\boldsymbol{W}$ is a vector of size D we say that this representation is an embedding $\phi : \mathrm{R}^V \longrightarrow \mathrm{R}^D$ that maps V-dimensional vectors to D-dimensional vectors. This function is defined as $\phi(v_w) := \boldsymbol{W}[\cdot, \mathrm{w2p}(w)]$. In order words, a word w that is assigned to position k (according to w2p), is embedded to the k'th column of $\boldsymbol{W}$.

Note that the previously defined process to generate training examples in Word2Vec resembles the sliding window mechanism detailed in the previous Section 2.1.1. We could interpret $\boldsymbol{x}^{(1)}$ as $\mathbf{His}_1^3$. That is, the history of a CRBM placed at the first position of the training example $(w_1, w_2, w_3, w_4, w_5)$ with length 3. In this example, w_4 would correspond to the visible vector of the CRBM. Nevertheless, there are important differences between these two approaches: the words from which Word2Vec is learned are essentially categorical values represented in a space of dimension V, whereas the CRBM is trained with real valued data.

2.3 RECURRENT NEURAL NETWORKS

A RNN is a function that maps a sequence of input vectors into an output vector. Specifically, the RNN takes a sequence $\boldsymbol{x}_{1:t}$, where each $\boldsymbol{x}_j \in \mathbb{R}^{d_{in}}$, and returns a single output vector $\boldsymbol{y}_t \in \mathbb{R}^{d_{out}}$, depending on a set of parameters $\boldsymbol{\theta}$ that need to be learned. Equation (2.7a) denotes the output of the RNN at time t, while equation (2.7b) denotes the whole sequence of outputs $\boldsymbol{y}_{1:n}$ produced by applying (2.7a) at every time step and concatenating the output results.

$$\boldsymbol{y}_t = \mathrm{RNN}(\boldsymbol{x}_{1:t}; \boldsymbol{\theta}) \tag{2.7a}$$

$$\boldsymbol{y}_{1:n} = \mathrm{RNN}^*(\boldsymbol{x}_{1:n}; \boldsymbol{\theta}) \tag{2.7b}$$

Output $\boldsymbol{y}_t$ is used to predict relevant information about the problem at hand at time t. Since we use the RNN to predict the expected resource demand of two concurrent applications at time t, $\boldsymbol{y}_t$ will be a vector containing the resources used by the two co-located applications.

In order to keep information from previously seen elements in sequences, RNNs have a hidden state $\boldsymbol{s}_t$ that is updated at every time step. The hidden state influences the predictions of the model over time. Although the hidden state is implicit in the equations (2.7a) and (2.7b), we can make the state explicit by using the notation $(\boldsymbol{y}_t, \boldsymbol{s}_t) = \mathrm{RNN}(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}; \boldsymbol{\theta})$. This indicates that the output $\boldsymbol{y}_t$ has been computed through input $\boldsymbol{x}_t$ and previous state $\boldsymbol{s}_{t-1}$. Figure 2.3 shows a diagram of an RNN.

Notice that while $\boldsymbol{y}_t$ (the output at time t) is not affecting $\boldsymbol{y}_{t+1}$, the updated state $\boldsymbol{s}_t$ will affect next state $\boldsymbol{s}_{t+1}$. The initial state $\boldsymbol{s}_0$ is usually defined as a vector of zeros, unless some other known contextual information can be provided for the model. The way in which $\boldsymbol{s}_t$ is computed is what mostly differentiates models such as a standard RNN from other

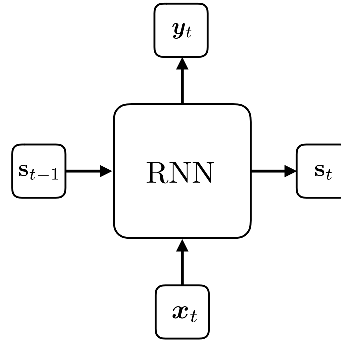


Figure 2.3: The computation of $\boldsymbol{y}_t$ and $\boldsymbol{s}_t$ depends on $\boldsymbol{s}_{t-1}$ and $\boldsymbol{x}_t$.

models such as a Gated Recurrent Unit (GRU) [17] or a Long Short-Term Memory (LSTM) [30]. Independently of how the hidden state is computed, RNNs are usually trained by an optimization algorithm such as SGD which minimizes the difference between the predicted outputs and the true outputs.

2.3.1 *Gated Recurrent Units*

GRU models are a type of RNN designed to avoid gradient vanishing problems [17]. The GRU is one of the simplest RNNs employing a gating mechanism that enables the network to learn which information needs to be kept over time and which can be forgotten. Equations (2.8a) to (2.8d) define, at time t, the output state $\boldsymbol{s}_t$, given the input data $\boldsymbol{x}_t$ and the previous state $\boldsymbol{s}_{t-1}$. In the presented equations, σ is the sigmoid function. Note that both sigmoid and tanh have a similar behaviour. The sigmoid function maps values from $(-\infty, \infty)$ to (0,1) and tanh maps $(-\infty, \infty)$ to (-1,1).

$$\boldsymbol{z}_t = \sigma\left(\boldsymbol{W}^{xz}\boldsymbol{x}_t + \boldsymbol{W}^{sz}\boldsymbol{s}_{t-1} + \boldsymbol{b}^{z}\right) \tag{2.8a}$$

$$\boldsymbol{r}_t = \sigma\left(\boldsymbol{W}^{xr}\boldsymbol{x}_t + \boldsymbol{W}^{sr}\boldsymbol{s}_{t-1} + \boldsymbol{b}^{r}\right) \tag{2.8b}$$

$$\tilde{\boldsymbol{s}}_t = \tanh\left(\boldsymbol{W}^{xs}\boldsymbol{x}_t + \boldsymbol{W}^{ss}(\boldsymbol{r}_t \odot \boldsymbol{s}_{t-1}) + \boldsymbol{b}^{s}\right) \tag{2.8c}$$

$$\boldsymbol{s}_t = \tilde{\boldsymbol{s}}_t \odot \boldsymbol{z}_t + (1 - \boldsymbol{z}_t) \odot \boldsymbol{s}_{t-1} \tag{2.8d}$$

The output state $\boldsymbol{s}_t$ is controlled by two vectors: the update gate $\boldsymbol{z}_t$ and the reset gate $\boldsymbol{r}_t$ defined in equations (2.8a) and (2.8b) respectively. Since both vectors come from applying a sigmoid function, most of their components will be close to 0 or 1, controlling the information passed by any element-wise multiplied vector. In extreme cases, when the gate components are 0 or 1, the flow of information can be completely blocked or remain intact. For example, if the reset gate $\boldsymbol{r}_t$ is zero and the update $\boldsymbol{z}_t$ is one, the previous hidden state $\boldsymbol{s}_{t-1}$ does not affect the next state $\boldsymbol{s}_t$.

Once the reset and update gates are computed, a proposed hidden state $\tilde{\boldsymbol{s}}_t$ is created. Equation (2.8c) defines the proposed hidden state $\tilde{\boldsymbol{s}}_t$ at time t. Notice that $\tilde{\boldsymbol{s}}_t$ depends on the input vector and the information from the past hidden state $\boldsymbol{s}_{t-1}$ that is allowed to pass by the reset gate $\boldsymbol{r}_t$. Afterwards the true state $\boldsymbol{s}_t$ is computed as a combination of components from the proposed state $\tilde{\boldsymbol{s}}_t$ and the previous state $\boldsymbol{s}_{t-1}$. Notice that in equation (2.8d) the element-wise multiplications $\tilde{\boldsymbol{s}}_t \odot \boldsymbol{z}_t$ and $(1 - \boldsymbol{z}_t) \odot \boldsymbol{s}_{t-1}$ can control which components of the proposed state and the previous state are kept. The matrices $\boldsymbol{W}^{**}$ and vectors $\boldsymbol{b}^{*}$ are the weights adjusted during learning.

2.3.2 *Sequence-to-sequence models*

Most Neural Network (NN) architectures prior to [73] were designed for problems where the goal is to map input vectors $\boldsymbol{x}$ to fixed size output vectors $\boldsymbol{y}$. Sequence-to-sequence models proposed an end-to-end trainable neural network architecture designed to tackle sequential data. The architecture is based on two RNN models named *encoder* and *decoder*. The encoder is designed to read the input sequence and produce a vector that is passed to the decoder. Since the hidden state of the encoder is updated by reading all the information from the input, it can be interpreted as a summary of the input sequence. The decoder reads the encoded vector and produces a new sequence, but instead of initializing the hidden state of the decoder with zeros, it is initialized to the last hidden state of the encoder RNN. These types of models have been successfully applied to different tasks ranging from machine translation, e.g [73][4], to image captioning [86], both of which share the sequential nature (and challenges) of the monitored traces that we use in Chapter 5 of this dissertation.

2.3.3 *Attention mechanism*

The main drawback of the *sequence-to-sequence* approach explained in Section 2.3.2 is that the whole input sequence is compressed into a single vector [4]. Since the amount of parameters and the computational cost of running RNNs grows with the number of neurons in the hidden state, it is not feasible to use hidden states sizes of the order of $n \cdot d$, where n is the number of elements in the sequence and d the number of features. Storing all the information of a sequence into a single vector of fixed size is therefore a difficult task.

The work done by Bahdanau et al. [4] improves the architecture of the previously mentioned sequence-to-sequence model by furnishing the decoder with an *attention mechanism*. At every step of the decoding process, this mechanism provides a vector containing information about what *might* be relevant from the input sequence. Therefore, the decoding process is not performed from a single vector as in [73], but from a vector generated at every time step. This vector generated by the *attention mechanism* is called the context vector. In [4]

authors showed that the performance of a sequentence-to-sequence model with attention did not degrade for sequences of various lengths, whereas the same model without attention gave far worse results for long sequences.

The decoding process for an RNN with attention works as follows: First, the encoder reads the input sequence and generates the matrix $\boldsymbol{E} = \text{RNN}^*_{\text{enc}}(\boldsymbol{x}_{1:n}; \boldsymbol{\theta})$. Then, at time t, the decoder RNN_{dec} receives as input (in addition to any recurrent inputs) the concatenation of two vectors: $[\tilde{\boldsymbol{y}}_{t-1}; \boldsymbol{c}_t]$. This concatenation provides extra information at the decoder at every time step. The *context* vector presented in [4] is defined by equations (2.9a) to (2.9c).

$$\boldsymbol{\mu}_t[k] = \boldsymbol{v}^T \tanh(\boldsymbol{W}^{\boldsymbol{\mu}} \boldsymbol{E}[\cdot, k] + \boldsymbol{V}^{\boldsymbol{\mu}} \boldsymbol{s}_{t-1} + \boldsymbol{b}^{\boldsymbol{\mu}}) \tag{2.9a}$$

$$\boldsymbol{\alpha}_t = \text{softmax}(\boldsymbol{\mu}_t) \tag{2.9b}$$

$$\boldsymbol{c}_t = \boldsymbol{E}\boldsymbol{\alpha}_t \tag{2.9c}$$

In (2.9c) the *context* vector is computed as a matrix vector product, which can be interpreted as a weighted sum of the columns of $\boldsymbol{E}$. The weights given to the columns of $\boldsymbol{E}$ are the values of the *attention* vector $\boldsymbol{\alpha}_t$. The vector $\boldsymbol{\alpha}_t$ from (2.9b) is called the *attention* vector because it contains the weights used to "focus" (or attend) to different parts of $\boldsymbol{E}$. This vector is computed by normalizing (with a softmax function) the *attention* energy $\boldsymbol{\mu}_t$. The *attention* energy is computed in (2.9a) by using a single MLP, with a single tanh layer. The MLP predicts a single scalar which defines the k'th coordinate from $\boldsymbol{\mu}_t$. The MLP is applied at every position k from 1 to n, where n is the number of columns of $\boldsymbol{E}$, making $\boldsymbol{\mu}_t, \boldsymbol{\alpha}_t \in \mathbb{R}^{n \times 1}$. Notice that the product of $\boldsymbol{W}^{\boldsymbol{\mu}} \boldsymbol{E}[\cdot, k]$ in (2.9a) does not depend on the time step t and therefore it can be precomputed in advance.

One of the key properties of the presented *attention* mechanism is the ability to construct a fixed-size vector $\boldsymbol{c}_t$ of dimension d which does not depend on the length of the input sequence. The vector $\boldsymbol{\mu}_t$ has variable size because, for a given sequence of arbitrary length n, vector $\boldsymbol{\mu}_t$ is computed by applying (2.9a) for $k \in \{1, \ldots, n\}$. Nevertheless, the product $\boldsymbol{E}\boldsymbol{\alpha}_t$ is a vector of size d, where d is the number of rows in $\boldsymbol{E}$ because $\boldsymbol{E} \in \mathbb{R}^{d \times n}$ and therefore $\boldsymbol{c}_t = \boldsymbol{E}\boldsymbol{\alpha}_t \in \mathbb{R}^{d \times 1}$. In many applications, $\boldsymbol{E}$ is generated using a bidirectional RNN which simply concatenates the hidden states of two RNNs. One RNN_{f} receives the input sequence forwards (from x_1 to x_n) while the other RNN_{b} receives the sequence backwards (from x_n to x_1). The first publication that showed the advantage of the attention mechanism [4] uses a matrix $\boldsymbol{E}$ that is created stacking $\text{RNN}^*_{\text{f}}(x_{1:n})$ on top of $\text{RNN}^*_{\text{b}}(x_{n:1})$.

2.4 BAYESIAN OPTIMIZATION

BO [70] is an optimization technique based on constructing a Gaussian Process (GP) (the infinite-dimensional analog of a multidimensional Gaussian) that is used to assess where to sample points in the search space. Let us consider we want to optimize a function

$f : \mathcal{X} \longrightarrow \mathcal{R}$ where $\mathcal{X}$ is the set of hyper-parameters we want to search over. That is, we want to find x^* defined as follows:

$$x^* := \arg\min_{x \in \mathcal{X}} f(x)$$

At a high level, BO creates an approximate function $\hat{f}$, called the surrogate of f, that approximates f. This function is iteratively updated and it is created because $\hat{f}$ is much cheaper to evaluate than f[1]. Once the surrogate is created, an acquisition function $\mu : \mathcal{X} \longrightarrow \mathcal{R}$ is used to generate new sample. This sample is used to update $\hat{f}$ and the process is repeated until an stopping condition is met. Note that this process depends on $\hat{f}$ and μ which we have not yet defined.

Now that we have a high level idea of the optimization process let us define it with detail. The optimization algorithm starts generating a set of random examples $D_n = \{x_1, \dots, x_n\}$. Then, f is used to evaluate the n points in D_n, generating $f(D_n) = \{f(x_1), \dots, f(x_n)\}$. The point with lowest evaluation is set to be x^*, that is $x^* = \arg\min_{x \in D_n} f(x)$. Once the points are evaluated, the pair $(D_n, f(D_n))$ defines a supervised dataset that can be used to fit a regression model that becomes $\hat{f}$. In the case of BO, the model that is fitted is a GP. We will not go into the details of how a GP is fitted (for more details see [70]) but it is sufficient to state that it can be thought as a regression algorithm that can be used to make predictions.

Once the surrogate model $\hat{f}$ is fitted for the first time, then $\mu(x)$ is used to generate x_{n+1}. There are many choices for μ. One of the most common choices for μ is the Expected Improvement which is defined as:

$$\mu(x) = max(0, f(x^*) - f(x))$$

The point x_{n+1} is selected to be the point with the highest $\mu(x)$. After generating x_{n+1}, the dataset D_n is updated as $D_{n+1} := D_n \cup \{x_{n+1}\}$ and $f(D_{n+1}) = f(D_n) \cup \{f(x_{n+1})\}$. Moreover, x^* is also updated as $x^* = \arg\min_{x \in D_{n+1}} f(x)$. Since a new pair $(x_{n+1}, f(x_{n+1}))$ has been added to the dataset, the surrogate $\bar{f}$ is updated to fit $(x_{n+1}, f(x_{n+1}))$. This process is repeated until a termination criteria is found, storing as x^* the point with highest $f(x^*)$ during the process.

1 Note that if f is fast to evaluate then the optimization process is meaningless and there is no need to create an approximation.

3

RELATED WORK

This chapter presents a summary of related work. It provides an overview on the three areas in which this dissertation aims to bring contributions: workload phase characterization, workload interference under co-scheduled scenarios and auto-tuning of Big Data workloads.

3.1 WORKLOAD CHARACTERIZATION AND WORKLOAD MODELLING

In order to feed the heuristics used to manage different aspects in modern Data Centers, it is a common practice to use workload models [22, 49, 74]. This section presents many methods attempting workload phase detection. In this dissertation we understand by phase detection the ability to assign integer values to different parts of the execution of a program. The goal is that the behaviour of applications under the same phase value (also referred to as phase state) is similar in the parts of the execution that share the same phase value. The general approach to found such phases is based on constructing a workload model. A workload model is a mathematical model that finds the phases in the execution of an application. In many cases, the workload model is also capable to predict future phase states based on the current execution state of the running workload.

Many of the presented works employ source code analysis or marking. This involves several drawbacks: such a process is tedious and specific to each source code; moreover, most "cloud" scenarios do not have the source code of the running applications available, since applications are submitted as black-boxes. One of the most crucial differences between the work that we present on Chapter 4 and most of the related works found in this section is that our methodology focuses on approaching the problem taking as input only resource usage logs and sensors, which is a non-invasive approach that can be applied from the cloud provider point of view.

Workload modelling has been widely explored in the literature to produce more efficient resource management methods. Some existing works use simulations to generate models, such as in [49], but these approaches are limited in their applicability in real-world scenarios as they require complex simulations to generate workload patterns. Other works use a

black-box approach based on the generation of workload profiles from previous executions, as in [22], where the authors perform efficient workload collocation to save Data Center energy consumption.

Works such as [67] allow user-provided phases that can be introduced by the programmer. This work presents a tool for workload modelling and reproduction parallel applications in which the user is responsible for building a task graph that is defined by a list of phases. Phases model different behaviours (CPU, IO, LOOP, FORK, JOIN). The main objective of this work is provide a tool for programmers to facilitate the understanding of parallel applications.

The ability to make look-ahead predictions on expected phase changes over time is an important control knob that can be leveraged for more accurate resource management as shown in [62]. Phase detection has been extensively studied, using both supervised and unsupervised techniques to find behavior changes in workloads.

The authors in [21] focus on applications phase detection and exploitation by means of two approaches, top-down and bottom-up, also taking into account off-line and on-line phase detection. In the top-down approach, execution is divided into candidate phases, based on the high-level structure of the source code. The beginnings of long-running subroutines and loops mark the potential boundaries between phases. Such an approach requires compile-time instrumentation to insert marks at candidate phase boundaries. The bottom-up approach starts with the behavior metrics observed during execution and looks for recurring patterns and changes. The beginning of long-running subroutines and loops marks the potential boundaries between phases. This can be done with unmodified program binaries, yet is likely to be strengthened considerably by going back to the source code to correlate observed phase transitions with certain groups of static instructions. However, profile-driven strategies such as [21] require the insertion of markers into the code, which implies being able to access the source code.

In [61], the authors present a method for learning to identify workload phases from live traces using Support Vector Machine (SVM) to classify phases that have been manually tagged from a Dataset of Storage traces. The ultimate goal of the paper is to trigger phase-specific system tuning for disk IO time-series. The main drawback is that data must be manually tagged, so in the scenario presented there the learning process would require supervision from the application owner. The method is evaluated using accuracy across all classes.

Some works, such as [57], make use of models that intrinsically model temporal dependencies to model sequences. In this work a Hidden Markov Model (HMM) provides a phase descriptor from the branch-instruction traces generated during the program executions. The authors pre-process the branch-instruction traces by binning together discrete observations from windows into a vector. The vector for a given window contains the number of times

at each component that a particular symbol appeared. This process maps the windows discrete observations into a single vector. The main drawback is that the original ordering of symbols is lost at granularities smaller than the window size. Since this is unsupervised learning, the data does not contain tags for phases, as the user specifies how many phases are expected to be found as hidden states on the HMM. The evaluation is conducted by measuring how much variance can be accounted for by the language and state probability transition matrices, then computing the accuracy with respect to their "prior-model".

Finally, authors in [54] focus on on-line phase detection algorithms. Their work also uses the source code to identify loops and repeated method invocations to build a baseline solution. It then compares the proposed phase modeling against the baseline solution. In order to identify periods of repetition (and then phases), loops and method invocations are selected from the source code and the entrance and exit of each repetition construct is recorded with a unique identifier. Their unsupervised learning methodology uses the minimum phase length as hyper-parameters, rather than determining the number of expected phases to be found. They also require the source code for such analyses.

3.2 WORKLOAD INTERFERENCE UNDER CO-SCHEDULED SCENARIOS

This section presents related work that is relevant for resource, runtime and interference prediction. These topics share a similar goal to our work presented in Chapter 5 and are closely related to each other. Moreover, they are still an active field of research with many different applications.Two relevant applications are dynamic provisioning and job placement. In dynamic provisioning scenarios the main goal is to predict whether there will be a change in the workload trace that needs some hardware decision to take place (like provisioning more nodes for a workload). In the case of job placement, resource predictions also play an important role, since similar resource needs for different jobs resources may affect. Thus, schedulers may aim for colocations of applications that do not degrade when trying to maximize the use of available hardware.

Resource prediction related work

The main goal of oracles in workload provisioning scenarios is to guess accurately when workloads will need more resources in the future. Works like [34, 39] use neural networks that take as input resource usage in a given time window with the goal of predicting the future resource requirements for the workloads. In [39] authors use differential evolution as a means to train the models, whereas in [34] they use standard gradient based algorithms.

Recurrent neural networks have also been applied successfully in the context of workload prediction behavior [56, 94]. In [94] RNNs are used to predict cloud resource requests

of Google cloud CPU and RAM requests, results are compared against Autoregressive Integrated Moving Average (ARIMA) forecasting, achieving lower error with the RNN. In [56] an autoencoder is trained to learn a representation that is fed to a RNN that predicts the number of requests (and the CPU time) of different workloads. These works use a RNN for forecasting but they do not take into account the degradation of the applications under co-scheduled scenarios or the challenges that appear when a full time-series is meant to be predicted from more than a input trace signal.

Works like Sanz-Marco et al. [50] apply machine learning to forecast memory requirements towards interference avoidance. They propose a mixture of expert approach to predict the amount of memory needed by a Spark application, given the size of the input data. First they create a dataset consisting of applications and Spark profiling features, captured from hardware counters (CPI, number of interruptions, number of cache misses, etc). Then, after using Principal Component Analysis (PCA) to reduce dimensionality, they use a mixture of experts to learn a regression function that takes as input the dataset size of the application. This function predicts the memory footprint of an application depending on the input dataset. To build such a function, the application is first profiled with a small input 10%, and it then passes through a K-Nearest Neighbor (KNN) in order to choose the most representative expert. Finally, the application is run with different data sizes in order to tune the function parameters to determine the best fit for this application. The main downside of this work is that the application memory profile that is provided is a single point estimate for a given input dataset. Therefore, applications that have distinct memory phases over time, are represented as a single point that can provide a high over-provisioning of memory for the full execution of the application.

Interference focus-related work

In [19] the authors present Paragon, an approach to predicting the interference between two applications by modeling them using Singular Value Decomposition (SVD) and PQ-reconstruction. In Paragon, the profiling across pairs is limited to the first minute due to time constraints, without covering applications with different execution phases and behaviors over time, although they acknowledge the problem by trying to mitigate it by labeling nodes with unexpected changes in resource demands, then leave them running until jobs are finished.

Heracles et al. [46] present an approach for interference mitigation in high priority applications through job isolation techniques. Among other tools, their solution uses *cpugroups*, *qDisc* or the *CAT* technology [33] to properly ensure resource availability for specific jobs. Such a solution tends to over- resource applications for solving the interference problem,

thus becoming subject to machine under-utilization. Furthermore, the optimization method poses an NP-hard solution, which makes it feasible for only a small number of jobs.

In a similar line, Mishra et al. [53] presented ESP, a predictive model for forecasting the interference between two applications. ESP uses low level metrics as input for the model, but instead of making predictions at sequence level they are held at a scalar level, both at the input and the output, by means of aggregation metrics such as the average of the sequence. Thus, given the average of two metrics, it predicts the average of the concurrent execution in a two-step model as well; one for feature selection and the other for using a regression model to retrieve the expected aggregated metric as output when both tasks are co-located.

Another interesting work dealing with protection against interference is Stay-Away by [65]. The idea they propose starts by projecting the resource consumption of the applications in a 2-dimensional space. They then predict how applications will move in that 2D space. If applications are predicted to have collisions between applications in that space, they then adopt a counteraction.

Run-time prediction-related work

In [1] authors show how performance models can be build efficiently sampling the space of possible query mixes. This work uses specific features that capture the completion time of a Query when running in a particular co-scheduled scenario. This is different than the percentage completion features that we propose in our work, which are vectors that are passed as input to the model with the goal of generating output percentage completion vectors that depend on the input workloads.

In [88] a mixture of runtime prediction and interference is presented in two methods: The first, aimed at calculating the cost of a Spark job run in isolation, is approximated by modeling a function from the execution parameters (the number of data partitions, the number of stages, the number of jobs per stage, etc.) to predict the total cost of the job. The second, designed to predict the cost when two jobs run in concurrency by modeling interference, is tackled by adapting the previous formulas. They run a small part of the input data for each Spark stage combination to compute a ration of interference which is used to create a performance model This work recognizes the different behavior across the execution caused by the different phases of executions, but is limited to predefined Spark phases that have to be provided by the user.

In [89] the authors present a Convolutional Neural Network (CNN) named PRIONN. The model predicts run-time and IO (bytes read and total bytes written) of applications based on the source code in the input script. This work maps job scripts code into image-like representations. This is done using different methods. For example they use a one-hot character transformation that converts each unique character to a unique 128 value vector.

Then they concatenate all vectors into a 2D array that creates "an image". Then, CNN models are trained from the image-like representation with the goal of predicting runtime and resource usage.

3.3 WORKLOAD AUTO-TUNING OF BIG DATA WORKLOADS

Auto-tuning Big Data workloads is a popular research field with a wide variety of solutions that target all major Big Data technologies [3, 6, 10, 42, 44]. In particular, tuning Hadoop and Spark workloads are an important concern in the Big Data community that has worked on different approaches to improve current solutions. This section presents a summary of related work, which can be split into two categories: model based solutions and model free solutions. Model based solutions use a statistical model that is trained on examples of the workload to be optimized. Then, the model is used to provide the expected results of the workload on an evaluation metric that the system aims to optimize, avoiding the execution of the application. Model free solutions group all the works that avoid training a model to avoid running the workload.

Auto-tuning techniques that leverage Performance Models

Let us consider S the space of hyper-parameters to be optimized and $s \in S$ a configuration of such space. Let ds be the dataset size of the input application (in case there is any). Let app_{id} be an integer number that is used as an application identifier. Most works that optimize Big Data configurations are based on performance models that take as input a subset of features from $\{s, ds, app_{id}\}$.

Works like [5] use s as input to train different learning algorithms for each application that the systems aim to tune. In this work, special focus is given constructing the dataset using a Latin Hypercube Sampling algorithm.

Works such as [92] or [55], use (s, ds) as input to the learning algorithm. The proposed solution in [55] learns a model for each workload. Then it uses Recursive Random Search [91] to find a good workload configuration. In [92] a Genetic Algorithm is used to perform the search. The genetic search is based on the estimates of the execution time predicted by a performance model. A geometric interpolation method mixed with a sampling strategy to build a faster and accurate model by a small number of historical job executions was introduced by [14]. Finally, a big training dataset created by exhaustive search and random exploration was the focus of [87], resulting in a 1500 training points for every four workloads explored with several common machine learning models. Note that the previous solutions require a model for each workload to be optimized, which is one of the problems that we focus to improve in our third contribution in Chapter 6.

ALOJA-ML [9] uses (s, ds, app_{id}) as input to the learning algorithms. This work uses a single performance model that is shared across applications. The main downside is that all of the application-related information is captured in a single categorical variable that is provided to the model. Therefore, unseen applications do not provide any information about the nature of the workload and retraining is required in order to search good parameters for new workloads.

The work from [37] focus on an orthogonal approach for improving the efficiency of model-based solutions for auto-tuning highly configurable software. This work focuses on building a simulator for the process from which they want to learn. Therefore, this work essentially builds a model that generates samples so as to avoid running workloads and, thereby, reduces the overall cost of the auto-tuning system. In our setting, this work could be applied to increase the generation of a dataset of (configuration, execution time) pairs. This could be achieved using a simulation mechanism, instead of actually running applications.

In [40], a vector of statistics is extracted from the runtime of the application to be optimized. This feature vector $\boldsymbol{q}$ is used to modify a BO procedure to guide the search process. This work presents a Guided Bayesian Optimization process (GBO) that uses an slight modification of the Expected Improvement function in the BO optimizer which receives as input the monitored metrics $\boldsymbol{q}$. Our work shares the core idea of providing specific knowledge about the running application to improve the search process. The main difference is that, in [40], the authors do not try to proxy application executions with metrics provided with an oracle.

To the best of our knowledge, [27] is the only publication in the area of auto-tuning that proposes a system that generalizes to unseen workloads by making use of features that capture low level features such as Task and Stage information from an application. This work builds a single model that is trained by taking as input a descriptor from each of the stages of a Spark application. After learning, once an application is selected to be optimized, the model is queried for each of the stages in the application and for each of the configurations that the system is expected to tune. Then the final predicted execution time is the sum for the predicted times of the stages. Once all predictions have been made for all the allowed configurations, the minimum sum is selected as the best one. Note that the predictions are made for all the allowed configurations, making this approach impractical for auto-tuning a big space of configurations, which is something that our third contribution aims to do. In [27], only two parameters are tuned and the overall search space contains only 12 possible combinations of the two parameters.

The idea of using information from logs, that we will cover in Chapter 6, is not completely new in some related areas. For example, in database management system (DBMS), tools like OtterTune [2] use Gaussian Process (GP) to recommend suitable parameters (or "knobs", as more known in DBMS) for different workloads by extracting the internal state of the

database to reflect the workload characteristics. This work uses statics collected from the amount of data written or read as well as the time spent waiting for resources.

Model free based auto-tuning

In [81], expert knowledge is used to select a relevant set of hyper-parameters to be studied for each workload. The performance of the different applications is evaluated under a discretized set of values for the selected parameters. This approach requires experts to determine which hyper-parameters have to be tuned, making the solution impractical for a general-purpose system that aims to tune different types of applications with different needs.

A straightforward approach to improve the previous solution relies on optimizing via trial-and-error [60]. In this work, authors tune 12 parameters from the configuration space for three applications using a simple but effective heuristic. For each application, a parameter is selected and several values of the parameter are tested. If there is a large gain with respect to the default configuration, the parameter is considered as significant to the overall performance. With this approach, a graph is generated with the most important parameters as nodes. Once an application is presented it is supposed to be run for all the nodes in the graph and the parameters that maximize performance are selected as the best parameters.

4

WORKLOAD PHASE CHARACTERIZATION THROUGH MACHINE LEARNING

This chapter describes the first contribution of this thesis. This contribution aims to assign cluster labels to each time step from a workload trace. The key observation that motivates this work is that similar recurring resource patterns (or phases) happen in many workloads. Such information could be used for a variety of decision making processes in a Data Center environment since similar workloads might benefit from similar hardware resources.

Section 4.1 presents an overview of the problem. Section 4.2 presents our proposed solution to find phases in workload traces. Section 4.3 presents the experiments that evaluate the phase descriptor quality of our proposed pipeline. Section 4.4 shows how the phase descriptors can be leveraged by other learning algorithms with the goal of auto-scaling applications running in a containerized environment. Finally, Section 4.5 summarises the conclusions of the chapter.

4.1 INTRODUCTION

The complexity of modern Data Centers, which are built from a large number of specialized technologies, poses a huge challenge: to develop technologies that allow for a holistic management of both workloads and the infrastructure while observing differentiated performance goals. As computational resource sharing becomes critical, environment set-up and schedule must be tailored for each application. Unfortunately, applications are often provided as black-boxes, and modeling must be done through sampling executions in sandboxes [95][78].

Existing literature in the area has studied the behavior of applications by attempting to understand common patterns across workloads, working on the assumption that different but recurrent behaviors occur during the course of the execution, which are known as phases [74][79][84] . Such phases display similar usage of computational resources over time. Recognizing which phases compose an application, and identifying the resource usages for each one, allows us to adapt the environment for a better performance as well as

predict what applications can be co-located without interfering in their usage of resources. In this way, applications can be scheduled by means of decomposing them into phases instead of looking at their complete runtime.

Some related works propose invasive techniques that place *phase markers* in applications source code [57][54]. Our objective is to produce a solution that can work on black-box environments in which the application can only be monitored through resource consumption patterns. In such scenarios workload activity is usually collected in the form of traces, which are usually logs for CPU, memory, disk or network usage, among others. Therefore, workload data can be naturally represented as a multi-dimensional time-series.

The contribution presented in this chapter aims to automatically find application behavior phases, using resource consumption traces as input to our algorithm. Our method combines CRBMs [77][76] and a clustering algorithm to distinguish changes on the resource consumption patterns over time. The two models are used for two different goals. The CRBM is used to model the time-series and generate a feature vector capturing the local behaviour of the workload at every time step. Then a clustering algorithm, such as a HMM [64], assigns a label to the vector, automatically detecting and tagging different resource consumption patterns. Using this approach, workload traces can be mapped to a series of abstract phases that give a high level description of the resource consumption characteristics.

CONTRIBUTIONS OF THE CHAPTER

The main contributions presented in this chapter can be summarized as follows:

- A combination of CRBM with a clustering algorithm to encode sliding windows of time-series as phase values. Our method provides a phase detection mechanism that is robust in front of local burstiness in the time-series.
- An experimental evaluation of the phase detection method. We propose an auto-scaling policy based detecting phase transitions. Our method provides better resource allocation and a lower number of auto-scaling changes when compared with other popular reactive methods currently used in the literature.

4.2 METHODOLOGY

Defining phases for time-series is not a trivial task. Workload traces contain complex non-linear relationships between the different components of CPU, RAM, Memory and Disk, so defining phases of similar behavior over time becomes a very challenging task. In order to facilitate this, we learn a representation that maps slices of those multidimensional sequences into vectors. This section describes how this is done using a CRBM. Once the CRBM is trained

we use an unsupervised technique such a HMM or a K-means to learn cluster labels on top of the learned features provided by the CRBM, with the goal of finding meaningful phases. Finally we compare the predicted phases with the meta-information we have obtained from workload indicators to verify the results. In general, evaluating cluster labels is hard. For this problem it is specially challenging because the data comes from a nonintuitive source multidimensional workload traces.

In scenarios like the one proposed, where no true labels exist, or existing labels are either approximate, inaccurate or too generalized, evaluating phase detection models is not trivial. For example, Hadoop executions have labeled "stages" indicating the predominant type of task being executed at each moment ("map", "reduce", "shuffle", ...). In this example, throughout their execution Hadoop workloads present different behaviors along their execution that change depending on the stage and on the application itself. In other words, two different workloads will present different behaviors for the same stage, but two similar ones will behave similarly.

Nevertheless, we can evaluate the quality of the phase prediction on workloads by computing the accuracy between predicted phases against labels on meta-data execution. Such metrics will not be indicative of the phases to be discovered, since this learning method is unsupervised for discovering unlabeled behaviors. However, they will indicate how plausible it is to use the produced phases to gauge the little information the application is providing about their execution stage. We would like to recall that the goal of this work is to learn phases in situations where labels may not be available. So, in this case the Hadoop meta-data is used only for side-validation but never as a target feature for supervised learning.

4.2.1 *Learnable Vector representations*

Let us consider a set of M sequences $\boldsymbol{X}$. In our context each sequence $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l) \in \boldsymbol{X}$ contains measurements from the execution of an application. The length of $\boldsymbol{x}$, is equivalent to the execution time (in seconds) of the workload. Each component $\boldsymbol{x}_t$ is a vector in $\mathbb{R}^{n_v}$, where n_v is the number of features (or measurements) used to describe the sequence at each time step. Notice that sequences are not required to have the same length.

Instead of using directly the sequences from $\boldsymbol{X}$, or manually defining features that aggregate resource consumption over time, we propose to learn a vector representation for our sequence components. A vector representation is a function $\phi : \mathbb{R}^{n_v} \longrightarrow \mathbb{R}^{n_h}$ that maps the original measurements of $\boldsymbol{x}$ at time t, $\boldsymbol{x}_t$, to a vector of length n_h. Given a sequence $\boldsymbol{x} \in \boldsymbol{X}$ we will map $\boldsymbol{x}_t \in \mathbb{R}^{n_v}$ to $\phi(\boldsymbol{x}_t; \boldsymbol{\theta}) \in \mathbb{R}^{n_h}$. The parameters $\boldsymbol{\theta}$ of the representation will be learned from the data, with the goal of maximizing the probability of the sequences in $\boldsymbol{X}$. The n_h value is a hyper-parameter of the representation.

Since our data is composed of sequences we would like the mapping $\phi(\boldsymbol{x}_t;\boldsymbol{\theta})$ to depend on $\boldsymbol{\theta}$ but also on the previous n components of the sequence. This means $\phi(\boldsymbol{x}_t;\boldsymbol{\theta}) = \phi(\boldsymbol{x}_t;\boldsymbol{x}_{t-1},\boldsymbol{x}_{t-2},\ldots,\boldsymbol{x}_{t-n},\boldsymbol{\theta})$. Notice that for the first n values we cannot use this mapping since there are not enough measurements for ϕ. One way to fix this problem would be to set the first history values to zero. Another option is to simply not to use ϕ for the first n time steps. In this Chapter we explore the use of a CRBM as a good candidate for $\phi(\boldsymbol{x}_t;\boldsymbol{x}_{t-1},\boldsymbol{x}_{t-2},\ldots,\boldsymbol{x}_{t-n},\boldsymbol{\theta})$.

4.2.2 *Data Pipeline and Architecture*

Let us assume we have trained a CRBM and therefore we have learned the parameters $\{\boldsymbol{W},\boldsymbol{D},\boldsymbol{b}\}$ of the model. We can compute the vector representation $\phi(\boldsymbol{x}_t;\boldsymbol{x}_{t-1},\boldsymbol{x}_{t-2},\ldots,\boldsymbol{x}_{t-n_{\text{his}}},\boldsymbol{\theta})$, or simply $\phi(\boldsymbol{x}_t;\boldsymbol{\theta})$, at every time step t using (4.1).

$$\phi(\boldsymbol{x}_t;\boldsymbol{\theta}) = \sigma(\boldsymbol{W}\cdot\boldsymbol{x}_t + \boldsymbol{D}\cdot\mathbf{His}_t^n + \boldsymbol{b}) \tag{4.1}$$

Then, we can discretize the result to get a binary code, using 0.5 as the cut point. Usually most output codes do not loose many information since the outputs of a sigmoid tend to be numbers close to 0 or 1. The binary code is used as input to an HMM. In this chapter we investigate HMMs and K-means to generate a cluster label from the embeddings generated by the CRBM. We will focus on the use of a HMM because it captures dependencies across time and therefore is suitable for our sequential data [11, 71]. Nevertheless, we can use any clustering method that maps the hidden states of the CRBM to cluster ids to generate the phase values.

To define a HMM we need a number of hidden states must be specified. This is a parameter that must be tuned by the user, similar to the k in K-means. Given a number of hidden states, which correspond to the number of distinct phases that will be found, the parameters of the HMM are found using the Baum-Welch algorithm [69]. Once the parameters are learned, the most likely state sequence for a given observation sequence can be efficiently computed using the Viterbi algorithm [24].

After having trained both the CRBM and the clustering method, the pipeline for a given sequence $\boldsymbol{x}^m$ of length l_m is composed of the following steps:

- Step one: the representation $(\phi(\boldsymbol{x}_n^m;\boldsymbol{\theta}),\ldots,\phi(\boldsymbol{x}_{l_m}^m;\boldsymbol{\theta}))$ for the sequence $\boldsymbol{x}^m$ is computed using (4.1).
- Step two: the clustering algorithm is applied to the previous sequence to get the sequence $(y_n,\ldots,y_{l_m})$.

This approach does not give phase assignment to the first n components of the sequence, which we will consider as the "initial phase". In the case a user wants to use a HMM to

generate the phase values, the Viterbi algorithm can be applied to get the most likely state sequence $(y_n, \ldots, y_{l_m})$. Therefore, in the case of an HMM, the visible states of the model are the $(\phi(\boldsymbol{x}_n^m; \boldsymbol{\theta}), \ldots, \phi(\boldsymbol{x}_{l_m}^m; \boldsymbol{\theta}))$ and the hidden states of the model generate $(y_n, \ldots, y_{l_m})$.

Figure 4.1 shows a diagram of the data pipeline. The input data and the history data are fed to the CRBM (at every time step). Then the CRBM gives a code a method that outputs a phase value.

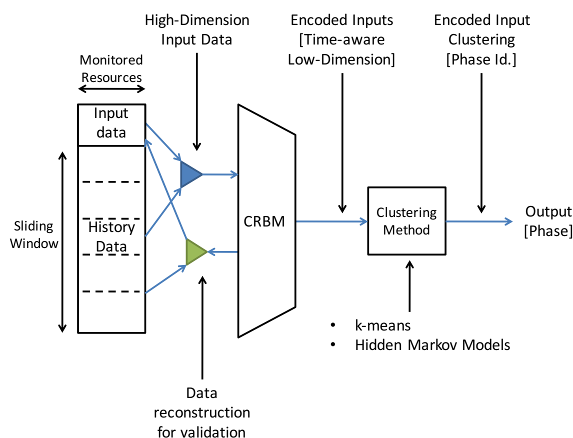


Figure 4.1: Data pipeline schema showing how the resource monitoring data passes through the CRBM and the clustering method.

4.3 EXPERIMENTS: EVALUATING PHASE DESCRIPTOR QUALITY

This section presents an evaluation of the phase descriptors found in previously defined in 4.2.2. The evaluation is done using three datasets (Dataset A, Dataset B, Dataset C) which are defined in the following subsection. The experiments in this section provide some insight on the phase descriptors. Nevertheless, evaluating clustering methods is a hard task. Some works even claim that, in the context of time-series, it is meaningless [45].

Datasets

The workloads used in the following experiments belong to the ALOJA Project, a repository of Big Data executions focused on benchmarking different infrastructure as well as software components. For each registered execution, the dataset contains the monitored usage of CPU, memory, network and disks. Moreover, other execution details, such as markers for Hadoop, Spark and Hive are present. The granularity of the dataset is around two records per second during the execution.

Table 4.1 shows a slice of 3 time steps from a workload extracted from the data. Data is aggregated per second, averaging the data when numeric. Column "instant" is used to identify the time in the series, but it is not used as an input for the machine learning pipeline. The selected features for this approach are "pc.user", "kbmemused", "rxpck.s" and "tps", corresponding to user process CPU usage (in % usage), Memory usage (in kilobytes), Network usage (received packages per second), and Disk usage (transactions per second).

instant	pc.user	kbmemused	rxpck.s	tps
9	11.370	18,730,504	333	−1
10	3.110	18,782,464	276	−1
11	0.930	18,791,856	332	−1

Table 4.1: Example of data slice from the ALOJA dataset. For non-available values, −1 is used instead.

It is true that other features can be added such as *transmitted* packages per second, or system process CPU usage, as well as maximum and minimum values for each feature, in addition to these. However, for this first proof of concept we decided to keep the input simply by selecting the most representative measurements from the workloads. The use of an extended feature version of this approach is intended for future work.

To simplify the feature naming, we will refer to the features as CPU, Memory, Net and Disk. As the workload is distributed among machines and processors, the CPU % usage is a sum over all used cores, and therefore can take values above 100.

Dataset A: Hadoop Workloads using BigBench

The first dataset, extracted from ALOJA Hadoop Time-Series Dataset v1, contains 182 series from Hadoop executions (up to 22 different features at this time), from the Intel HiBench [32] benchmark suite. These workloads contain Map-Reduce algorithms for sorting (*Sort* and *Terasort*), word counting (*wordcount*), machine learning (*K-means* and *bayes*), input-output stress tests (*dfsioe-read*, *dfsioe-write*), etc. All the jobs have been running in on-premise infrastructures, with similar Hadoop configurations. Data generation jobs, usually accompanying workloads, have been excluded from the experiments.

Dataset B: Spark Workloads using TCPx-BB

The second dataset, extracted from ALOJA Spark Time-Series Dataset v1, comprises 900 executions of 30 different Spark applications contained in the TPCx-BB (BigBench [25]) benchmark. TPCx-BB contains 30 frequently performed analytical jobs in the context of retailers with physical and online store presence. They represent different types of workloads (including Natural Language Processing, SQL queries, Mapreduce jobs and Machine

Learning workloads), comprise different data types (Structured, Semi-Structured and Unstructured data), provide a mix of long and short running jobs and can run at different data scales (in our case, 1, 10 and 100GB). For each of the queries, we included 30 instances, comprising the different data scales mentioned before. All the jobs were run in the Microsoft's Azure cloud using Spark 2 as the engine. We used HDInsight PaaS to spawn the spark clusters, running a 16-slave node cluster (plus several redundant head nodes). Data was stored in the Azure Data Lake Store of Azure.

Dataset C: Human motion dataset

For sanity check purposes, in the final experiment presented in this section we leveraged the well-established *Motion* dataset from Hsu [31], also used in Taylor's CRBMs validation [76], to validate our method against a well-known dataset.

Experiments

Here we introduce six experiments to validate the presented phase generation method. The following experiments describe how the proposed methodology differentiates phases throughout workload executions on different workload types. Notice that, for the following figures found in this section, two kinds of plots are produced: detected phases and resources usage. For the detected phase plots *barplots* are used, where each phase is differentiated by color and height. The height is simply present to visually facilitate differentiating the phase values over time. We will refer as *phases* the tags given by the K-means algorithm. We will refer as *regimes* the tags given by the HMM.

- Experiments 1-4 use dataset A (Hadoop workloads).
- Experiment 5 uses dataset B (Spark workloads).
- Experiment 6 uses dataset C (human motion identification) as a sanity check of the proposed method based on classical literature in the field.

Experiment settings

The CRBM model for experiments 1-5 has been trained using a randomly selected 66% of the series. The model has been evaluating using the remaining 33% of the data.

The CRBM selected in the experiments has a history length of 50 samples and 100 hidden units. Training has been performed using a SGD algorithm with learning rate set to 0.001 momentum of 0.4. We have used CD-k, with $k = 1$, to find the gradients needed to update the parameters of the model. The learning rate was set manually testing different values with different orders of magnitude: $0.1, 0.01, 0.001, 0.0001$. Values higher than 0.001 produced big spikes in the reconstruction error monitored during learning. We have found that the

reconstruction error plateaus during the first 40 epochs and further training does not help. Moreover, the reconstruction error achieved by the model using 100 hidden units is not significantly improved by models with more hidden units (200, 300) and the same history size, therefore we chose a model with 100 hidden units to perform the experiments shown in this section.

Moreover, for the following experiments, several values of K (for the K-means) and expected number of hidden states in the HMMs have been tested. The most distinctive value found for this hyper-parameter is 5 clusters, since for lower values of K the algorithm displayed randomly-joined phases, while for higher values it converged by returning empty or underpopulated clusters. This led us to choose $K = 5$ as the fittest value for the current kind of workloads. Notice that for other kinds, this hyper-parameter must be tuned.

We used Python 3 to implement the experiments shown in this chapter. The scikit-learn library [59] was used to train a K-means and a HMM. We wrote a custom CRBM implementation with the sciki-learn API to facilitate building the whole pipeline that transforms input sequences to the output labels.

Table 4.2 shows the time needed to train (in minutes) until the loss plateaus. The train time could be reduced by computing all matrix operations using GPUs, which our implementation did not use because it was already fast enough for our purposes. The training times are reasonable in a production environment and training could be performed in a batch process, when new, very distinct workloads enter the system.

n_h	3	10	50	100	200	300
minutes	11.3	13.0	23.3	36.3	99.9	136.7

Table 4.2: CRBM training time, n_v is the number of hidden units. All models have the same delay, 50 time steps.

Note about the portability of the model across workloads

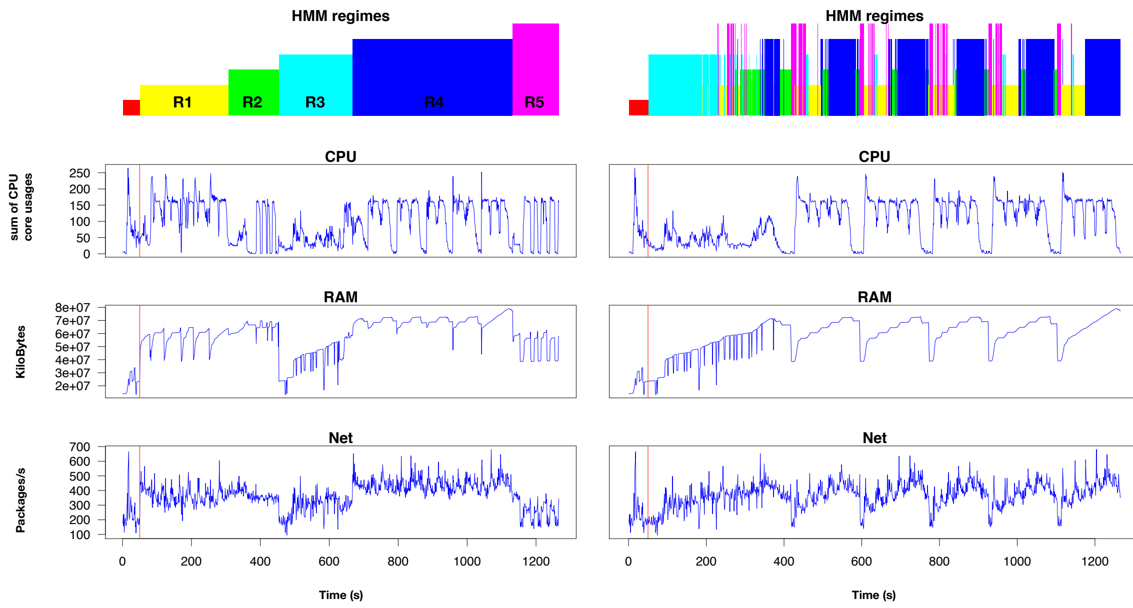
A model trained on a specific type of workload might not be suitable for use on another. This could be because the data may be quite different in shape as well as in feature ranges. For example, our experiments used CPU feature values in the range $(0, 100 * n_{cores})$, where n_{cores} is the number of cores. If all the training data contains workloads that use single core of the machine, then all the phases will be take into account CPU values in that range. Therefore, if a new trace appears taking values outside the trained range, the model may give unexpected phase results. It is important to determine the range of the features of the production/test data at which we aim to apply the method. It is recommended to train the CRBM with a dataset containing a variety of applications that cover the different feature ranges from the input metrics.

Experiment 1: Unsupervised automatic Phase Detection

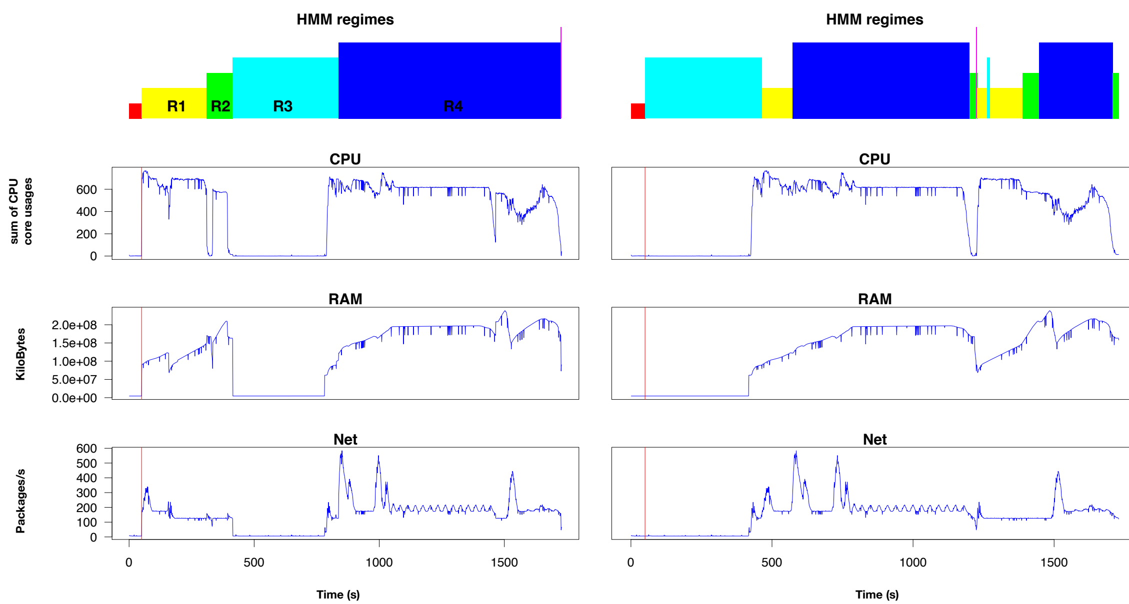
In order to understand the different behaviors found in the predicted clusters given by the K-means and the HMMs, here we show some workloads with the associated tag sequences (the discovered phases). Although we have generated the study for all the workloads available in the data-set, we display here the most representative ones. The history period of the CRBM is marked in Figures 4.2, 4.4, and 4.5 by a vertical red line in the workload trace that marks the time $n = 50$.

Figure 4.2 (next page) shows a couple of workload traces with the predicted phases $R1, \ldots, R5$ given by the HMM. The right hand side images from sub-figure 4.2a and sub-figure 4.2b contain the workload resource usage and the predicted phases in chronological order. The left hand side images contain the same information of CPU, Memory and Net traces, but grouped by the phase tag in order to see how each resource behaves in each given phase.

The aim of grouping the time-series elements by phase is to display the general trend of consumption for each resource, defining the phase. We have the supported hypothesis that each discovered phase will be characterized by a trend in one or more resources distinguishable from the other phases. The fact that usage in some resources does not need to be constant is covered by the encoding done through the CRBM. The left hand side images provided in sub-figure 4.2 are precisely created to visually aid distinction among different behaviors in the time-series.



(a) Workload A



(b) Workload B

Figure 4.2: Input trace behaviour for each phase value: $R1, \ldots, R5$

Figure 4.3 contains the histograms of the different traces across all data, grouped by $R1, \ldots, R5$.

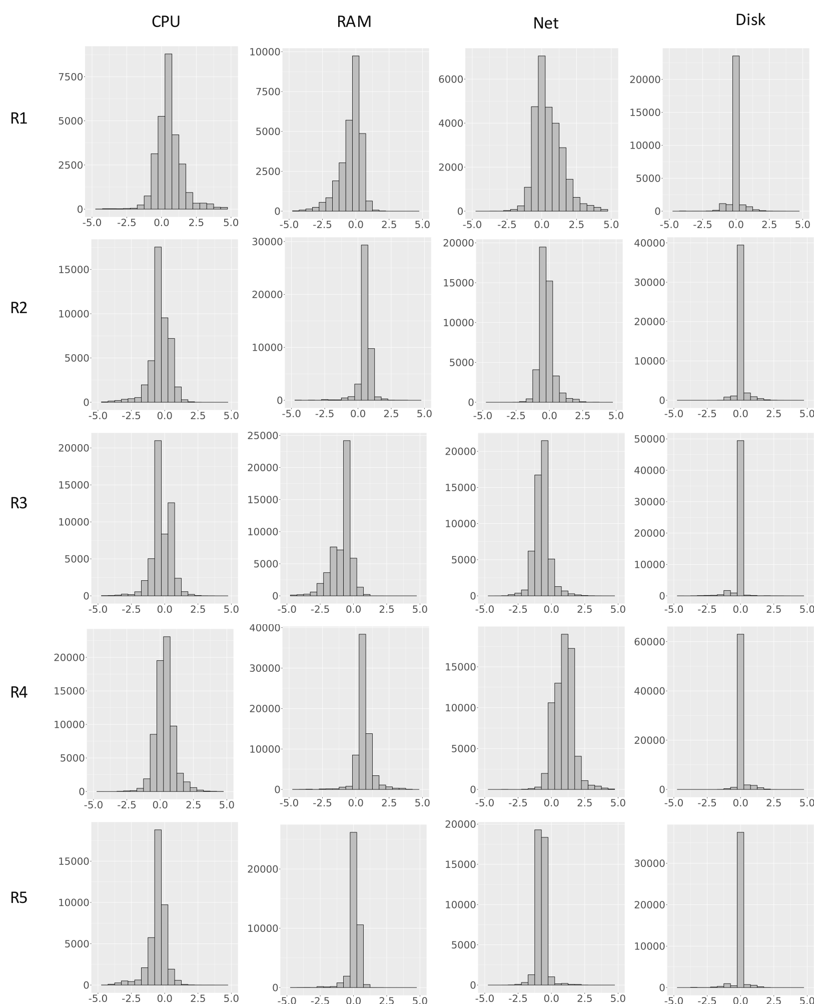


Figure 4.3: Histograms of the normalized input features grouped by $R1, \ldots, R5$.

Table 4.3 contains the mean and standard deviation of the different trace components grouped by $R1, \ldots, R5$.

Regime	CPU.mean	Mem.mean	Net.mean	Disk.mean	CPU.std	Mem.std	Net.std	Disk.std
$R1$	**0.724**	−0.379	0.519	−0.015	**1.515**	**0.866**	**1.062**	**0.399**
$R2$	−0.348	0.539	−0.222	0.032	0.766	0.554	0.526	0.303
$R3$	−0.186	−0.857	−0.710	−0.060	0.732	0.834	0.588	0.342
$R4$	0.341	**0.585**	**0.946**	**0.047**	0.698	0.726	0.753	0.270
$R5$	−0.527	0.045	−0.731	−0.020	0.677	0.757	0.349	0.320

Table 4.3: Mean and standard deviation of the normalized traces under the different *regimes* given by the HMM.

The following brief description is a simplified textual description of the behaviors found in Table 4.3.

- $R1$ contains trace behavior with high CPU usage and high variance across all other traces. This pattern may be observed on the left-hand side workload in Figure 4.4, which shows the model detecting phase $R1$ around time step 1300, where there is a peak of CPU usage. Table 4.3 shows that $R1$ has the highest mean CPU usage.
- $R2$ is similar to $R1$, but the Memory usage under R2 is higher and the CPU usage is slower. Table 4.3 shows that $R2$ contains the second highest mean Memory usage.
- $R3$ detects regions with low Memory usage with low CPU usage. Table 4.3 shows that $R3$ contains the lowest mean Memory usage and the second lowest Net usage.
- $R4$ contains high Memory, high Net usage. Table 4.3 shows $R4$ as containing the highest mean Memory, Net and Disk usage.
- $R5$ contains similar behavior to $R2$ but with lower resource usage than $R2$.

Experiment 2: Phase detection from workload traces

It is important to notice that Hadoop stages do not determine the behavior of the CPU, Memory and Net traces. Figure 4.4 shows two workloads with different Map, Reduce and Shuffle stages, containing similar behaviors in the traces for different stages. The vertical boxes in the figure show a slice of "$R4$" behavior with high Memory and above average Net usage taking place in two different Hadoop stages (*map* for workload 1 on the left, and *reduce* for workload 2 on the right).

The presented methodology is not intended to detect Hadoop stages as "phases", but for the same kinds of workloads it detects the same phases for the same stages, while for different workloads, for the same kind of behavior it detects the Hadoop stages that behave similarly to one another. This allows us to characterize applications according to sequences of phases during the execution. As the methodology presented herein never sees the Hadoop stages, it relies on the provided resource traces, which makes it extensible to any other application and framework.

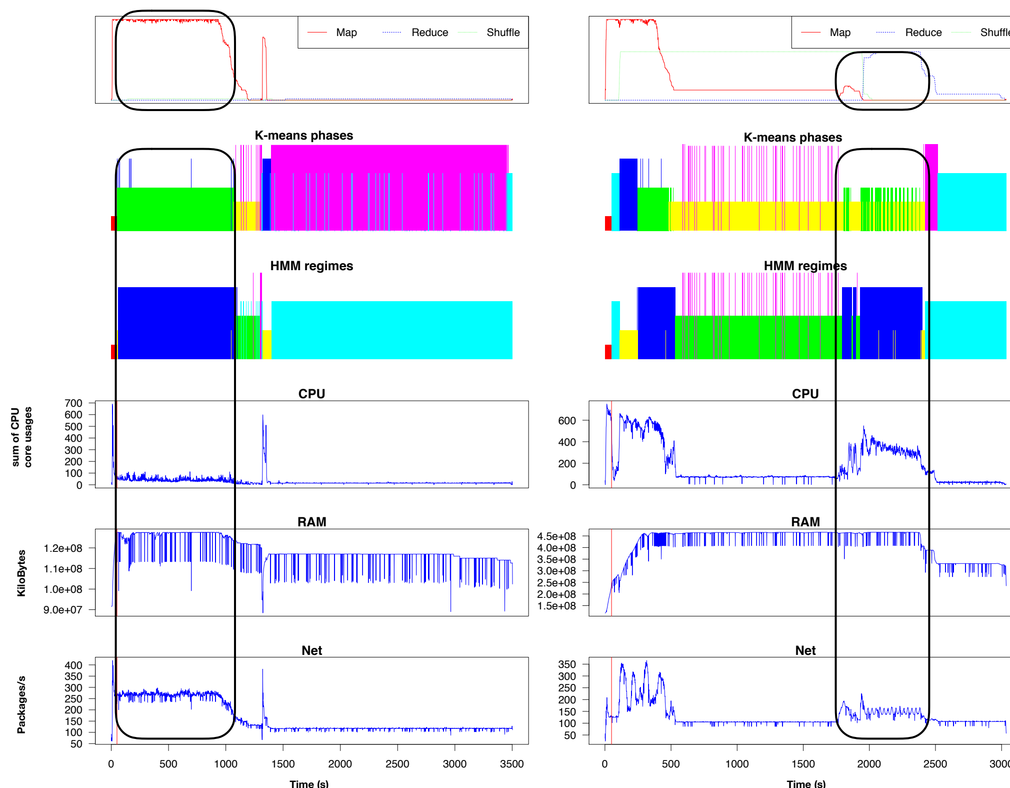


Figure 4.4: Two different workloads side by side. This example shows our phase descriptors do not exactly match the Hadoop phases.

Experiment 3: Accuracy Analysis using Hadoop logs: mapping detecting phases to MapReduce phases

The Map-Reduce data used contains several tags at each time-step. Tags have been used only for evaluation purposes (not for training the algorithms). We have previously remarked that Hadoop phases do not determine the resource consumption, as can be seen in Figure 4.4. Nevertheless, we can make an approximate validation of our model by comparing the predicted phases with the Hadoop phases. Moreover, we can compare the phases given by the K-means and the HMM in the learned representation.

The most representative phases of this type of workloads are the map phase, the reduce phase and the shuffle phase. We have codified the tags as integer values, which we will refer to as the true tags. The codification of the true tags has been performed as follows. Let us consider binary valued vectors (m, s, r) where each index taking value 1 represents that the data is in a particular state. The use of this form $(1, 0, 0)$ represents that the data is in a map state, $(0, 1, 0)$ in a shuffle state and $(0, 0, 1)$ in a reduce state. Any other combination represents data in a combination of states; for example, $(1, 1, 0)$ would represent the data being in a map and shuffle phase. Each possible binary vector has been assigned to an integer, the equivalent number in binary form. For example, $(0, 0, 1) = 1$ and $(1, 0, 1) = 5$.

Experiment 4: Finding a correspondence between true phases and predicted phases

To assess numerically the quality of our phases, we find for each value of K (number of clusters) the correspondence that most closely matches the predicted phases and the true phases. That is, we find a matching function f^* that maximizes the accuracy of the predicted phases and the true phases across all our data. Let $\boldsymbol{Y}$ be the set of sequences containing the correct phases. Let l_y indicate the length of a sequence of phase tags $\boldsymbol{y} \in \boldsymbol{Y}$. Then, the best matching between the predicted and the true phases is

$$f^* := \arg\max_f \sum_{\boldsymbol{y} \in \boldsymbol{Y}} \sum_{j=1}^{l_y} \mathbb{1}_{(y_j = f(\hat{y}_j))} \tag{4.2}$$

where f is an injective function from the first K integers to the total number of true distinct phases. Notice that f has to be injective, since we do not want to allow naive solutions where two distinct predicted clusters are aligned to the same "true cluster". Results of the best alignments for $K \in \{2, \ldots, 7\}$ can be found in Table 4.4.

K clusters	K-means train	HMM train	K-means test	HMM test
2	0.447	0.449	0.498	0.494
3	0.491	0.493	0.507	0.537
4	0.490	0.511	0.464	0.547
5	0.506	0.531	0.483	0.547
6	0.344	0.461	0.302	0.472
7	0.409	0.440	0.447	0.452

Table 4.4: Accuracy results of the best alignment between true and predicted phases.

Both K-means and HMM models achieve similar results, but the HMM obtains consistently better accuracy in both training and test sets across all number of clusters, which shows that according to this intrinsic evaluation it is a better model for this type of data. This result is consistent with our prior knowledge about the model. The HMM hidden states take into account the previous hidden states when generating a phase sequence. The K-means is not aware of any time-dependencies when proposing phases, although the representation that is fed to the K-means summarizes historical information.

Experiment 5: Validity of model across workloads

Here we present the application of the method for phase detection results on more heterogeneous non-Hadoop set of workloads (Spark dataset), to demonstrate that the presented approach can be applied of different kinds of jobs, such as Machine Learning, SQL-query based, usual User Defined Functions for databases, and NLP workloads.

The goal of this experiment is to validate the methodology for different workloads. For that purpose, we use dataset B (see Section 4.3). In this particular case, we used 10GB data scale samples of the 30 TPCx-BB jobs. For the learning process we keep the same hyper-parameters from the previous experiment.

Figure 4.5 shows the phases predicted for three of the new workloads: a NLP (TPCx-BB query 19), an SQL-query based workload (TPCx-BB query 14) and a Machine Learning workload (TPCx-BB query 20). As it can be seen, similar to previous experiments different learned *regimes* capture characteristic patterns that are consistent along workload traces. This set of experiments show that the pipeline can be used not only in Hadoop traces, but also in other types of workloads. The results provide learned *regimes* that match the differenced behaviors that we would expect when looking at the workload traces.

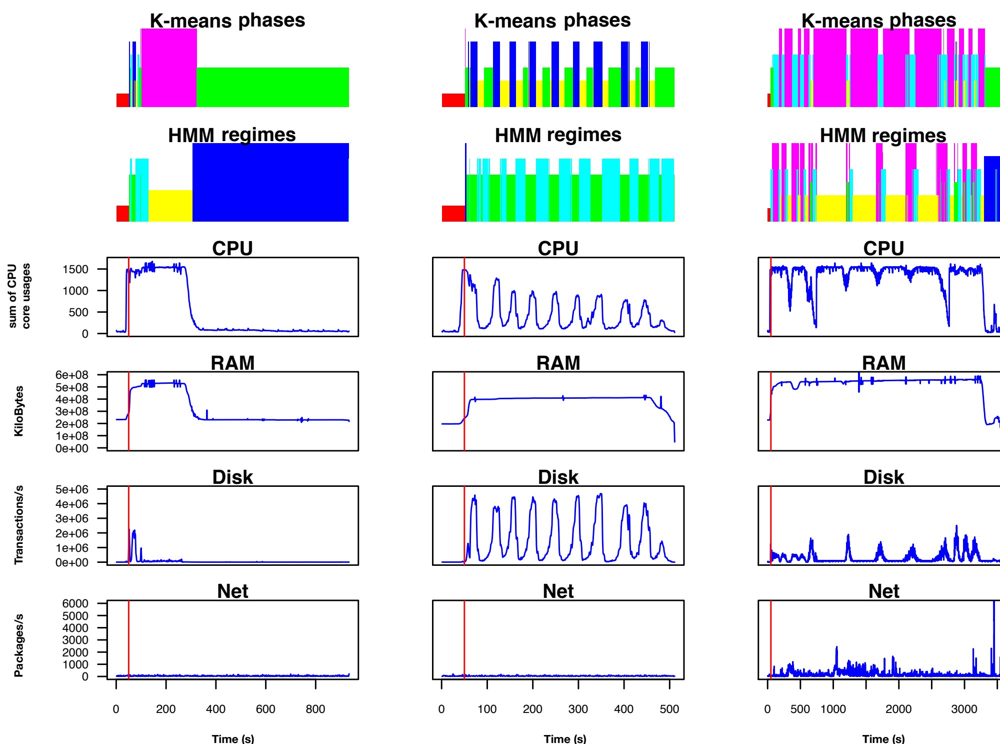


Figure 4.5: Three different applications and the predicted phases.

Experiment 6: Validating the method against a classical phase detection benchmark

To further validate the phase detection method, we have used it to predict phases in human motion data from Hsu et al. [31], a well known dataset used to validate learning of multi-dimensional time-series. This simple test illustrates that the proposed phase detection mechanism finds sensible phases. Unlike in the previous experiments, where we do not have a target to compare with, this experiment has a target value at each time step. Therefore, we can validate if the model finds phases that match the underlying label in the input sequences.

The data from [31] contains time-series with information concerning humans performing different movements. The time-series values correspond to measurements of body parts; for example, one of the dimensions of the data corresponds to the axis-angle rotation of the pelvis joint. We have prepared a couple of tests involving different motion styles, to show that the method is able to detect different behaviors from different kinds of time-series. Both tests use the original data, which contains 108 features per time step.

The first experiment illustrates the importance of the learned representation given by the CRBM. We have taken two sequences of length 2000 from the dataset, one containing walking traces and the other jogging traces. We have concatenated the sequences to create a single example of length 4000. Then we have trained 30 K-means models (with different random initializations); 15 models use the original data and the other 15 use the processed data by the CRBM. Figure 4.6 shows the results of the phases given by the 30 models. The top 6 outcomes correspond to the different results of the 15 K-means trained with the original data. The bottom two outcomes correspond to the different results of the other 15 models. Notice that while raw data (six top series) produces inconclusive results, passing data through the CRBM allows K-means to discover a single stable pattern. The CRBM version produces two patterns which are actually the same if we flip the labels. Moreover, these two solutions match the walking and the jogging phases with some mixing around time step 3000.

For the second experiment, we have selected four traces of length 500 containing "walking" at slow/normal speed and "jogging" at slow/normal speed. Then we have concatenated the traces to create a single sequence of length 2000. We trained several times with different random initializations 3 types of pipelines. The first pipeline is a simple K-means using the original trace. The second pipeline is a CRBM followed by a K-means. The third pipeline is a CRBM followed by an HMM.

The first sequence in Figure 4.7 shows one of the several possible solutions of the K-means. As in the previous experiment, the results depend greatly on the random initialization. The second sequence shows one of the two possible outputs given by the CRBM K-means pipeline (the other is the same with the labels flipped). The third sequence shows one of the two solutions given by the CRBM-HMM pipeline (and again the other is the same with the labels

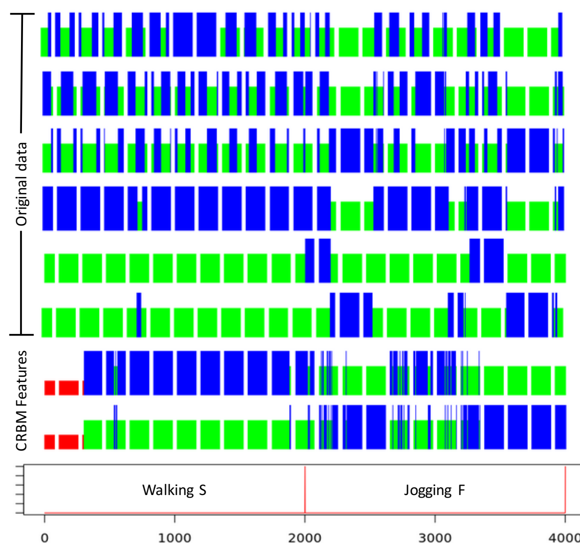


Figure 4.6: Results for different random initialization of K-means.

flipped). Note that the tags are Jogging and Walking and both tags contain two different speeds: medium (M) and slow (S).

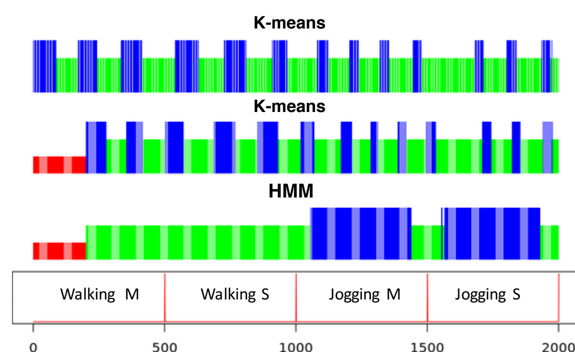


Figure 4.7: Phase prediction on human body data.

We can see that the CRBM-HMM pipeline is able to correctly differentiate the walking phase from the jogging phase, with some error around position 1500, where the trace behavior changes from jogging at middle speed to jogging at slow speed. This is a sensible error since we concatenated the data from different examples of the original dataset. In the original dataset, the actions start and finish in each timeseries. Therefore, the movements of a person are recorded from the moment at which an action starts and until it ends. When a person starts jogging it starts with velocity 0. Therefore, the first seconds in the "Jogging M" or "Jogging S" traces track the action of a person that is starting to run but has not started the "jogging action". This happens at the beginning and at the end of each sub-sequences, which is precisely the points at twhich the predicted phase changes.

4.4 EXPERIMENTS: EVALUATING PHASE DESCRIPTOR IN AN AUTO-SCALING TASK

In Section 4.3 we have shown that the proposed pipeline composed of a CRBM and a clustering algorithm can learn interesting phase descriptors for different datasets. This section presents an application of the proposed method evaluated on two tasks.

- Phase forecasting task: We train time-series forecasting methods to predict future phase state values from an sliding window of historical phase values.
- Resource Auto-scaling task: We use a forecasting method to proactively scale up or scale down resources in a container. The decision is taken when the forecast detects a phase change in the workload trace that requires a resource change to adapt the running environment to the future expected application requirements.

This section uses a private dataset from the IBM Watson platform. The dataset is comprised of executions of "Deep Learning as a Service" (DLaaS). We use 5000 container traces for training and 500 for testing.

Experiment hyper-parameters

The experiments performed in this section use a CRBM with a history of 3 time-steps. Therefore, if the sampling period in the data is 15 seconds, the time window used to discover phase is one minute (of 15 seconds + 45 seconds of history). Tuning the hyper-parameters, we find the minimum loss with a learning rate $lr = 0.0001$, trained for 2000 epochs with 10 hidden units. Here we use a k-Means for assigning phase ids because it proved to work as well as a HMM. The K in the k-Means was selected studying the Square Sum Within clusters (SSW) [26]. We find that under 90% of the SSW is explained with $k = 5$, with little gains for higher K values. Therefore, we use $k = 5$.

Experiment 1: Phase Forecasting

In order to choose a forecasting method, we have tested different models to predict the next phase descriptors within a window of 1 minute, which corresponds to 4 time-steps. These methods vary from naive policies to sophisticated neural network-based approaches, including:

1. the most frequent observed phase (the mode) from the previous time window as expected phase along with the next time window;
2. the last observed phase as an expected phase for the next time window;

3. the predicted phases for the next time window using a one-hot encoded MLP, forecasting the future window from the current window;

4. Long Short Term Memory (LSTM).

Predictions made on an example container using various models are shown in Figure 4.8. This figure shows a slice of the resource usage coded as phase values, the slice shown contains a phase change around position 4. We do see that both MLP and LSTM models can predict the sudden phase change from "green" to "red" correctly when observing four continuous "green" phase. The comparison among prediction methods is shown in Table 4.5

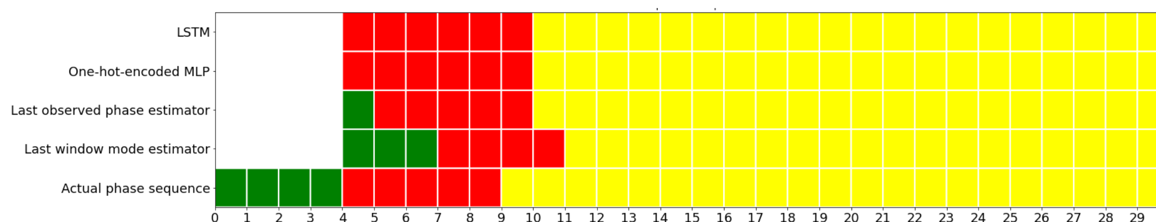


Figure 4.8: Phase prediction via different models.

with the average F1 score of the different models. As it gets harder to predict more steps ahead of time, we show the F1-scores for each of the future four steps separately.

	t=1	t=2	t=3	t=4
Last window mode estimator	0.654	0.614	0.579	0.554
Last observed phase estimator	0.730	0.689	0.648	0.613
One-hot-encoded MLP	0.926	0.876	0.827	0.786
LSTM	0.913	0.863	0.812	0.756

Table 4.5: Comparison of the prediction models.

Results show that MLP and LSTM models tend to perform better than the basic methods. It is worth to notice that the performance of the "last phase estimator" is not bad. We observe that phase changes rarely happen for some long-running containers, and the container stays on a particular phase throughout its whole life span. For such containers, using the previous phase as the next will not lead to many errors. In our ML workloads, MLP and LSTM give better accuracies by learning typical phase changes. Since the quantitative results of LSTM and MLP are similar, we use the MLP for our next experiment as it is simpler and faster.

Experiment 2: Resource Auto-scaling

When applying our phase-detection and phase-forecasting in container auto-scaling, we explore two different types of auto-scaling policies: reactive policies (used in [7, 68]) and proactive policies (policies based in forecasting).

Reactive policies: such policies resize containers according to the information observed in the past. We can further classify the reactive policies into two categories according to the way to determine the container size. 1) The time window statistic approach resizes containers according to the maximum or the top 95th percentile observed in the previous time window. 2) The phase approach identifies the current phase based on resource usage data in the previous time window and uses the phase's profile, namely the statistics of all resource usage data identified as this phase (that may include data from multiple containers), to resize the container.

Proactive policies: such policies allocate resources through forecasting resource usage, i.e., predicting the next several phases and using their profiles to resize the container proactively. When a proactive policy decides how much to provision, it has taken into account the size of the "next window" to be predicted, affecting the statistical profile to be chosen. Larger prediction window leads to more proactive actions and more phase profiles to choose. Our policy chooses the maximum of the candidate profiles for that future window, noticing here that the forecasting window size becomes a hyper-parameter to study.

Regardless of the policy chosen, we denote the "maximum observed" or "N^{th} percentile" by $\mu + \{1 \ldots 3\} \cdot \sigma$, to prevent few outliers lifting the limit unnecessarily. Besides, when applying phase profiles, we discard the "maximum observed" because it contains the maximum deviation across all observed containers.

Finally, we need to decide how frequently to predict and resize the containers. Two strategies are available: 1) we resize the containers periodically each N minutes, or 2) we trigger the resizing only when necessary (when there is a need for more resources). In the IBM Cloud services, metrics can be collected every 15 seconds as a "step" where a phase can be detected (i.e., with $d = 3$, the CRBM + clustering can detect a phase with a time window of 4 steps = 1 minute).

Here we apply an auto-scaling policy that 1) at each time step predicts phases in the next time window (1 minute), 2) retrieves the required CPU and Memory resources from all predicted phase profiles, and 3) determines the maximum resource demand from all predicted phase profiles (taking the percentile rule into account as mentioned above). If the predicted resource demands indicate an increase over a given tolerance (namely 10% of the current demand), the container is scaled up with the predicted resources. And when the predicted resources indicate a decrease of demand below the current limit, we scale

the container down with larger tolerance to avoid some slight usage fluctuations causing frequent scaling up and down.

Evaluation of Auto-scaling policies

In the previous section, we show that we can predict the phases for the next time window using the MLP model, and in this section, we show how a proactive auto-scaling approach can improve the overall resource efficiency over the traditional reactive policies. Besides, we are also concerned about whether the killed containers due to an Out-of-Memory (OOM) error can be reduced by foreseeing future phases.

In Figure 4.9, we show an example of a DLaaS container. Here we can see that the reactive approaches provide a fitter lower bound of the true resources, but they produce a lot of resizing operations, and they sometimes under-provision the memory, which in practice would lead to an Out-Of-Memory error to kill the container. Our proactive approach shows to be more conservative on resource provisioning, causing no OOM and CPU throttling, and with minimal resizing actions along with the execution.

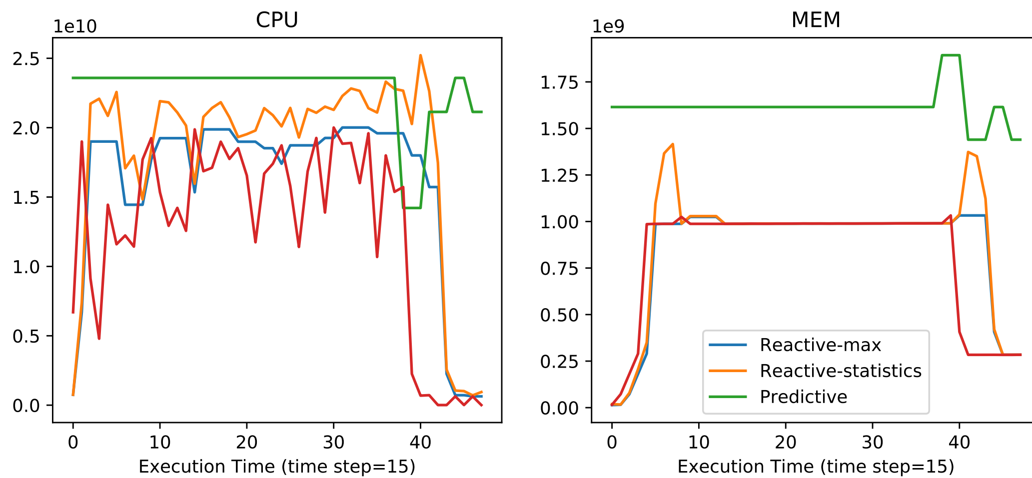


Figure 4.9: Example of DLaaS execution, with the provisioned CPU and Memory for each policy.

Overall System Performance

Finally, we evaluate the overall system performance under different policies. Here we measure the total slack for the different policies (amount of resource over-provisioning), the number of container kills due to the OOM errors, and the number of resizing actions produced by each policy.

Table 4.6 shows the results of auto-scaling performance for each policy in comparison. We observe that the average number of auto-scaling changes for the reactive-max strategy

is very high, with 33 changes, whereas the proactive and reactive-statistics policies have a much lower number (around 3-4 changes) per container. It indicates that container resource usage tends to fluctuate over shifting time windows, so reactive policy scales up and down a lot over time.. In our case, both predictive and reactive make minimal changes as the usage fluctuations are probably identified as a particular phase. The principal advantage of the proactive policy is to foresee potential phase transitions, avoiding container kills due to OOM errors. Overall, our phase prediction based proactive auto-scaling policy reduces the amount of CPU over-provisioning from $10 \cdot 10^8 CPU \times hour$ to $6 \cdot 10^8 CPU \times hour$. Besides, we find that our proactive policy results in much fewer container kills (20) due to OOM errors, while both reactive policies result in hundreds of container kills, 357 (reactive- statistics) and 402 (reactive-max) OOM errors, respectively. Therefore, our prediction based proactive policy appears to be much safer than reactive approaches with fewer resizing operations. Besides, we find that the proactive policy results in much fewer container kills (20) due to OOM errors, while both reactive policies result in over 300 container kills.

	Predictive	Reactive-max	Reactive-stats
Auto-scaling changes	3.74	33.28	3.40
OOM containers	20	402	357
CPU Over-provision	6.37e+08	10.20e+08	101.79e+08
Mem Over-provision	5.03e+07	3.74e+07	2.28e+08

Table 4.6: Average number of auto-scaling changes, number of OOM containers for each policy, and over-provisioning in *CPU_hour* and *KBytes_hour*

4.5 CONCLUSIONS

In this work, we present a method for modeling and discovering phases in time-series in an unsupervised way, by using Conditional Restricted Boltzmann Machines to encode n_v dimensional feature input vectors into n_h dimensional vectors, taking the time dimension into account, and feeding them to HMMs. We understand as "phases" periods of time displaying similar behaviors.

Workload profiling and resource consumption phase detection are very relevant problems in the areas such as High Performance Computing (HPC) and Cloud Computing. For this reason, we validated the approach on a couple of datasets containing traces from application executions on Data Centers: One dataset containing executions traces of Apache Hadoop jobs and the other dataset containing Spark jobs. Such a scenario implies multi-dimensional time-series data, without either clear labels or clear expert methods for automatically identifying

phases. The proposed approach does not require feature engineering, so it can be easily automated, thereby helping decision systems when applications become more complex. Moreover, we find no reason to consider that this method can not also be used for other similar scenarios with time-series.

To verify the validity of the phases, we have presented some sequential performance data to the model, such as workload traces from the ALOJA dataset as a case of use towards data-center management and application characterization. The model is able to generate phases that, upon careful examination on the workload traces, separate different behaviours found in the telemetry traces. Further, as a known case towards a sanity check, the *Motion* dataset used for evaluating time-series. The proposed approach is able to identify distinct behaviors in both cases. In the principal case for the workload traces, we are able to verify that the proposed phases capture different properties from the workloads, consistently characterizing executions by resource consumption for different kinds of application. In the case of the Motion data we are able to show that the pipeline would differentiate walking from jogging traces.

From the experimental results, we have find that CRBMs plus clustering algorithms are able to discover phases on different workload executions, each one corresponding to a specific resource usage pattern. Given that one of the used datasets corresponded to Hadoop executions, we are able to compare the discovered phases with the different Hadoop stages, with the observation that different Hadoop workloads have different behaviors on same phases. This enable us to identify characteristic patterns not only for complete executions, but also for parts of an execution. This method also allows us to generate automatically a fingerprint for applications which can be used to identify them.

We test the phase detection pipeline in an auto-scaling problem using workload data from IBM DLaSS containers. The experiments show that machine learning models trained on top of the phase descriptors can foresee phase transitions. We can use this information to proactively resize containers to accommodate the sudden changes in resource demands. By modeling phase transitions using an MLP model, we prove that such sudden behavior changes can be predicted via phases and can be leveraged in a proactive auto-scaling policy. Evaluations show that our proactive auto-scaling policy can significantly improve the resource efficiency in a safer (from 65% of potential OOM scenarios to 4%) while maintaining the same amount of auto-scaling changes than the best of the reactive policies.

5

SEQUENCE-TO-SEQUENCE MODELS FOR RESOURCE ESTIMATION OVER TIME UNDER CO-SCHEDULING.

This chapter describes the second contribution of this thesis which aims to estimate the resource behaviour of two applications sharing hardware resources. Modern cloud environments can share hardware resources in order to improve their operation efficiency. Nevertheless, co-locating applications can drastically degrade the performance of the running workloads or, in some cases, it can yield to failed executions. In order to estimate how applications might behave under different co-scheduling pairs of applications we train a machine learning model that takes as input the trace of two input applications and outputs the expected trace of resources when two applications share the computational environment.

Section 5.1 presents an overview of the problem. Section 5.2 presents our proposed solution to model the output trace of two co-located applications from the input traces of the applications executed in isolation. Section 5.3 presents an evaluation of our work. Finally, Section 5.4 presents the conclusions of the chapter.

5.1 INTRODUCTION

Resource under-utilization is one of the major problems in data center management, since the average utilization is estimated to be below 50% [66][13] due to over-conservative scheduling policies, to ensure good QoS levels. However, most applications do not use resources continuously, even when there is full quota on CPU, memory and I/O to avoid degrading the QoS or violating Service Level Agreements (SLA). Flexible policies allow resource sharing among applications, while at the same time risking concurrent applications to top their requirements. Sharing resources between applications in data centers is crucial to achieve efficient utilization of those resources, reducing power consumption, allowing proper scaling out for currently running applications or accepting new applications into the system.

The principal problem when co-locating resource-sharing applications is to ensure that competition will not ruin their QoS, even when a certain tolerable degradation is expected. Co-

located applications do not need to behave as "the sum of their behaviors", since interference creates a new characteristic footprint for each set of concurrent applications.

The statistical analysis of monitored metrics is fundamental when automating this workload placement. Since resources are finite, smart resource sharing is encouraged by administrators in order to increase resource availability while maintaining energy efficiency [8]. Different approaches exist for predicting resource demand and interference (slow-down produced by resource competition) on co-located applications.

Most applications exhibit differentiated resource demands over time resulting in resource consumption phases [61, 75, 85]. Our method aims to predict the resource demand over time, introducing the temporal dimension to the resource demand prediction problem. Nevertheless, most methods reduce the interference prediction problem to a regression problem. Therefore, they produce a single value global prediction estimate, instead of a sequence of predictions over time.

Most classic machine learning approaches for this problem, such as [20] or [53], do not consider the temporal dimension of executions, and therefore do not predict resource usage over time. This means that schedulers are obliged to optimize fixed blocks of expected resource usage. In this scenario, schedulers can take suboptimal decisions to block possible colocations of applications that only complete for a fraction of their execution.

In this work we present ResourceNet, a workload-to-workload forecasting methodology to predict the effects of application co-location interference, using sequence-to-sequence models based on Recurrent Neural Networks (RNNs). RNNs are powerful models with the capability of dealing with time series. Of equal importance, RNNs are able to deal with inputs and outputs of arbitrary lengths. The method presented herein predicts the footprint for resource demands of co-located applications, given that the traces of these applications run in isolation. Furthermore, we show how this model can be used to predict accurately the run-time of applications sharing resources.

Our model employs two GRU models [16] as building blocks; one GRU processes the trace signal of the incoming applications and passes the processed information to the other GRU, which outputs the expected resources of the co-located applications over time. The model predicts the whole resource demand trace throughout execution, thereby providing schedulers with a sufficiently accurate estimation for placing applications together and thus minimizing interference. Training the model using a diverse set of benchmarking applications enables it to attempt predictions to unseen co-located applications. The recurrent nature of the model allows it to process input sequences or different (and arbitrary) lengths. Moreover, it is able to generate output sequences of arbitrary length, thereby making our solution adaptable enough to address the problem in question.

We validate the proposed methodology by computing the error of the predicted resources (of co-joined application traces) with respect to the real resources. We use benchmarks from

Big Data applications (from both Apache Hadoop and Apache Spark) as workload data. We have selected Hadoop and Spark benchmarking workloads because of their popularity in HPC applications, where scheduling and environment configuration have a high impact on the application performance. Experiments measure the prediction of the low-level resource usage (i.e. CPU, memory and I/O) of pairs of co-located workloads over time. The different benchmarks we use belong to the Intel HiBench [32], the IBM SparkBench [43] and the Databricks Spark-perf benchmarks [72]. The method is trained and evaluated on low-level monitoring metrics which are reasonably available on any HPC setting. Our method is compared against different simpler machine learning alternatives in order to assess the possible drawbacks of using standard models.

For our experiments, we created a dataset with execution traces from the previously mentioned benchmark suites. The dataset consists of triplets $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{a} \wedge \boldsymbol{b})$ where $\boldsymbol{a}$ and $\boldsymbol{b}$ contain the traces of the isolated executions from two applications, and $\boldsymbol{a} \wedge \boldsymbol{b}$ represents an execution of the co-located pair. To build a reasonable dataset, we generated scenarios in which both applications did not start at the same time. In particular, we created co-scheduling situations in which the start time for one of the applications was shifted by a factor of 25%, 50% and 75% of the length of the longest application being co-scheduled.

CONTRIBUTIONS OF THE CHAPTER

The main contributions of our work presented in this chapter can be summarized as follows:

- A novel use of Recurrent Neural Networks that estimates the monitored metrics of two co-scheduled applications $\boldsymbol{a} \wedge \boldsymbol{b}$ from the information of $\boldsymbol{a}$ and $\boldsymbol{b}$ gathered running the applications in isolation.

- A novel feature, *percentage completion time*, for estimating the completion time of co-scheduled applications. This feature improves predictions made by using the standard stopping criteria based on the *end of sequence* feature.

- A comprehensive evaluation of the method against other relevant machine learning approaches. We show the advantages of our method, which are especially noticeable in two cases: when co-located executions have different lengths, and when co-scheduling heavily impacts the execution time of the applications due to high interference.

The proposed method can be used to characterize applications according to their compatibility with other workloads. Furthermore, it enables resource managers to plan resource allocation and load balancing better in advance.

5.2 METHODOLOGY

ResourceNet is a fully learnable system based on a sequence-to-sequence architecture with an attention mechanism. The model takes as input pairs of sequences containing measurements of applications run on isolation. The input sequences are aggregated and fed to a decoder which produces the expected measurements of running both co-located applications sharing resources. Since the model is based on a sequence-to-sequence architecture it can naturally work with sequences of different lengths, which makes it suitable for the scenario presented here, in which measurements belong to the workload traces.

The reader can note that the model assumes that both applications are already executed in isolation, while this is certainly a limitation, it is not unexpected on a Cloud scenario to have a Sandboxing service, that executes applications in isolation. Having executions of applications prior to predicting application interference is a common hypothesis in this area [53]. In some cases, even partial co-executions of the applications are performed [19] in order to assess interference.

Figure 5.1 shows the diagram for the model, indicating the inputs and outputs, and the four building blogs composing the data pipeline. The entire shaded box represents the whole model at a high level. The first component, $\begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{bmatrix}$, performs the joining operation where input sequences $\boldsymbol{a}$ and $\boldsymbol{b}$ are stacked to form a single matrix of measurements; this process is explained in detail in Subsection 5.2.1. The second component, RNN_{enc}, encodes the input sequences with a GRU and produces as output a matrix $\boldsymbol{E}$. The third component, $\boldsymbol{E}$, represents the stored encoded input sequences. The fourth component, $\boldsymbol{A}$ is the attention mechanism, which receives the hidden state from the decoder and generates a context vector that is fed to RNN_{dec}. The last component of the diagram is the decoder. The decoder is a GRU that generates as output the expected resource demands of the co-scheduled applications. We denote those resource predictions by $\boldsymbol{a} \wedge \boldsymbol{b}$.

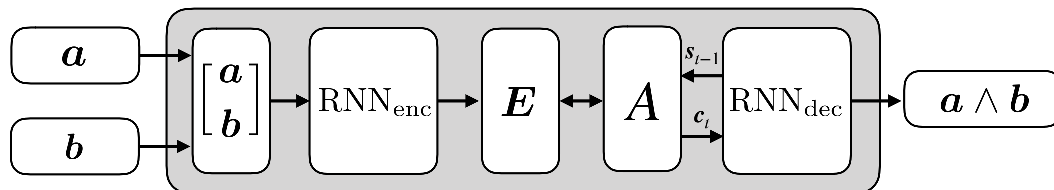


Figure 5.1: Input-output diagram of the proposed model

The learning process is performed by minimizing the Mean Squared Error (MSE) loss function, which computes the error between the predicted outputs and the true trace values at every time step. We update the parameters of the model using a minibatch gradient descent procedure. Section 5.3.2 provides a detailed description of the hyper-parameters of the model and the training procedure.

5.2.1 *Stacking measurements of the input sequences*

The input sequences of the model are stacked in order to generate a single matrix of measurements. Given two vectors $\boldsymbol{v} = (v_1, \ldots, v_p)^\top$ and $\boldsymbol{w} = (w_1, \ldots, w_q)^\top$ we will denote by $\left[\begin{smallmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{smallmatrix}\right]$ or $[\boldsymbol{v}; \boldsymbol{w}]$ the vector $(v_1, \ldots, v_p, w_1, \ldots, w_q)^\top$.

Given two arbitrary sequences $\boldsymbol{a}$ and $\boldsymbol{b}$ of length $\mathrm{l}(\boldsymbol{a})$ and $\mathrm{l}(\boldsymbol{b})$ respectively we denote by $\left[\begin{smallmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{smallmatrix}\right]$ or $[\boldsymbol{a}; \boldsymbol{b}]$ a sequence of length $N = \max(\mathrm{l}(\boldsymbol{a}), \mathrm{l}(\boldsymbol{b}))$ containing the measurements of both sequences at every time step. By convention we write the longest sequence first. Since sequences may differ in length we append the shortest sequence with zeros to match the length of the longest sequence. If we denote by $\mathbf{0}_d$ a vector containing d zeros where d is the number of elements in $\boldsymbol{b}_k$ we have:

$$\begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{bmatrix} := \left(\begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{b}_1 \end{bmatrix}, \ldots, \begin{bmatrix} \boldsymbol{a}_{\mathrm{l}(\boldsymbol{b})} \\ \boldsymbol{b}_{\mathrm{l}(\boldsymbol{b})} \end{bmatrix}, \begin{bmatrix} \boldsymbol{a}_{\mathrm{l}(\boldsymbol{b})+1} \\ \mathbf{0}_d \end{bmatrix}, \ldots, \begin{bmatrix} \boldsymbol{a}_{\mathrm{l}(\boldsymbol{a})} \\ \mathbf{0}_d \end{bmatrix} \right)$$

Example: Let us consider $\boldsymbol{a} = (\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3, \boldsymbol{a}_4)$ and $\boldsymbol{b} = (\boldsymbol{b}_1, \boldsymbol{b}_2)$ containing measurements on $\mathbb{R}^2$. That is $\boldsymbol{a}_i, \boldsymbol{b}_j \in \mathbb{R}^2$ for any valid i, j. Then the generated matrix for two applications that start at the same time is $\left[\begin{smallmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{smallmatrix}\right] := \left(\left[\begin{smallmatrix} \boldsymbol{a}_1 \\ \boldsymbol{b}_1 \end{smallmatrix}\right], \left[\begin{smallmatrix} \boldsymbol{a}_2 \\ \boldsymbol{b}_2 \end{smallmatrix}\right], \left[\begin{smallmatrix} \boldsymbol{a}_2 \\ \mathbf{0}_2 \end{smallmatrix}\right], \left[\begin{smallmatrix} \boldsymbol{a}_4 \\ \mathbf{0}_2 \end{smallmatrix}\right] \right)$.

The notation $[\boldsymbol{a}, \boldsymbol{b}]$ is reasonable for describing the situation in which the model is fed with traces of programs that start execution at the same time. Nevertheless, we may want to predict scenarios where applications do not start at the same time. In order to provide the network with such information, we pad a sequence with zeros before it starts. If we wish to tell ResourceNet that application $\boldsymbol{b}$ starts k time-steps after $\boldsymbol{a}$, we add k vectors containing zeros at the beginning of $\boldsymbol{b}$. In order to make ResourceNet capable of making predictions in that environment, where applications are not required to start at the same time, our training data will contain co-scheduled applications starting with different time delays.

5.2.2 *Encoding input traces, Decoding co-scheduled trace*

The encoding process reads the traces from the isolated runs and generates the matrix $\boldsymbol{E} = \mathrm{RNN}^*_{\mathrm{enc}}(\boldsymbol{x}_{1:n}; \boldsymbol{\theta})$, using the equations of the GRU from Subsection 2.3.1. To this end, we use a GRU as our encoder. The notation $\mathrm{RNN}^*_{\mathrm{enc}}(\boldsymbol{x}_{1:n}; \boldsymbol{\theta})$ represents the concatenation of the n hidden states of the recurrent net when processing the input sequence $\boldsymbol{x}_{1:n}$.

The decoding process takes the produced $\boldsymbol{E}$ as input and generates the output sequence one vector at a time. This output vector has as many components as it does features we wish to predict (in our case, 9), plus some features used to decide when applications finish. A detailed description of the features can be found in Section 5.3. The decoder $\mathrm{RNN}_{\mathrm{dec}}$ receives as input at time t the vector $\boldsymbol{x}_t := [\tilde{\boldsymbol{y}}_{t-1}; \boldsymbol{c}_t]$ which is the concatenation of $\tilde{\boldsymbol{y}}_{t-1}$

and $\boldsymbol{c}_t$ (in addition to any recurrent inputs). Vector $\tilde{\boldsymbol{y}}_{t-1}$ is the output predicted by the decoder at the previous time step. Vector $\boldsymbol{c}_t$ is the "context vector" generated by the attention mechanism of the decoder. Given a hidden state $\boldsymbol{s}_t$ and a input $\boldsymbol{x}_t$ the predicted resources of the colocated applications at time t is $\tilde{\boldsymbol{y}}_t = \mathrm{RNN}_{\mathrm{dec}}(\boldsymbol{x}_t, \boldsymbol{s}_t; \boldsymbol{\theta})$.

Predicting completion time

We have experimented with two mechanisms for predicting the completion time of the co-scheduled jobs. The standard End of Sequence (EoS) feature approach [73] and our own Percentage Completion (PC) feature approach.

End of Sequence feature

The first strategy, based on an EoS feature, is the standard approach used to decide when an RNN should stop producing more vectors. The idea of using feature to encode the stopping time can be found in [73]. This feature vector takes value 0 at every time step except the last one, where it takes value 1. This vector simply tells the decoder when both applications finish. If our decoder can predict EOS with reasonable precision then we can build stopping criteria based on those values to predict the completion time of the co-scheduled applications. Nevertheless, in our experiments, this was not a successful strategy to stop the decoding process. We can provide a reasonable argument on the poor behaviour of this strategy. If the decoder produces always a 0 for the EoS feature, it will guess correctly the outcome of that feature for all the elements in the sequence, with the exception of the last one. Therefore, there is little incentive during training to change this behaviour, since the penalisation for correctly predicting a 1 in the last time step is quite small, compared with the benefit of predicting the correct value at all the other time steps. This insight lead us to present the Percentage Completion Features, that penalize a decoder that produces always a zero in the feature used to stop the decoding process.

Note that the presented EoS approach for our problem is not equivalent to the analogous problem in other domains such as Neural Machine Translation (NMT). In NMT, the decoder emits conditional probability over the vocabulary at every time step which can be used to select the word with the highest probability. In that scenario, there is a token EOS that is just a special word that stops the decoding process, which means that if the decoder emits the EOS symbol the decoding process is stopped. In our setting the EoS is a new feature that takes a real value at every time step.

Percentage Completion Features

The second strategy consists of a novel approach based on two additional features (one per job) which we denote with PC features. PC features keep track of the percentage completion of the workloads, providing ResourceNet with extra information relevant to the job estimation runtime.

We denote by $\text{PC}_{\boldsymbol{a}}$ and $\text{PC}_{\boldsymbol{b}}$ the percentage completion features for input sequences $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively. Both features contain at time t how much of the workload has been completed until t, expressed as a percentage. For a given sequence $\boldsymbol{s}$ of length N, we define $\text{PC}_{\boldsymbol{s}} = \left(\frac{1}{N} \cdot 100, \frac{2}{N} \cdot 100, \ldots, \frac{N}{N} \cdot 100\right)$. Notice that, by construction, PC features contain monotonically increasing values that must finish with value 100. Nevertheless the rate of increment at every time step will depend on the overall number of time steps of the sequence.

Example: Let us consider $\boldsymbol{a} = (\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_{100})$ and $\boldsymbol{b} = (\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{300})$. Then $\text{PC}_{\boldsymbol{a}} = (1, 2, \ldots, 100)$ and $\text{PC}_{\boldsymbol{b}} = (0.333, 0.666, 1.0, \ldots, 100)$ are two vectors of length 100 and 300 respectively.

5.3 EXPERIMENTS

Workload data

In order to capture the trace of each execution we have profiled them by using the basic Linux performance analysis tools: *vmstat*, *iostat*, *ifstat* and *perf*. These tools gather system performance metrics of running processes in a non-invasive way. Additionally, these tools have a lower performance impact in the system, causing negligible overhead to the executions. From these tools we have gathered a total of 141 features with time granularity of one second.

For our study we have selected 9 key features, shown in Table 5.1, that are especially relevant for interference prediction and resource estimation [63][20][50].

The dataset used in the experiments contains traces generated by a variety of micro-benchmarks (workloads). The workloads used have been extracted from different suites: HiBench [32], Spark-perf [72] and SparkBench [43]. These suites contain different Big Data workloads, and have also been used in similar studies [82][35][51][15]. The benchmarks include machine learning, data mining and big data benchmarks. Some examples of workloads are executions of *PageRank*, *K-means*, *Naive Bayes*, *Logistic Regression*, etc. The traces are executed using a server with two Intel Xeon E5-2630 processors and 128 GB of RAM, up to 400 isolated and co-located executions.

Feature	Description
CPU	Percentage of total CPU used
RAM	Amount of Bytes of main memory used
IOR	Blocks received from a block device (blocks/s)
IOW	Blocks send to a block device (blocks/s)
CPI	Cycles per instruction
LLCM	Last level cache misses
FLCM	First level cache misses
PF	Page faults
TLBM	Translation lookaside buffer misses

Table 5.1: Workload metrics recorded at each time step.

5.3.1 *Dataset description*

The dataset has been created using workloads from HiBench [32], Spark-perf [72] and Spark-Bench [43]. The dataset is composed of workloads *triplets*. Each triplet contains a combination of three execution traces. The first two traces correspond to isolated executions of the two applications. The third trace contains the execution of the co-located application from the first two traces. Therefore, the data has the form $D = \left\{(\boldsymbol{w}^{[i]}, \boldsymbol{w}^{[j]}, \boldsymbol{w}^{[i]} \wedge \boldsymbol{w}^{[j]}) \mid i, j \in I\right\}$ where I is a set of indices of the workloads.

In real world scenarios, co-located applications do not necesarilly start at the same time. In order to increase the co-location cases in our collected dataset, we prepared different scenarios where co-located applications $\boldsymbol{a}$ and $\boldsymbol{b}$ start with different delays. For the benchmarking executions, one of the concurrent applications is delayed to start after its co-located peer application. The phase differences used in the dataset generation are 0 (synchronized), 0.25, 0.5 and 0.75. A phase difference of 1.0 is not taken into consideration since it would mean applications running completely in isolation, one after the other. Notice that bigger differences usually give rise to less interference, since applications have fewer runtime sharing resources.

We have executed 500 random pairs of applications from the three previously mentioned benchmarks with randomly generated shifts. The executions are used to create a train set containing two thirds of the pairs and a test set containing the remanding executions.

ResourceNet is validated through a test set of executions which are used to evaluate the prediction capabilities in different job co-location scenarios. We use the Mean Absolute Percentage Error (MAPE) to assess the quality of the predictions at every time step of the execution trace. We have selected MAPE because it is an easily interpretable and well stablished metric to evaluate regression models that has already been used in relevant related work [41, 56]. In order to evaluate our work we have designed two experiments:

- In Experiment 1 (subsection 5.3.3) we evaluate the quality of the predictions when both applications are running with the different models presented. We present a table with the error metrics computed in a test set.
- In Experiment 2 (subsection 5.3.4) we evaluate the accuracy of the model presented when predicting the runtime of co-scheduled applications. We present experiments with different methods to assess the job runtime of the co-scheduled applications.

5.3.2 *Experiment settings*

The experiments performed in this section use an encoder and a decoder with 512 hidden units trained during 1000 epochs with a learning rate of 0.001. We have used standard SGD algorithm to train the model. This architecture was selected after evaluating different models with the same number of hidden units in the encoder and decoder. We evaluated architectures with 128, 256, 512 and 1024 hidden units. The best results were obtained with 512 hidden units. We implemented the model using PyTorch [58], a Python library that already provides the building blocks for our model such as the GRU layers for the encode and the decoder. The other models used in this chapter are from scikit-learn [59].

Evaluation methodology

To evaluate the advantages of the proposed method we compare it with other sensible alternatives. Firstly, we use a naïve baseline model to estimate resources. The baseline model predicts the resource usage at time t as the sum of the resources of the isolated applications at time t. This means that the output at time t for a given input $[\boldsymbol{a}; \boldsymbol{b}]$ is $\tilde{\boldsymbol{y}}_t = \boldsymbol{a}_t + \boldsymbol{b}_t$. Notice that $[\boldsymbol{a}; \boldsymbol{b}]$ is a matrix of features that already contains padded zeros to encode any temporal phase difference of sequences (should they exist). Therefore, if $\boldsymbol{b}$ starts k time steps after $\boldsymbol{a}$ then this baseline should predict correctly the features values of the co-located applications for the first k time-steps (since there is a single application running and there is no competition for resources). Nevertheless, when two applications are co-executed the resource behaviour is not easily predicted as a sum of the resources in isolation.

We also compare our approach with a Linear Regression (LR) and a MLP. The two regression models present a natural improvement over the baseline approach, because learning is involved in order to adjust the predicted co-execution resources, instead of using a simple addition. Both methods predict the resources usage at time t as a function of the input features $\boldsymbol{a}_t$ and $\boldsymbol{b}_t$. The baseline, the LR and the MLP take an input vector containing measurements from the isolated sequences at time t and estimate the features of the the co-located sequence at time t. This approach involves three critical problems:

- The temporal nature of the data suggests that values at time t might be correlated with nearby values. This is not taken into consideration.
- If the input has length n and the output has length $n + k$ there will be k time-steps where the models cannot make predictions, because the model has no input data from which to make predictions.
- A model predicting co-execution resources needs to predict the overall runtime of the co-execution, otherwise the expected resources might be "expected forever".

The first problem can be somewhat mitigated by using a sliding window mechanism over the input data. Nevertheless, including a sliding window mechanism increases the complexity of the solution and adds a new hyper-parameter to be tuned (the length of the sliding window).

The second problem is even harder to assess. A sensible approach could be to pad the execution with zeros in the k time steps where there is no data. However, this solution invents time-steps with no resource usage and does not address the issue that, in reality, co-scheduled executions resource consumption "stretches in time" when co-scheduled jobs share resources. Models therefore require some sort of memory from past values in order to understand the effect left produced by the co-location over time.

The third problem cannot be solved by sliding window models because, by construction, such models receive as input a window of resources and produce an output vector of resources. Therefore, such techniques are only aware of a sliding window of resources. Since these techniques do not keep track of the number of generated time-step resources during prediction they do not provide an expected finish time of the predicted trace. Therefore, they are not suitable in an scenario where it is critical to assess how low applications might run together, in order to make scheduling decisions.

Sequence-to-sequence models with an attention mechanism naturally deal with these three issues because the hidden state of the RNN cells allow the model to "keep track" of previous resources and we can use a feature to train the model to predict the expected termination time of the co-scheduled applications.

5.3.3 *Experiment 1: Resource usage predictions*

In this experiment we evaluate the accuracy of the predictions made by the different models on co-scheduled jobs. Table 5.2 contains the MAPE errors of the predictions on the test set. The best results are obtained by ResourceNet, followed by the MLP. Notice that for some metrics, such as CPU and IOR, the error produced by the proposed model is reduced more than half when compared to the other models. Moreover, the model presents a lower standard deviation in most cases.

	baseline		linear regression		multilayer perceptron		ResourceNet	
	MAPE	std	MAPE	std	MAPE	std	MAPE	std
CPU	14.9	19.6	16.5	12.7	14.2	12.7	6.4	10.4
CPI	24.4	14.9	7.6	6.0	7.5	5.9	4.1	5.9
IOR	13.7	15.6	11.7	8.5	10.7	8.2	3.8	4.5
IOW	3.5	8.0	1.7	1.8	1.8	1.9	1.4	1.6
RAM	57.7	37.6	13.8	12.3	12.5	12.0	7.0	9.1
TLBM	6.5	05.3	3.9	3.3	3.9	3.3	1.7	3.1
PF	3.7	6.9	4.5	4.8	4.6	4.9	2.7	4.4
LLCM	8.8	12.2	6.4	7.9	6.4	7.6	4.4	7.9
FLCM	8.1	08.2	5.3	5.3	4.9	4.8	3.1	4.7

Table 5.2: Mean Absolute Percentage Error for each metric and model, evaluated in the test set

Let us visually assess the behaviour of some of the evaluated models in some examples from the test set. We have selected and plotted the resource usage and the predictions of three examples. Figures 5.2, 5.3 and 5.4 show three different pairs of co-located applications with different properties. Each figure is composed of three columns. The first two columns show the resources usage trace in the isolated runs. The third column shows the trace of the co-located applications with the predictions made by the MLP and the sequence to sequence model. The shaded region shown in the third column displays the period at which both applications run at the same time. This is the period used to compute the error metrics (when both applications are running at the same time). Moreover, vertical discontinuous lines mark the time step at which the input $[\boldsymbol{a}; \boldsymbol{b}]$ finishes. In the interest of clarity, the figures do not contain the predictions of the other models in order to avoid excessively cluttered images.

When applications have low resource demands, and especially when they compete for resources only during small periods of time, the expected demands are easy to predict. Figure 5.2 shows a pair of low-demand applications competing for resources during only a small period of time. In this example the second application starts roughly at time step

20 while the first one finishes around time step 25. This is a simple case where all models can capture the increase in the metrics (specially CPU usage) during the brief period in which both applications run at the same time. Notice that the vertical discontinuous line is around time step 48. This means that the applications took around the same time when they were run in isolation than when they are run co-scheduled. In contrast, in cases where more demanding applications are co-located, the expected resource usage of the co-located applications over time is not straightforward to predict. One of the main difficulties in this type of scenarios is the slowdown of both applications while competing for resources, which usually implies a big difference in execution time with respect to the applications being run in isolation. This can be seen in Figs. 5.3 and 5.4. In Fig. 5.3 the input jobs take around 40 time steps to execute in isolation, but require more than 80 under the presented co-schedule. The same effect, even more pronounced, can be seen in Figure 5.4. The degradation of quality for long sequence prediction is a known issue found in [73].

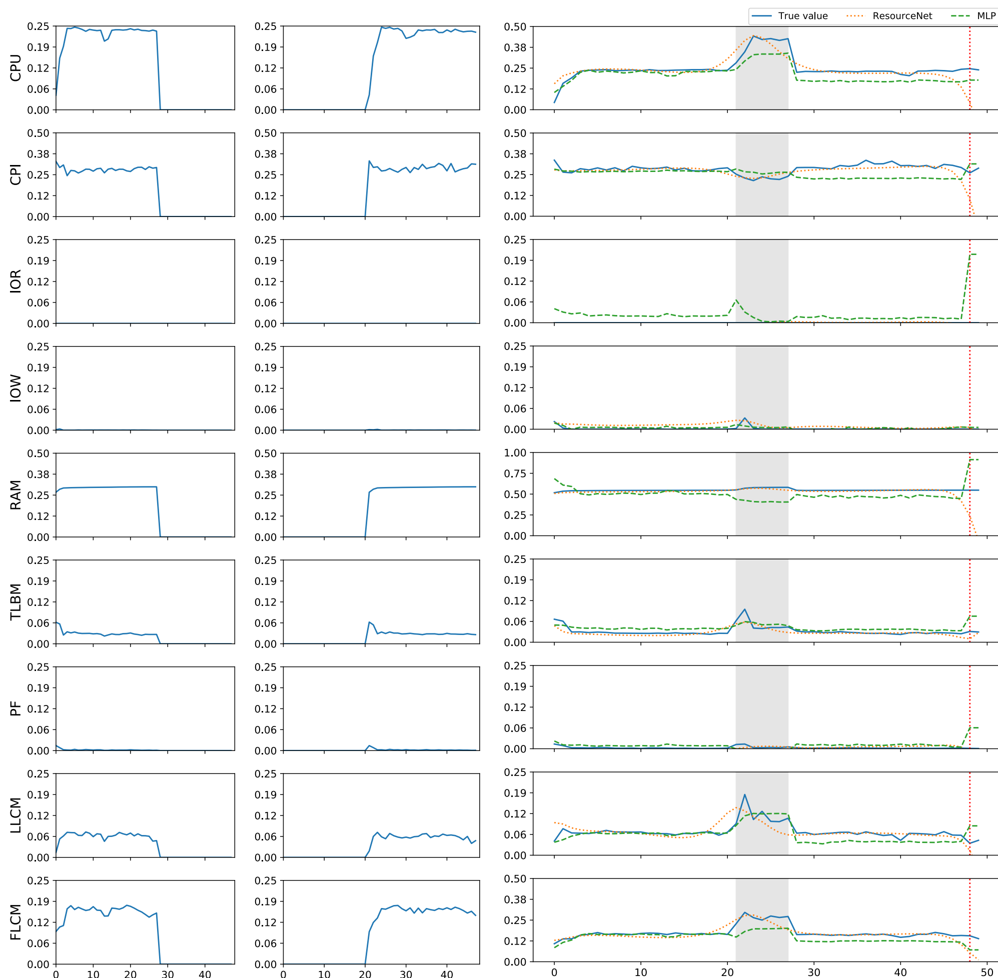


Figure 5.2: Example with two co-located applications that share resources only during a fraction of the execution.

In the cases presented, ResourceNet is capable of capturing the trend of the resources correctly; not only in the shadowed region but throughout the entire sequence. For example, the CPU usage in Fig. 5.3 starts at around 75% of usage, then decreases at around 50 and ends again at around 75% of CPU. In this figure, one may clearly observe the undesired drawback of the element-wise models previously explained. Around time step 38, we can see the predictions of the MLP getting stuck at a particular value. This is easy to observe for IOR, RAM and CPU usage, where predictions are far from the true values. Such behavior is caused due to the lack of input features in the isolated traces, so the model has to rely on making predictions out of zero-padded feature values because either $\boldsymbol{a}$ or $\boldsymbol{b}$ or both have finished. Notice how the MLP makes high error predictions for CPU and IOR when the input traces finish (marked by the vertical red line). Nevertheless the resource usage trend is predicted semi-accurately while both applications run. The sequence-to-sequence model is capable of generating and output sequence longer than the input sequence in a "natural" way, without any need to pad the original features.

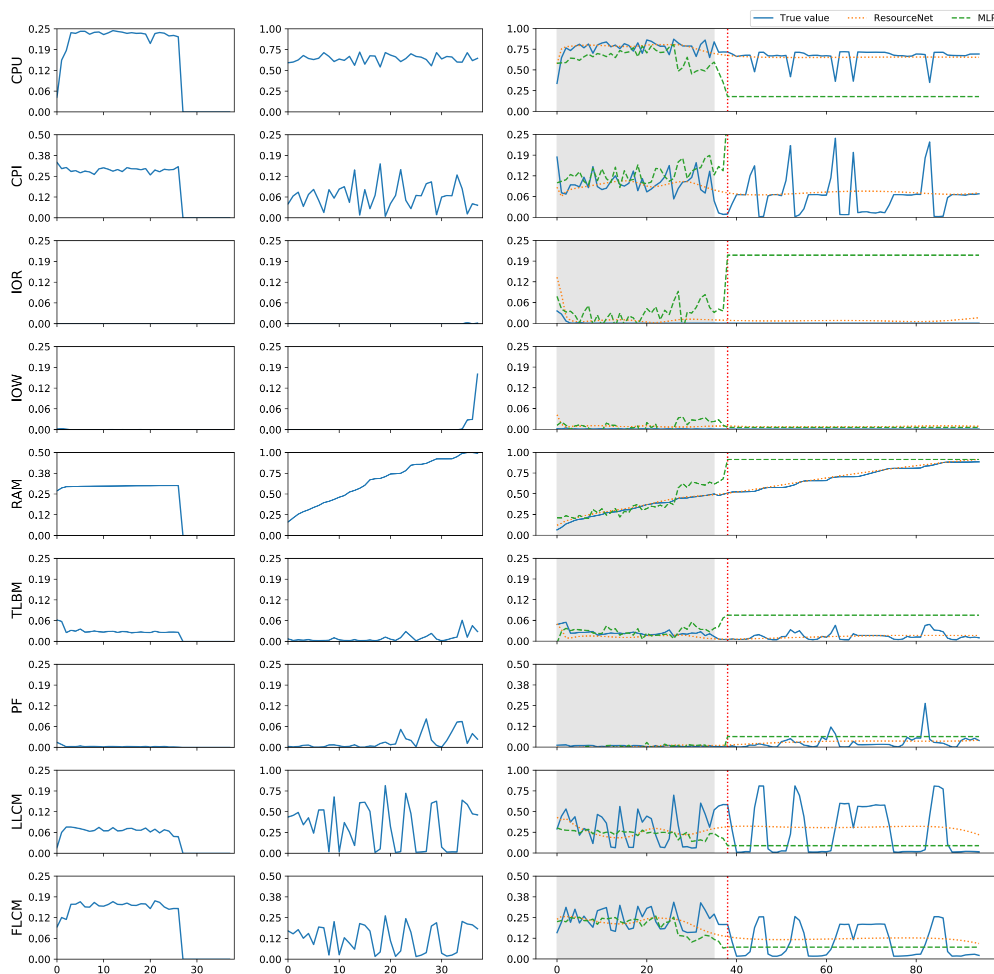


Figure 5.3: Example where one application demands almost all memory available.

Figure 5.4 shows two jobs which require a high amount of RAM. The first requires low CPU and high IOR, while the second is just the opposite. We can see that once the first job has finished the IOR demand in the co-located trace goes to zero and our model can successfully detect this trend. Even though models capture the trends of all features reasonably well, none of them are able to predict the burstiness of some features, especially LLCM and FLCM. Note that the execution time of the concurrent applications is roughly five times more the amount they need while running in isolation. This poses a challenge since the Sequence-to-Sequence model needs a decision criteria to stop producing the output trace. The image simply shows the trace until the point where the true sequence ends. The experiments in the following section precisely study different approaches to provide an stopping criteria to the model.

Figure 5.4: Example where both applications use all available memory at some point during the execution.

5.3.4 *Experiment 2: Estimating co-scheduled execution time*

Stopping criteria of the co-scheduled trace

We have experimented with different criteria to predict the runtime of co-scheduled applications. Our methods are based on PC and EoS features which are detailed in Section 5.2.2. Using these features, we can test different criteria to predict the completion time of a co-scheduled pair of jobs.

Since we do not know in advance how long co-scheduled applications can take to finish, we have decided to treat the length of the generated trace as a hyperparameter to be adjusted. The different criteria tested are as follows:

- *first max EOS*: applications are predicted to finish when the first local maximum is found in EOS.
- *first max PC sum*: applications are predicted to finishwhen the first local maximum is found in the sum of PC_a and PC_b.
- *argmax EOS*: applications are predicted to finish at the argmax of the sum of EOS.
- *argmax PC sum*: applications are predicted to finish at the argmax of the sum of PC_a and PC_b.

For each of the criteria, Table 5.3 reports the MAPE error and the standard deviation depending on the length of the generated trace. The first column *length* contains at position kx the expected error when the co-scheduled trace is generated up to k times x time-steps, where x is the longest of the input traces when executed in an isolated environment. Results show that if the length of the generated trace is too short (for example $1x$), then errors are large because the stopping criteria is fired before the co-scheduled applications might finish. Errors decrease as k increases up to a point where the errors start to increase. The best results are achieved at $k = 4$. Our experiments show much lower errors for the criteria based on PC features than for criteria based on the EoS feature.

The errors made by our criteria behave differently in accordance with the length of time we predict applications will need to finish with respect to the runs made in isolation. In Figure 5.5 one may observe the distribution of errors made by our criteria, grouped according to where we predict the execution will end. The plot shows how the farther away the co-located applications are predicted to finish the larger the expected error. The box plot shows results for the criterion *argmax PC sum*. We can see how applications that are expected to run between $1x$ and $2x$ have low errors, while applications that are expected to finish between $3x$ and $4x$ have higher errors and more variance. This increase on error and

variance can be supported by the behaviour of the predicted PC_a and PC_b features found in Fig 5.6. In the figure we can observe how the dotted lines that predict the end of the applications predict very accurately the end of the execution on the first column but do not work as well on the third column. This behaviour of loosing quality for longer sequences is consistent with other works such as [73].

Notice the capability of knowing when application finish can be used as a rule for deciding when applications should not be co-scheduled. A simple rule for avoiding co-scheduling applications could be to forbid scenarios in which the model runtime prediction falls within $3x$ or $4x$.

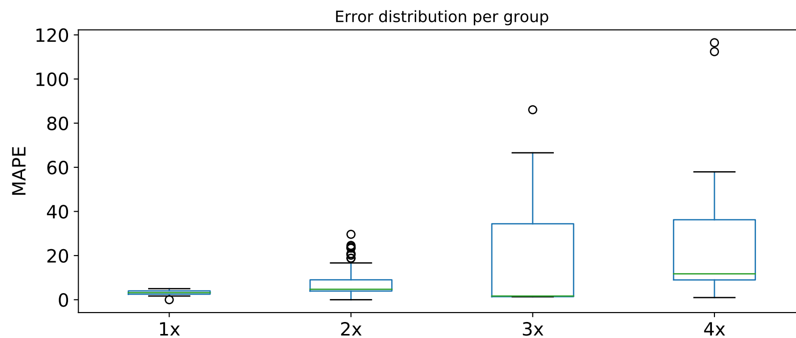


Figure 5.5: This figure shows the distribution of the MAPE grouped into 4 categories.

	first max EOS		first max PC sum		argmax EOS		argmax PC sum	
trace length	MAPE	std	MAPE	std	MAPE	std	MAPE	std
1x	96.6	5.1	98.3	8.1	77.5	21.3	59.0	17.7
2x	96.6	5.1	77.7	39.1	49.1	35.9	25.1	20.4
3x	96.6	5.1	46.2	45.2	18.8	24.7	14.3	15.6
4x	96.6	5.1	29.4	37.8	30.8	35.9	12.7	18.2
5x	96.6	5.1	29.4	37.8	31.5	45.6	28.2	39.0
6x	96.6	5.1	29.4	37.8	43.8	64.5	30.0	44.1
7x	96.6	5.1	28.1	37.0	48.8	75.1	106.9	96.9
8x	96.6	5.1	28.1	37.0	63.6	97.2	107.6	98.6

Table 5.3: MAPE errors made using the different stopping criteria.

Looking at EoS and PC feature predictions

Figure 5.6 shows the EoS and PC features behaviour for three different co-scheduled workload pairs. Every column in the figure corresponds to a different colocation of two workloads. The continuous lines show the true values of the different features and the dotted traces show the predictions made by the sequence to sequence model. Vertical lines show the length of the longest workload executed in isolation. Plots in this figure show predictions generated up to five times the length of the longest input sequence. The CPU resource usage of the co-scheduled pair has been added into the plot to contextualize the other features. Notice that the longer a pair takes to execute, the less a signal is shown by the EoS feature. Nevertheless, PC features maintain a relatively good quality even for co-scheduled pairs with high resource competition.

Our results suggest that criteria build to predict the length of a co-scheduled trace using EoS should not be based on the actual value of the EoS feature, but rather on a function taking into account the maximum value achieved during the prediction process. Notice that, in the first case, the predicted EoS arrives roughly at 0.75. However, the second case barely takes value 0.25 and the third case the EoS scarcely barely goes above zero. This behavior increases the difficulty of predicting runtime even more with this feature.

A reasonable explanation for the behavior of EoS is that our model is trained by minimizing the mean squared error between predictions and true traces. It is logical to argue that the optimization procedure induces the network to focus more on predicting PC features better than EoS, since the EoS feature takes value 1 at only 1 time step while PC features increase at every time step. The overall contribution to the error of always predicting 0 in EoS is therefore much lower than always predicting 0 in PC_a or PC_b.

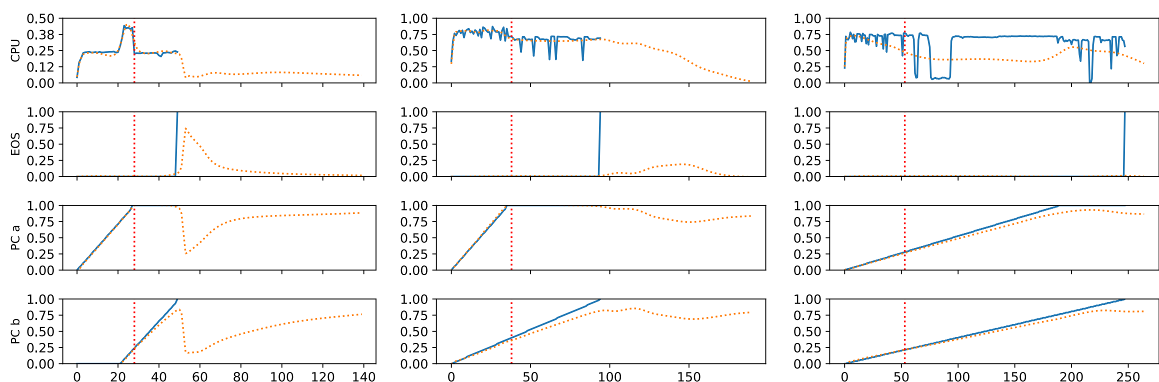


Figure 5.6: Behaviour of EoS and PC features for the workloads seen in Figures [5.2] [5.3] and [5.4].

5.4 CONCLUSIONS

This chapter introduces the use of recurrent neural networks for interference and resource prediction tasks of co-scheduled applications. We adapt a sequence-to-sequence model to work with two input sequences instead of one, thereby providing a padding containing the start time of one application with respect to the other. Our method predicts resource usage of the monitored metrics over time and is adaptable enough to be trained with workloads of arbitrary input and output lengths. Moreover, since training is done for a regression task instead of a classification task, the standard stopping criteria used in similar architectures does not provide good results. Therefore, we introduce the percentage completion features which significantly improves completion time prediction of co-scheduled applications.

Our experiments show that the model is able to predict the resource usage of co-located applications sharing resources over time, even when application performance degrades drastically due to a high demand of similar resources at the same time. Moreover, as a baseline, we compare our model with standard machine learning algorithms. The experiments show that our model makes more accurate predictions and is able to deal with the sequential nature of the data, thus making it suitable for the presented scenario where input/output pairs might have different lengths.

6

FAST TIME-TO-SOLUTION BIG DATA WORKLOAD AUTO-TUNING WITH REUSABLE PERFORMANCE MODELS

This chapter describes the third contribution of this dissertation which aims to provide a fast time-to-solution method to auto-tune Big Data workloads. Big Data Framekworks, such as Hadoop or Spark provide a high level language for building distributed applications that can run across multiple machines. These frameworks expose many tunable parameters to programmers. Nevertheless, tuning jobs is not a trivial task and requires in depth knowledge of the application to be executed and the impact of the parameters on the workload to be optimized. To facilitate this task many practitioners use auto-tunning systems that free them from the burden of having to manually adjust many parameters. State-of-the-art auto-tuning systems tune configurations by iteratively running jobs with different configurations, and smartly choosing the samples to quick find good candidates. Many optimizers enhance the time-to-solution using black-box optimization processes that do not take any information from the workloads.

The work presented in this chapter finds configurations that provide a good performance from one or two runs of the workload. To achieve this, we mine the log file generated by the Big Data framework once an application is executed. Since the log file contains a large amount of information from the application, we can build a model that is trained on low-level features that summarize relevant information from the application. We encode all the gathered information from the log into a feature vector that is provided to machine learning models. This process provides the learning algorithms an application specific feature vector with information of the workload to be optimized. This allows our system to predict sensible configurations for unseen jobs, given that it has been trained with reasonable coverage of applications. Experiments show that the presented system correctly produces good configurations achieving up to 80% speed-up with respect to the default configuration, and up to 12x speed-up in search time when compared to a standard Bayesian Optimization. We evaluate our system with standard benchmark applications for Big Data frameworks, using Spark, which is one of the most popular frameworks for distributed computing. This log file contains a large amount of information from the executed application. We use this

information to enhance a performance model with low-level features from the workload to be optimized. These features include Spark Actions, Transformations, and Task metrics. This process allows us to obtain application-specific workload information. With this information our system can predict sensible Spark configurations for unseen jobs, given that it has been trained with reasonable coverage of Spark applications.

Section 6.1 presents an overview of the problem and a summary of the contributions. Section 6.2 presents our proposed solution to build feature vectors from the log file and a search mechanism based on performance models which have been trained using as input those feature vectors. Section 6.3 shows experiments that evaluate our work. Finally, Section 6.4 presents the conclusions of the chapter.

6.1 INTRODUCTION

Big data processing frameworks, such as Apache Spark, are being increasingly utilized in a wide range of industries. To support the diverse range of applications, such frameworks allow users to tune parameters that might significantly affect the performance of the workload. There are many parameters to be tuned in all Big Data frameworks. Spark, for example, has more than 100 parameters that can be configured, which makes the selection a difficult task.

There are different strategies for overcoming the problem of selecting good parameters for a specific job. Several methods have been proposed for auto-tuning hyper-parameters of configurable software, many of which are based on machine learning models. Such models can be used to provide estimated performance metrics of the application under a particular configuration of the hyper-parameters. The best parameters found during the search are then provided to the user.

Current state-of-the-art work based on model search techniques (model-based systems) have the drawback that retraining is required in order to generalize to unseen workloads [9]. This is a consequence of current models only receiving information from the configuration space that they need to optimize. Even works that provide an application identifier or the dataset size as an input to the learning algorithm need retraining for new applications. This limitation has led the community to explore how to improve current performance models and how to build solutions that do not need machine learning models (non-model-based systems). Some systems [23] without performance models have been built based on an efficient search mechanism of the configuration space. This approach can achieve a relatively fast time-to-solution, while maintaining good quality in the results.

The claim that some model-based systems are slower than non-model-based systems is reasonable. Auto-tuning systems that do not use a performance model do not need to spend time training an oracle when a new workload needs to be optimized. Therefore, it is

possible that such systems can find a solution with fewer job executions than the number of runs required to train the oracle in a model-based system. The objective of our work is to improve the current model-based approaches in such a way that very few samples are needed to find a good configuration. In the most extreme case, we would like to perform model-based optimization with a single run. Nevertheless, to achieve this goal we need a way to transfer previous knowledge about past examples to optimize new workloads.

Recent work from the area of configurable software [37] explores the idea of using transfer learning to facilitate the task of extracting measurements from a system in order to build a dataset. In our scenario, the data consist of many application executions, and we can assume that cloud providers already possess a large number of execution logs from those applications. Therefore, we focus our method on improving the time-to-solution of finding good configurations for applications, instead of focusing on decreasing the number of executions that have to be performed before good performance models are created. Nevertheless, we adapt our methodology with the goal of using transferable knowledge across workloads, providing a information to the oracles that is transferable across applications.

Model-based systems have to deal with three problems: sampling the search space to build a dataset, learning a model that can predict performance metrics accurately on new data, and searching in the configuration space to find a suitable configuration for a new workload. Our work focuses on improving the last two components in the context of workload auto-tuning. We want to build better features to achieve higher quality models that can more accurately predict performance metrics for new applications. Moreover, we want to speed up the time-to-solution of the whole process for a new application that the learning model has not seen in the training data.

To address the above challenges, we present a system that builds upon a model-based search technique modifying a standard Bayesian Optimization [70] process. Our system is enhanced with a mechanism that transfers knowledge from the execution logs to provide more information to the Machine Learning model. The key part of the proposed approach is that, in order to make the model generalize to unseen applications, we create a rich feature descriptor from the internal log file of events generated once an application is executed. This descriptor contains a summary of the tasks performed by the workload, providing the oracle with low-level details about an application, without the need for extra instrumenting of the computational environment. Using this information, the model can learn the relationships between the parameters to be optimized and the application descriptor. This allows our model to generalize better to new unseen workloads without retraining. Moreover, our method does not need input from the user with respect to the type of application executed. In addition, since the feature descriptor is built from the log file, our methodology respects

code privacy because it does not need to access the source code of the application to be optimized.

Using the proposed feature descriptor, we can simulate a Bayesian Optimization process guided by an oracle that makes predictions conditioned on the application to be optimized. In this way, we can avoid costly runs of the application, which allows our methodology to achieve a faster time-to-solution with respect to a standard Bayesian Optimization process, while achieving almost the same quality.

We evaluated our approach using Spark. Our dataset contains a wide range of popular workloads from two popular Spark benchmarks suites: Hi-Bench [32] (from Intel) and Spark-perf (from DataBricks). Our experiments show that the proposed feature descriptor enhances the quality of the machine learning models with respect to other features used in the literature. Moreover, the cost of finding a good configuration is considerably reduced because our methodology can perform a reliable search in the configuration space adapted to the type of application to be optimized.

CONTRIBUTIONS OF THE CHAPTER

The main contributions presented in this chapter can be summarized as follows:

- We propose a novel feature descriptor created from the log file of an application to provide extra knowledge to the performance models used for auto-tuning. We show that this extra information improves the quality of the predictions of machine learning models for unseen workloads by up to 34% with respect to the same models trained on standard feature representations.

- We develop a practical strategy that can auto-tune configurations for new, never seen, workloads. Our solution can achieve up to 80% speedup on the execution with respect to the default configuration. We show that our method can achieve up to 12x speedup compared to a standard Bayesian Optimization process, with almost equal results.

6.2 METHODOLOGY

Problem formulation and motivation

Let S be the Configuration search space containing the parameters tunable parameters to be optimized. Let $s \in \mathcal{S}$ be a particular configuration of hyper-parameters in the search space. Let $e : \mathcal{S} \longrightarrow \mathbb{R}$ be the evaluation metric we aim to minimize, for example the execution time of a workload. The problem of finding a good Spark configuration has been framed by some studies [37] as finding s^* defined as follows:

$$s^* = \arg\min_{s \in S} e(s) \tag{6.1}$$

The formulation of the problem in equation (6.1) states that the optimization to be performed only depends on $s \in \mathcal{S}$. Some works, such as [92], define the problem with the following e function: $e : \mathcal{S} \times \mathcal{X} \longrightarrow \mathbb{R}$, where $\mathcal{X}$ is a space of features that might capture relevant information about the problem. We will use an evaluation metric of this form.

Given a vector of features $x \in \mathcal{X}$ the optimization problem becomes:

$$s^* = \arg\min_{s \in S} e(s; x) \tag{6.2}$$

Note that in equation (6.2) $s \in \mathcal{S}$ can change but $x \in \mathcal{X}$ is fixed. In some works, $\mathcal{X}$ is a single variable containing the dataset size [55, 92]. This implies that the only information expected to affect the performance of an application is the dataset size. Since applications can be very different, this becomes an oversimplification of the problem, particularly in environments that use heterogeneous workloads. To improve this feature vector, other works add an identifier of the application [9], providing more contextual information to the optimization problem. Nevertheless, it is difficult to reuse and extrapolate from this contextual information, since it is an identifier that is usually encoded as a categorical variable.

Popular methodologies for the problem

A standard, general-purpose procedure for finding good configurations for workloads consists of using a sequential optimization procedure that learns over time which areas of the search space are likely to contain good solutions. A popular algorithm that follows this logic is Bayesian Optimization [70], which uses a Gaussian process to decide which of the regions of the search space are likely to contain a probable gain in performance. This methodology has been tested for searching workload configurations [37] with great success.

First, a time budget T and a search space $\mathcal{S}$ are provided to the optimization procedure. Then, finding a configuration corresponds to updating the black box optimizer with the execution time $y = e(s)$, as shown in Fig. 6.1. This process generates vectors from the search space $\mathcal{S}$, while storing the best configuration found so far. Once the time T expires, the optimization process finishes and the best configuration is returned. The main drawback of this approach is that a job execution is required for each update of the optimizer. Nevertheless, Bayesian Optimization techniques [36] have been used successfully to tune software. Since executions might take a long time, this technique might not be practical in some scenarios. To improve upon this solution, some works such as [23] focus on identifying a subspace of $\mathcal{S}$ relevant to the problem. They then focus on optimizing the subset of relevant parameters, considerably reducing the number of samples needed, and achieve good results.

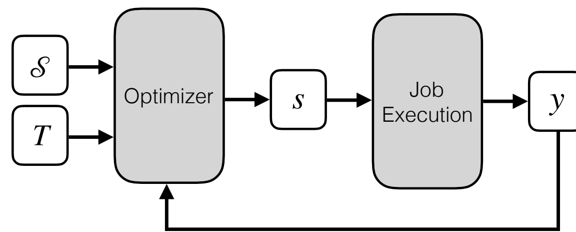


Figure 6.1: Classical optimization pipeline followed by a Bayesian Optimization process.

In order to avoid running an application many times to find a good configuration, many works use a machine learning model trained to predict the evaluation metric from past executions [5, 55]. This model, sometimes called the performance model, aims to replace many of the job execution calls with the predictions provided by the model. The diagram in Figure 6.2 represents the overall pipeline for the optimization process. Note that this process has 2 different phases, which depend on whether an application is known or not. Let A be the set of applications that the system has observed. If the identifier of the app, app_{id}, is in A, then the optimization procedure is fast and the oracle can be used to make the predictions $\hat{y}$ that are provided to the search algorithm. Then the best prediction according to the model is retrieved and executed. If the application is not in A, the system enter into a sandboxing phase, where many samples have to be generated and executed, and the oracle needs to be retrained. This is necessary because the only information that the tuning system has is the application identifier.

The question as to which optimization pipeline is better is still open. Works like [23] suggest that by smartly selecting a subset of the configuration space, good results can be achieved without any need of a performance model.

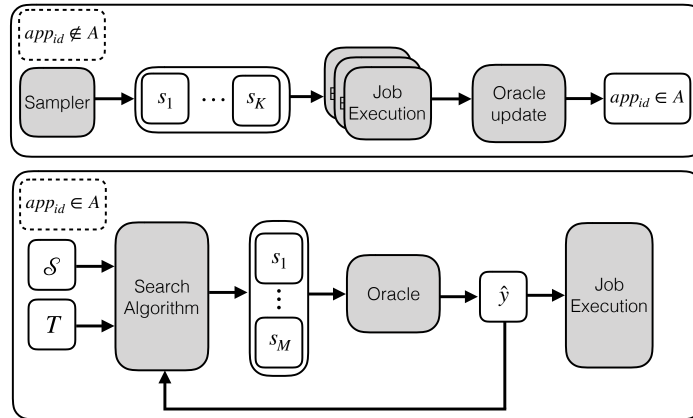


Figure 6.2: Classical optimization pipeline with a performance model (Oracle).

Overview of our proposed solution

The drawback of the optimization pipelines in Figures 6.1 and 6.2 is that the whole process needs to execute the workload multipe times. In practise, the solutions that use a performance model need advance knowledge of the application in order to make sensible predictions. If an application is not known, several job executions are required to update the oracle. Recent work [9] has studied methods to reduce the number of samples required to update the oracle to around 100 executions, which is still a high number of executions that we would like to decrease. Since the selection of the parameters depends on the quality of the model, improving the model predictions might boost any model-based search results. In our solution, the oracle is a machine learning regressor that predicts the application execution time based on a feature vector that characterizes its behavior.

In order to mitigate the main drawbacks of the previous methodologies, our work focuses on improving two paramount pieces of model-based auto-tuning systems. First, in order to improve the generalization of the machine learning models, we built a feature descriptor that characterizes applications without any specific input from the user of the application. This vector is generated by extracting information from the log file of the application, which is synthesized in a feature vector $\boldsymbol{x}$. This part of the optimization pipeline can be seen in the upper part of the diagram in Figure 6.3.

Note that the job is only executed once to extract the workload characteristics. Then, the workload characterization $\boldsymbol{x}$ is created and passed to the orable which is used in the optimization process. It should be noted that, before the oracle is queried, the job is executed (with the default configuration) and $\boldsymbol{x}$ is extracted. This vector is then used to condition the output predictions of the oracle. In subsection 6.2.1, we describe in detail how $\boldsymbol{x}$ is constructed. It is relevant to emphasize that this feature vector contains information from the

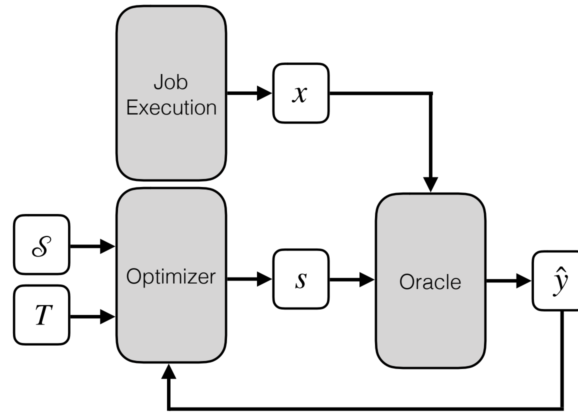


Figure 6.3: Proposed search pipeline for auto-tunning Big Data workloads.

application that can potentially be transferred to new applications because new workloads might be implemented using a similar type of execution graph, which will yield a similar $\boldsymbol{x}$.

The optimization process then follows a simple Bayesian Optimization procedure where, instead of running the application, we query the oracle, providing information about the application with the feature vector $\boldsymbol{x}$.

6.2.1 *Feature descriptors from the log file*

A log file is a text file containing information about the application that is executed, we are interested in gathering information from this source. The literature contains many types of descriptors to summarize raw text. One of the most popular approaches to summarize a raw string consist on creating a term frequency vector. This vector contains at each coordinate the number of times a particular word in the raw string is found. Note that, in order to extract "words", a pre-process needs to be performed to partition the raw input string into "words". This process is called tokenization.

Once all raw strings are tokenized, a vocabulary can be created containing the set of all possible words (or tokens). This vocabulary can be used to create a bijection that maps each possible token to a different natural number. Once this mapping is constructed then any raw string from a log file can be mapped into a term frequency vector that summarizes the log file. In the event that a new token is not in the vocabulary, this feature representation simply ignores the token.

Feature descriptor for Spark workloads

In this work we will use Spark for all our experiments, therefore, the vocabulary that we will use is the set of unique tokens that are found in the Spark logs of our dataset. Since the

log files from Spark, called SparkEventLogs, are structured as a list of key-value pairs in the form `"key": "value"` we will build a slight modification with respect to the standard term frequency vectors to gather information about the workloads. Each element in the list of dictionaries starts with the `Event` key that we used to decide whether the row needs to be parsed or skipped. Therefore, the tokenization process in this type of raw string is already done for us. A reasonable vocabulary is the set of keys found in the input string.

In order to build our feature vector we parse three different types of key-value pairs. We take the SparkEventLog elements with key equal to "Event", since they contain the most relevant information about the workload. Such "Event" keys contain as values strings of the form `SparkListener + p + e` where $p \in \{\texttt{Application}, \texttt{Stage}, \texttt{Task}\}$ and $e \in \{Start, End\}$. As the names suggest, `Event` strings are initiated with the suffix `Start` and finished with `End`, marking the beginning and end of the `Event`, respectively. The most interesting features are retrieved from `Stage` and `Task` events because they provide us information about the execution graph of the application. We only use `Application` keys for verifying if a job starts and finishes without errors.

The resulting descriptor contains a total of 75 features. From those features, 11 are retrieved from Stage events and 64 are retrieved from Task events. We denote by ef (eventlog features) this feature vector of 75 components containing Task and Stage features. A detailed description of the features is presented below.

Stage Features

`Stage` events provide information about the Spark actions and transformations used in the Stage. This type of event provides high-level information about the workload because it shows core Spark function calls made during the execution of the workload. Some examples of `Stage` events include the well-known "map", "reduce" and "reduce by key" operations typically used to leverage distributed computing systems. This information tells us the percentage of time an execution spends on each of the actions and transformations found in the workload. The descriptor contains the following actions and transformations from the following list: ["collect", "count", "countByKey", "first", "flatMap", "map", "reduce", "reduceByKey", "runJob", "takeSample","treeAggregate"].

Note that a workload might only use a subset of the features presented.

Task Features

`Task` events provide information about the different Tasks found in a Stage. Tasks provide us with low-level metrics such as the number of bytes read and written to disk, the time spent during garbage collection and the CPU time. We aggregate the information across

all tasks into a vector of aggregated statistics. This information tells us whether, overall, the execution was read or write-intensive, CPU intensive, etc. Table 6.1 lists all the features gathered from Task events.

	Task features
0	Input Metrics: Bytes Read
1	Executor Deserialize Time
2	Executor Deserialize CPU Time
3	Executor Run Time
4	Executor CPU Time
5	Result Size
6	JVM GC Time
7	Result Serialization Time
8	Memory Bytes Spilled
9	Disk Bytes Spilled
10	Shuffle Read Metrics: Remote Blocks Fetched
11	Shuffle Read Metrics: Local Blocks Fetched
12	Shuffle Read Metrics: Fetch Wait Time
13	Shuffle Read Metrics: Remote Bytes Read
14	Shuffle Read Metrics: Remote Bytes Read To Disk
15	Shuffle Read Metrics: Local Bytes Read
16	Shuffle Read Metrics: Total Records Read

Table 6.1: Task features.

Since there can be an arbitrary number of Tasks in an SparkEventLog, we generate four feature vectors containing the mean, minimum, maximum and standard deviation across all the features in Table 6.1. The final descriptor summarizes the behavior over multiple tasks and is created by concatenating the previous four feature vectors. This process creates a $64 = 16 \cdot 4$ dimensional feature vector. Finally, we add a feature "dataset size" that contains the total data size read from disk. This feature is extracted by summing the values in "Input Metrics: Bytes Read" across tasks.

SparkEventLog example

The example given in this subsection shows different parts of the SparkEventLog created when executing a job. Our goal is to show how the source text of the file corresponds with the previously defined Stage and Task features. To make the explanation clearer, some parts of the SparkEventLog have been omitted. The snippets of text shown below correspond to the SparkEventLog generated executing an Fp-growth algorithm. This is a popular data-mining application that generates association rules.

- `ApplicationStart` and `ApplicationEnd` mark the start and end of the application. In this example, we can see that the `AppID` is `local-1561554220820`. This string is precisely the name given to the SparkEventLog generated when running this application. If an application does not crash, it will have a `SparkListenerApplicationEnd` event at the end of the list.


```
{"Event":"SparkListenerApplicationStart",
"App Name":"05_fpgrowth_itemset_mining.py",
"App ID":"local-1561554220820","Timestamp":1561554312331,"User":"dbuchaca"}
```


- `StageSubmitted` and `StageCompleted` denote the start and end of a Spark Stage, respectively. In the example below, we can see the field `Stage Name`, which takes the value `count`, one of the features from Table ??. The word `count` is a Spark keyword that describes the type of action or transformation being performed in the stage (in this case, Stage 0).


```
{"Event":"SparkListenerStageSubmitted
"Stage Info":{"Stage ID":0,"Stage Attempt ID":0,
"Stage Name":"count at FPGrowth.scala:217","Number of Tasks":2,
"RDD Info":[{"RDD ID":4,"Name":"MapPartitionsRDD",
Scope":"{\"id\":\"2\",\"name\":\"map\"}",,... }]
```


- `TaskStart` and `TaskEnd` denote the start and end of the task, respectively. In the snippet provided, we can see several metrics that tell us the read and write statistics, garbage collection metrics, etc. The information found here can be used to fill in the features in Table 6.1.


```
["Event":"SparkListenerTaskEnd","Stage ID":0,"Stage Attempt ID":0,
"Task Type":"ResultTask", "Task End Reason":{"Reason":"Success"},...
"Task Metrics":{"Executor Deserialize Time":33,
"Executor Deserialize CPU Time":19157489,
"Executor Run Time":647,"Executor CPU Time":147466180,"Result Size":1569,
"JVM GC Time":0,"Result Serialization Time":1,"Memory Bytes Spilled":0,
"Disk Bytes Spilled":0,"Shuffle Read Metrics":{"Remote Blocks Fetched":0,
"Local Blocks Fetched":0,"Fetch Wait Time":0,"Remote Bytes Read":0,
"Remote Bytes Read To Disk":0,"Local Bytes Read":0,"Total Records Read":0},
"Shuffle Write Metrics":{"Shuffle Bytes Written":0,"Shuffle Write Time":0,
"Shuffle Records Written":0},"Input Metrics":{"Bytes Read":125875,
"Records Read":304}, "Output Metrics":{"Bytes Written":0,"Records Written":0},
"Updated Blocks":[]}}]
```


6.2.2 *Simulated Bayesian Optimization*

Using an optimization procedure that requires executing jobs to evaluate configurations can be a slow and expensive process. In order to avoid executing a program many times with different configurations, our work proposes performing search. Instead of performing the optimization process in the true configuration space of hyper-parameter configurations, we perform the process in an alternative space that is modelled by an oracle that is provided with the characterization of the workload $\boldsymbol{x}$. Therefore, the optimization we propose uses estimated workload times instead of running the workload. The inputs to the Simulated Bayesian Optimization (SBO) procedure detailed in Algorithm 1 are:

- h: Machine learning model already trained with access to a method *h.predict* that can be used to predict the execution time from an input vector (s, x). Here s is a vector containing a particular configuration of hyper-parameter values and x is a vector of features that characterizes the workload.

- Opt: Class of optimization procedure used to instantiate a Bayesian Optimization opt with the Expected Improvement as acquisition function. Here, opt has access the two methods. The method $opt.generate()$ creates a configuration s. The method $opt.update(s, \hat{y})$ updates the internal Bayesian Optimization process for the configuration s with value $\hat{y}$.

- *job*: This is the workload that SBO needs to optimize. Note that we assume we have access to a function $run(job; s)$ that runs the workload and returns the execution time and the log file of the workload.

- R: Maximum number of runs allowed to our method.

- $\mathcal{S}$: Search space definition. We used it to provide boundaries for $s \in \mathcal{S}$.

- s_0: Default configuration of the hyper-parameters.

- T: Maximum Time allowed for the optimization process. Note that even if the total number of runs does not reach the maximum R, the optimization process can be stopped if it reaches a maximum time T.

- ϕ: Function that generates a feature descriptor for the current log file of the application, executed with the default configuration of hyper-parameters. In our experiments we use the descriptor described in Section 6.2.1.

Algorithm 1 is a simplified version of the SBO procedure which keeps only the best configuration in memory. Once the algorithm finishes, it returns the estimated best configuration s^*. To ensure that s^* is a good choice, we need to run the workload with the provided configuration. Therefore, a reasonable way to implement this algorithm involves storing the best K values found during the optimization process. Then the process would return $s_1^*, \ldots, s_K^*$, which should be run to find which is the best configuration. We refer to this process as SBO-K. If a single configuration is returned and executed, we would say that it followed an SBO-1 optimization procedure. In the event that SBO-K returns a solution that is worse than the default configuration, the application should be flagged and the model h should be updated with training samples of that application. In that case the default configuration should be returned.

Algorithm 1

```
1: procedure SBO(h, Opt, job, R, S, s_0, T, φ)
2:     t ← 0
3:     opt ← Opt(S)                                  ▷ Optimizer is created
4:     y_0, LogFile ← run(job; s_0)
5:     y* ← y_0
6:     s* ← s_0
7:     x ← φ(LogFile)
8:     opt.update(s_0, y_o)                          ▷ Optimizer is updated with s_0
9:     for r ∈ 1...R do
10:        s ← opt.generate()
11:        z ← (s, x)
12:        ŷ ← h.predict(z)
13:        opt.update(s, ŷ)                          ▷ Optimizer is updated with s
14:        if y < y* then
15:            y* ← y
16:            s* ← s
17:        end if
18:        t ← t + get_time()
19:        if t >= T then
20:            break
21:        end if
22:    end for
23:    return s*
24: end procedure
```

The previous explanation describes SBO in general terms. The following Section 6.3 includes a detailed description of our selection process for the different inputs involved in the SBO algorithm, as well as the applications used to evaluate our methodology. In particular, the election of regressor h (as well as the tuning of the model) is described in Section 6.3.1. The search space S, the Default configuration s_0 and the maximum number of executions allowed R can be found in Section 6.3.2. We did not specify a maximum time budget T.

6.3 EXPERIMENTS

We evaluated the effectiveness of our proposed solution by comparing the results with popular state-of-the-art approaches. In particular, we aimed to answer the following research questions:

(i) **Feature descriptor Quality**: Do the proposed features help to improve predictions based on machine learning models? Does an oracle trained with the SparkEventLog features predict sensible configurations for unseen Spark workloads?

(ii) **Time-to-solution speedup**: Does the overall methodology provide a good configuration when compared with a solution found with a well-known optimization process, such as a Bayesian Optimizer? Does the process provide a good configuration in less time?

The answer to (i) was obtained by comparing our results with other feature representations found in similar works [5, 9, 47, 55]. For (ii), we needed to measure the quality of the results and the time-to-solution of our approach with respect to a Bayesian Optimization process.

Dataset Description

To evaluate our system we decided to use a set of workloads that cover different popular uses of Spark. The set of workloads we used is based on two benchmarks. First, we included workloads found in HiBench [32], a popular benchmark for Spark, which includes micro benchmarks, as well as workloads from machine learning, data mining, natural language processing and web-search. Second, we also included benchmarks from Spark-perf [72], a benchmark created by DataBricks, the company founded by the original creators of Spark. Table 6.2 shows the different workloads used.

Id	Workload name	Application type
01	n-gram	NLP
02	Logistic Regression	Machine Learning
03	Support Vector Machine	Machine Learning
04	Pi computation	Scientific Computing
05	Fp-growth	Data Mining
06	Word Count	NLP
07	K-Means	Data Mining
08	PCA	Data Mining
09	GaussianMixture	Machine Learning
10	Pagerank	Graph Processing
11	Random forest	Machine Learning
12	Databricks K-Means	Data Mining
13	Databricks Naive Bayes	Machine Learning
14	Databricks Pearson Correlation	Statistics
15	Euler computation	Scientific Computing

Table 6.2: Applications used for the experiments.

We used the previous benchmarks to build a dataset containing executions of Spark jobs. To build our dataset, we generated 100 combinations of parameters for each workload, as

other works [23] have done. We used a BO process to generate the samples for our dataset. The main reason for using a BO process instead of a random search is to minimize bad quality examples and the number of failed runs. We did try a random process to build a dataset, but it generated many configurations that, once run, returned memory errors (or were slower than the default configuration). Since the execution time of an application with a particular configuration can have some variance, we ran the jobs five times and stored the mean over the repetitions. This value was used as input to the Bayesian Optimizer during the dataset creation.

Computational Environment

To conduct all the experiments described in this section we used a 4 node cluster, with Spark version 2.4.1 configured in cluster mode with one node as master and three as slave nodes. Each node had two 10-core Intel®Xeon E5-2630 v4 CPU @ 2.20GHz, which sums up a total of 40 threads per node, as the hyper-threading was active. Each node also had 128 GB of RAM and they were interconnected by a 10Gbps Ethernet network.

In order to make a fair (and easy to understand) comparison the executions for **Time-to-solution speedup** are evaluated in containers executed with exclusivity of resources. In other words, a single Spark application is being executed for each Spark configuration tested. It is out of the scope of our work, but an interesting area of research, to generalize the oracle for environments with co-executions as well as heterogeneous hardware. We used scikit-learn [59] to build the different performance models presented in this section.

A note on the distances between feature descriptors

The SBO algorithm proposed uses a feature vector $\boldsymbol{x}$ that is built after an application is run using the Default Spark configuration. A natural question that may arise is how well the information summarized in the feature vector using these settings describes similar feature vectors generated with other Spark configurations for the same application. Note that the descriptor is not invariant with respect to the Spark configuration provided since several of the metrics that are gathered (such as the aggregated executor CPU usage across tasks) are influenced by configuration decisions (such as the number of executor cores).

In Section 6.3.1, we show that the descriptors boost the performance of the learning algorithms. Nevertheless, another reasonable approach to assess the quality of a descriptor is to verify that the vector for a particular job is closer to vectors from the same Spark workload than to vectors from very different workloads. Let us denote by $\boldsymbol{x}^{(0,w_j)}$ the descriptor for workload w_j generated using the default Spark configuration, which we will call the default

feature vector for application w_j. Let us denote by $\boldsymbol{x}^{(k,w_i)}$ the feature vector (for application w_i) generated using the k-th Spark configuration sampled to create the training data.

Let us denote by M a matrix containing at position $M_{i,j}$ the mean of the euclidean distances between $\boldsymbol{x}^{(0,w_j)}$ and the vectors $\boldsymbol{x}^{(1,w_i)}, \ldots, \boldsymbol{x}^{(100,w_i)}$. That is,

$$M_{i,j} = \frac{1}{100} \sum_{k=1}^{100} \left\| \boldsymbol{x}^{(0,w_j)} - \boldsymbol{x}^{(k,w_i)} \right\|_2 \tag{6.3}$$

This matrix contains at column j the average distance between the default feature vector of application w_j and all the feature descriptors of application w_i for $i \in \{1, \ldots, 15\}$.

We observed that the matrix has a diagonal containing most of the lowest values in each column. This shows that the default descriptor for application w_j is generally closer to the vectors for the same application $\boldsymbol{x}^{(k,w_j)}$ than to other vectors from different applications $\boldsymbol{x}^{(k,w_i)}$. A manual inspection of M reveals some interesting results:

- Some applications are similar in their workloads types and and feature descriptors. For example, the smallest value in column 2 is $M_{2,2} = 2$. This means that the default feature vector from application 2 is at an average distance of 2 from all the vectors of the same application. Moreover, the second lowest value in column 2 corresponds to $M_{3,2} = 2.1$. This is reasonable, since applications 2 and 3 correspond to a Logistic Regression and a Support Vector Machine respectively; both of these workloads share a very similar iterative training algorithm.

- Some applications are disimilar in their workloads types and feature descriptors. For example, the smallest value in column 6 is $M_{6,6} = 5.7$. This means that the default feature vector from application 6 is at an average distance of 5.7 from all the vectors of the same application. Moreover, the second lowest value in column 6 corresponds to $M_{14,6} = 9.7$. In this case, this suggests that applications 6 and 14 are not very similar. This is reasonable, because applications 6 and 14 are a Word Count and a Pearson Correlation test, respectively.

6.3.1 *Feature descriptor quality evaluation*

Feature descriptor Quality is critical for our system. If the oracle is not able to provide sensible predictions for new, unseen workloads, then the selection process based on the values given by the oracle will be flawed. Moreover, our work requires that the features extracted from the SparkEventLog generalize to new workloads, otherwise the models cannot be used for the SBO process.

The goal of this experiment was to assess the quality of the features described in Section 6.2.1 with respect to standard feature representations of workloads found in many state-

of-the-art works [5, 9, 47, 55]. The most common feature representations of the workloads combine three sources of information: the Spark configuration (sc), the dataset size (ds) and an identifier for the application (app_{id}). Works like [55] use (sc, ds), whereas [5, 47] use (sc) as input and [9] uses (sc, ds, app_{id}). Our work uses (sc, ds, ef) as the feature vector, where (ds, ef) corresponds to the "SparkEventLog features" described in Section 6.2.1.

Data Processing: In order to input the data to the machine learning models, we performed some preprocessing on the raw data. The types of variables involved in the Spark Configuration are shown in Table 6.3.

Spark Configuration Parameters	Search Space	Space Encoding	Default Configuration
spark.task.cpus	[1, 2]	Integer	1
spark.executor.cores	[1, 40]	Integer	40
spark.executor.memory	[4, 40]	Integer	20g
spark.memory.fraction	[0.4, 0.5, 0.6, 0.7, 0.8]	Ordinal	0.6
spark.memory.storageFraction	[0.3, 0.4, 0.5, 0.6, 0.7, 0.8]	Ordinal	0.5
spark.default.parallelism	[20, 400]	Integer	40
spark.shuffle.compress	[true, false]	Categorical	true
spark.shuffle.spill.compress	[true, false]	Categorical	true
spark.broadcast.compress	[true, false]	Categorical	true
spark.rdd.compress	[true, false]	Categorical	false
spark.io.compression.codec	[lz4, lzf, snappy]	Categorical	lz4
spark.reducer.maxSizeInFlight	[24m, 48m, 96m]	Categorical	48m
spark.shuffle.file.buffer	[32k, 64k]	Categorical	32k
spark.serializer	[JavaSerializer, KryoSerializer]	Categorical	JavaSerializer

Table 6.3: Search Space of the different parameters we tuned with the default values. The units in the Default Configuration correspond to the notation used by Spark. For example, 20g corresponds to 20 GB.

The heterogeneity of the variable types required some preprocessing in order to build the sc vector. In particular, categorical variables were converted using a one-hot encoding transformation. The remaining variables were then treated with either Rescaling or Standardization. Rescaling was performed on the [0,1] range and Standardization transformed the variables so that the values have zero-mean and unit-variance.

The models may perform differently depending on whether Rescaling or Standardization is applied. We implemented a pipeline mechanism to automatically decide during cross-validation which preprocessing was best for each model, so as to adapt the decision to each of the learners. The data transformation was introduced as one of the steps during model selection, as if it were a hyper-parameter of the model.

Experiment settings

Model Evaluation: We created 15 test sets by splitting the data 15 different ways. The final test set metrics provided in this section show the average of the results for a model with a fixed set of hyper-parameters over the 15 test sets. The 15 splits were created using the application identifier in order to evaluate models on samples of a workload that did not appear during training. This methodology of generating partitions is usually referred to as Leave One Group Out Cross-Validation (LOGO-CV). Using this methodology we ensure that models are evaluated on samples of a workload that had never been seen during training (here the "group" consists on all the examples that contain the same workload identifier). We chose LOGO-CV, instead of standard Cross-validation, because it more closely resembles the scenario we want our model to generalize. In a production environment, we would like a system that can provide good predictions for workloads that have never appeared in the past. In general, we do not want to assume that the training and validation splits contain examples of all possible workloads, because in a production environment this scenario is unlikely.

After deciding which algorithm and hyper-parameters are the best ones, the error values computed using LOGO-CV reveal how well (on average) a model should perform on a new workload never been seen during training.

Model selection: We trained 8 machine learning models using 5 different sets of features. In order to select the models robustly, we performed LOGO-CV on each of the 15 splits of our data. The average errors across all of the validation partitions of the different splits were used to select the best oracle candidate.

We tested the following combinations of features: $(sc), (sc, app_{id}), (sc, ds), (sc, ds, app_{id})$ and (sc, ds, ef) as input to the learning algorithms. The last combination of features corresponds to our proposed solution. In order to make a fair comparison between the different features found in the literature, we performed all the experiments with the same space of hyper-parameters for the machine learning models. The search space is given in Table 6.4. An exhaustive exploration of all the combinations of hyper-parameters was performed. Training each of the models from Table 6.4, including hyper-parameter optimization, takes less than one minute in a single node of our computation environment. Therefore, the overhead of training a model in our system is negligible.

Model	Hyper-parameter search space
Elastic Net	alpha = [1e-10, 1e-9, ..., 10.0]
GBM	learning_rate = [0.001, 0.01, 0.1, 0.5] ,max_depth =[3, 10, 20, None]
KNN	n_neighbors= [1,2,3,4,5,6], p=[1,2]
Lasso	alpha = [1e-10, ..., 1.0, 10.0]
MLP	hidden_layer_sizes = [[200], [300],[500], [200, 200], [300, 300], [500, 500]],activation = ["tanh","relu"],
	learning_rate = [0.0001, 0.001, 0.01]
Random Forest	max_depth = [10, 20, None], max_features: ["auto", "sqrt"], n_estimators: [10, 20, 50, 100, 200]
Ridge	alpha = [1e-10, 1e-9, ..., 10.0]
SVR	C = [1e-10, 1e-9, ..., 10.0], kernel = ["linear", "poly", "rbf"]

Table 6.4: Hyper-parameter search space.

Results

The results of the experiments of the different models are given in Table 6.5, which contains the training and validation MSE for the best parameters found for each model. The table is organized in five column groups, one for each set of features tested. Each row in the table shows the results for a particular model containing the best hyper-parameters found during training. This table presents a comparison of the proposed features (sc, ds, ef) with different sets of features found in the state-of-the-art work. The results in the table show that the best results in the validation were achieved with the descriptor (sc, ds, ef). It can also be seen that with input features (sc, ds, ef), the Random Forest Regressor achieved the best results in the validation data. Therefore, we chose the Random Forest (with the features (sc, ds, ef)) as our oracle for the experiments in Section 6.3.2. The selected Random Forest was trained with the hyper-parameters that minimize the error during hyper-parameter exploration.

After the best hyper-parameters for each model had been found, we evaluated each model across the test sets. Table 6.6 provides the test set results for the models selected from Table 6.5 for each of the different feature groups. Note that (sc, app_{id}) has no Reference because we could not find any work using that feature vector as input to the learning algorithms. Nevertheless, we have included it for completeness because it is a sensible approach that improves the results over sc.

Features	(sc)		(sc, app_{id})		(sc, ds)		(sc, ds, app_{id})		(sc, ds, ef)	
	train	valid	train	valid	train	valid	train	valid	train	valid
model										
Elastic Net	161.2	198.4	69.4	204.5	153.0	193.7	67.4	199.5	50.3	167.9
GBR	85.6	**178.6**	14.3	**164.2**	50.9	195.3	12.4	**165.1**	2.8	173.7
KNN	117.7	224.5	47.3	224.5	106.5	211.1	44.1	211.1	8.7	227.6
Lasso	161.0	196.2	63.4	205.2	152.4	192.7	61.1	214.9	29.2	157.2
MLP	194.4	211.5	190.3	211.2	194.6	210.5	194.4	210.7	17.2	371.6
Random Forest	17.2	183.5	5.7	178.8	12.2	**175.5**	4.6	179.0	0.8	**132.6**
Ridge	159.2	202.6	61.2	212.3	150.5	202.8	57.9	236.2	20.7	272.0
SVR	167.5	209.6	73.5	213.0	158.0	207.8	73.1	219.8	31.1	258.5

Table 6.5: Training and validation MSE errors for different models.

Features	Test MSE	Improvement over sc	Reference
(sc)	167.9	0.0%	[5, 47]
(sc, app_{id})	162.0	3.5%	-
(sc, ds)	155.6	7.3%	[55, 92]
(sc, ds, app_{id})	158.4	5.6%	[9]
(sc, ds, ef)	110.7	34.1%	ours

Table 6.6: Test errors and improvement over the basic *sc* feature set.

6.3.2 *Time-to-solution speedup evaluation*

To validate the time-to-solution speedup, we need to measure two metrics. First, the quality (runtime) of the solution provided by SBO in comparison with the best configuration found by BO. Second, the time required by SBO to provide a configuration with respect to the time needed by BO.

Default Settings: To assess the quality of a SparkConfiguration found by BO or SBO, we compared the execution time of the provided configurations with the execution time achieved by the default configuration. This quantity is the speedup with respect to the default configuration form Spark. The default configuration we used is given in Table 6.3. Note that this configuration is the same used by Spark by default, except for the number of

executor cores and the amount of executor memory. We set the number of executor cores to 40 (the total number of cores per node in our cluster) and the executor memory to 20 Gigabytes ("20GB"). The memory was increased to avoid job failures for the workloads with the default Spark configuration. This is important because in the event of an everlasting runtime for a job, the speedup with respect to the default configuration could be theoretically infinite, providing unrealistic speedups with respect to a bad default baseline. Changing the memory settings is a common decision also found in other works such as [55]. In our case, eight of the fifteen jobs failed using the default Spark configuration, which uses $1GB$ of executor memory. To prevent this issue arising, we tested the default configuration with the executor memory set to $1GB, 2GB, \ldots, 50GB$ and stopped at the first value able to run all workloads without any memory crash, which was $20GB$.

Experiments: The objective of the experiments was to measure the quality of the solutions found by SBO with respect to the best solution found with BO. Here we will denote by BO-K a Bayesian Optimization process that executes an application K times. In order to make a useful comparison, all the results were evaluated on workloads that neither BO nor SBO had accessed before starting the optimization procedure. In the case of BO, we set the budget to 20 iterations because it is the minimum amount of iterations needed to reach a speedup over the default configuration across the workloads we used. As other works, such as [23], use 35 or 100 iterations as maximum budget, we decided to perform BO-100 as a reasonable upper bound on the quality of the results. In the case of SBO, we allowed a budget of 100 queries to the oracle (which takes less than two seconds). Then only one or two evaluations of the workload were executed (which we refer to as SBO-1 and SBO-2, respectively), because the goal of our methodology is to quickly find solutions with a reasonable quality.

Speedup over default configuration results: The improvement of SBO with respect to the default configuration is shown in Figure 6.4, which contains the runtime of the default configuration and the runtime of the configuration found with SBO-1 and SBO-2. The figure also shows a box plot of the distribution of runtimes found during the Bayesian Optimization process performed to create the dataset. The box plots how the distribution of the execution times found by a BO process with a budget of 100 runs per application. We can see that SBO-2 improved on the default configuration in all cases, except for workload 11, where there is no speedup. It is interesting to notice that workloads 1, 10 and 14 need SBO-2, since SBO-1 does not improve the quality of the results. Nevertheless, such cases are precisely the examples where the box plot shows small variance in the runtime. In those cases, even BO-100 was not able to achieve a significant speedup with respect to the default Spark configuration. The box plot shows that workloads 10 and 14 have most runtimes clustered around a single point. This suggests that those cases are very hard to optimize since a BO with 100 executions of the application was not able to improve the results.

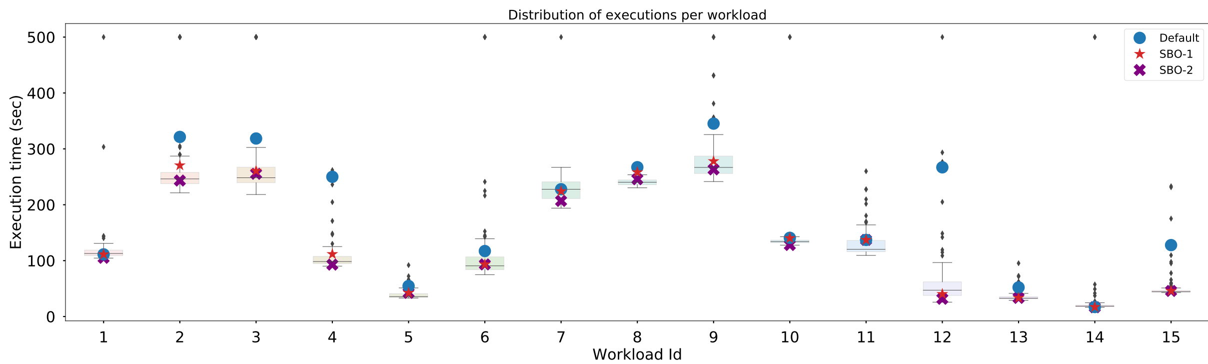


Figure 6.4: Best solutions found by SBO-1 and SBO-2.

The results of the speedups found by BO-20, BO-100, SBO-1 and SBO-2 with respect to the default configuration runs are shown in Figure 6.5. Note that BO achieved higher quality solutions than SBO-1 in almost all cases, except for 12. Nevertheless, the difference between BO and SBO-2 is minimal. It should also be noted that SBO-1 always achieves better results than random search, even if we perform 10 random searches. Nevertheless, the random search can achieve surprisingly good results, which is consistent with the experimental results observed in other literature [23].

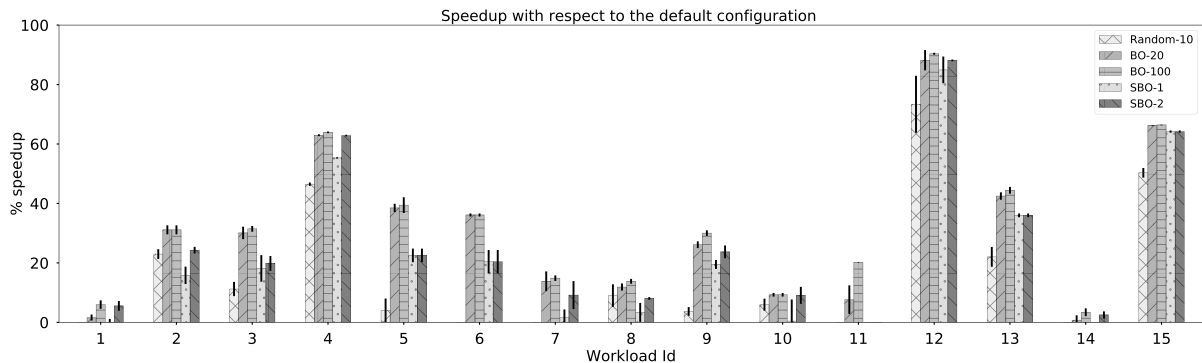


Figure 6.5: Speedup with respect to the default configuration per application (the higher the better). An empty bar reflects no improvement over the default configuration.

Time-to-solution: Table 6.7 gives the time-to-solution speedups of our proposed method with respect to the BO process with 20 iterations (BO-20). The results shown in the table include the time spent running the Spark configurations, as well as the time needed to parse the SparkEventLog and the time required by the oracle to make predictions. Moreover, the table includes the total search time for the different algorithms tested. Our methodology allows us to find a good solution in 5 minutes (on average) using SBO-2. A naive solution, such as random search, takes almost half an hour to run, while achieving worse results than SBO-2, as can be seen in Figure 6.5. BO-20 can achieve, in some cases, better configurations

than SBO-2, but with a higher average cost of 53 minutes. Figure 6.6 shows the total search time for the different methods across applications.

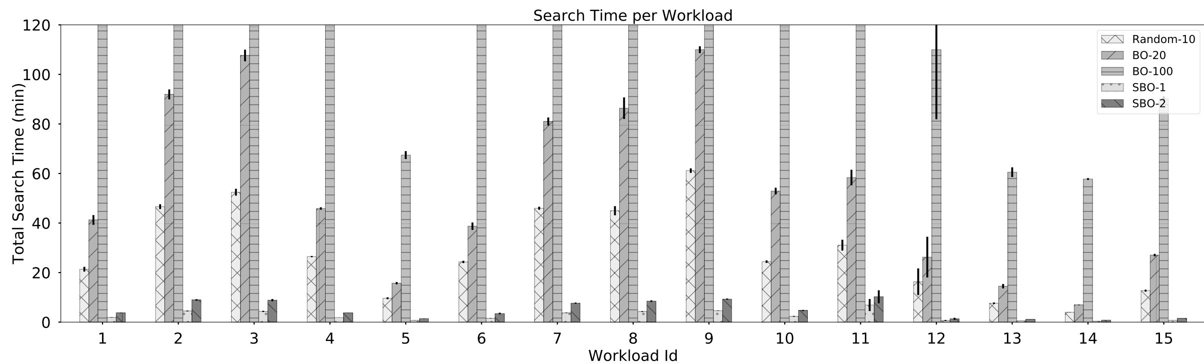


Figure 6.6: Total search time, in minutes, to find a configuration for each method (the lower the better). The plot was cropped at 120 min to improve readability.

In Table 6.7 we can see that the SBO-1 optimization process is 25 times faster than the BO-20 process. At first glance, it might be surprising that the SBO-1 is 25 times faster than BO-20, which performs only 20 iterations. The explanation for this behavior is the runs performed by BO which may include bad configurations that are executed during the optimization procedure, which have to be sampled in order to explore the search space and provide a good exploration-exploitation tradeoff. In the case of SBO-1, we only executed the best configuration predicted by the oracle and none of them was bad. This allows the average speedup of SBO-2 to be up to 12x faster than BO-20.

	Speedup over BO-20	Search Time (min)
BO-20	$1.0x$	53.6
BO-100	$0.2x$	243.2
Random-10	$1.9x$	28.6
SBO-1	$25.1x$	2.6
SBO-2	$12.4x$	5.0

Table 6.7: Time-to-solution for the different methods. The first column shows the improvement with respect to 20 runs of Bayesian Optimization. The second column the overall search time in minutes.

6.4 CONCLUSIONS

This chapter introduced a non invasive technique to create a feature descriptors from the log file of an application. The key idea of this work is that the created feature descriptor contains relevant information from the workload to be optimized. To verify this, we show that the descriptor significantly benefits the quality of machine learning models trained with the presented feature vector. In particular, our experiments show that models improve by up to 34% the prediction quality when trained and evaluated with our feature vecture with respect to other state-of-the-art features.

Moreover, we introduced a simple, easy to implement, modification of a Bayesian Optimization (BO) process that we called "Simulated Bayesian Optimization" (SBO). Using this algorithm, in conjunction with the performance models built on top of the proposed Spark feature descriptor, we implemented an optimization pipeline that can be used for auto-tuning Spark workloads. Our experiments show that SBO can speed up the optimization process considerably with respect to the standard BO approach, up to a $12x$ speedup with results of almost equal quality.

7

CONCLUSION AND FUTURE WORK

7.1 Summary of the results

In this thesis, we presented three contributions covering workload characterization, estimation of workload colocation behaviour and application performance estimation under tunable parameters. Most current state of the art techniques that tackle the three previous problems are based on application dependent knowledge that is introduced into the solution. The goal of this thesis was to show that learning algorithms could be applied to the previous three problems providing solutions based on input metrics extracted from applications observed as a black boxes.

This final chapter reviews all the relevant results achieved in this dissertation, and discusses future work that is inlined with the core idea of interfacing with applications provided as black boxes.

Workload phase characterization through machine learning

The first contribution of this thesis presents a mechanism for mapping slices of workload trace to phase ids that can be used as feature vectors for a variety of tasks. Our solution was based on metrics such as CPU and Memory usage and did not have any knowledge about the monitored application.

First, we studied the behaviour of the cluster ids found in the workload data, where we could see interesting resource patterns that described sensible workload phases. Then, we showed that we could use phase descriptors as input to learning algorithms trained to predict future phase behaviour. Using this approach we showed that we could create a policy for auto-scaling workloads that provided better results than other reactive policies found in the literature.

SEQUENCE-TO-SEQUENCE MODELS FOR RESOURCE ESTIMATION OVER TIME UNDER CO-SCHEDULING SCENARIOS

The second contribution of this thesis presents a sequence-to-sequence model that translates traces of applications executed in isolation to the expected trace of applications co-located sharing resources. Our work introduces a simple "Percentage Completion" feature that consistently improves the standard stopping criteria used to decide when to stop generating the output trace.

The model was tested using a dataset created combining three benchmark suits containing a wide variety of applications. The experiments showed that the presented model provides lower error with respect to classical Machine Learning regressors. Moreover, we showed that our solution is capable of predicting the execution time of co-scheduled applications with reasonable quality.

Despite the promising results, we could not find any other work that predicted execution traces over time, making the work original but "hard to compare with". Moreover, our work is meant to be used in environments where the hardware resources are fixed. Note that with a small modification we could apply the model in environments with different servers, if we provide information about the underlying hardware, such as the CPU cores and CPU frequency. This was out of the scope of our work and probably would only be reasonable for a big company that already possesses a diverse collection of nodes.

FAST TIME-TO-SOLUTION BIG DATA WORKLOAD AUTO-TUNING WITH REUSABLE MACHINE LEARNING MODELS

The third contribution of this thesis is an algorithm that allows fast time-to-solution results for tuning Spark workloads using a simple search mechanism based on a Bayesian Optimization method. The algorithm is based on a feature vector that can be created inspecting a Spark application as a black box, since it uses a file of event logs that is accessible from the outside of the application.

Our experiments showed a faster time-to-solution when compared with other optimization techniques found in the literature. The main reason of our success was the fact that we did not include a categorical variable into the learning algorithm containing a label for each application with the goal of creating a solution that was not application dependant. Nevertheless, to achieve this, we had to extract information about the underlying application behaviour in one way or another. To do this, we parsed the SparkEventLog. Our goal was to extract relevant features about the application and then use this information to provide the learning algorithm more features that other methods in the literature did not use.

Despite the good results, our solution was based on creating a feature vector that was developed for Spark. The proposed solution is general enough to be used in other settings, but the preprocessing step to create the feature vector was created to extract information relevant for Spark workloads (such as the Spark Actions and Transformations). Therefore, to apply a similar solution for a different platform, assuming are provided with a file containing the logs of the application, a specific preprocessing step should be adapted.

7.2 FUTURE WORK

The work presented in this dissertation faced some limitations of modern Machine Learning methods which current developments in the area of Deep Learning try to solve. One of such limitations is the amount of feature engineering needed to solve a task, which is intended do decrease or disappear with newer techniques.

The first contribution showed how a CRBM can be used to find phases in a workload trace and we used the presented phase descriptor to solve a container auto-scaling problem. Current container technologies, such as Docker, do not allow monitoring GPU metrics, which are very relevant in the HPC environment. Therefore, future work could investigate how phase descriptor jointly maps GPU and CPU usage with the goal to improve container auto-scaling with workloads that are executed on instances with GPU hardware acceleration.

On a different front, one of our experiments showed that the activations of the CRBM gave rise to a vector that clearly untangled the latent activities performed in the time-series (jogging vs walking). This suggests that the presented mechanism could be used to extract features for time-series, not only for detecting phases, but for generating features used for time-series classification. In particular it would be interesting to compare those features with features such as Shapelets [90], that are very powerful but very expensive to compute.

The second and third contributions in this work are based on first running a program and then extracting information from the execution of the program. In our second contribution we need to run two applications to extract the workload traces which are then feed to a Sequence-to-Sequence model. The neural network architecture was developed to tackle scenarios where two applications are co-located. Further study is needed to generalize this type of network to handle several input applications that are expected to be run together. Moreover, a time aware resource scheduler could be implemented to show that the proposed approach, which allows resource prediction over time, provides relevant knowledge to improve current software scheduling software. We would like to emphasise that workload scheduling is a huge area of research, and this investigation was out of the scope of our work.

In our third contribution a program needs to be run in order to extract a descriptor that characterises relevant aspects of the runtime execution. A natural question arises: could we extract information without even running the application? An application is simply a list of instructions to be executed, which are written in a certain programming language. Therefore, assuming we are allowed to read the source code of an application (which in some cases it might not be available for privacy reasons), we could probably extract valuable information from the raw source code of the application. We think this could be a very interesting research line to pursue.

In our third contribution we generated feature descriptors from the log file of Spark applications to provide the learning algorithms some information about the workload. This descriptor provided relevant information such as the actions and transformations that an application used during the execution. Nevertheless, our approach was "blind" to the code running inside those high level routines. Despite our methodology is general the descriptor in the experiments was designed for Spark. There are other approaches to gather information from log data, or even the source code of the application. Works such as [18] optimize OpenCL kernels with performance models that are based on the source code of the program. This setting is easier than the Big Data optimization problem because OpenCL kernels typically have from 10 to 50 lines of code. Nevertheless, this is a promising direction for future research because it provides a methodology that requires no feature engineering. The overall idea of this works is to use a RNN that reads the source code of the application (word by word) and outputs the execution time of the code. This methodology does not require, ideally, any hand coded parser. Here, we emphasize "ideally" because even though the work from [18] reads the source code of the application, hand crafted text processing functions were used to clean the raw source code of the OpenCL kernels. This is done to remove unwanted text from the code to facilitate the RNN its objective. Therefore, it could be argued that the programmer effort of feature engineering was replaced by the programmer effort of creating code that cleaned and standardised the source code of the applications.

Whatever future Data Centers become, it is clear that Machine Learning techniques will play an important role for the efficient management of hardware resources.

BIBLIOGRAPHY

[1] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. "Predicting completion times of batch query workloads using interaction-aware models and simulation." In: *ACM International Conference Proceeding Series* (2011), pp. 449–460. DOI: 10.1145/1951365.1951419.

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. "Automatic database management system tuning through large-scale machine learning." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* Part F127746 (2017), pp. 1009–1024. ISSN: 07308078. DOI: 10.1145/3035918.3064029.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una May O'Reilly, and Saman Amarasinghe. "OpenTuner: An extensible framework for program autotuning." In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* (2014), pp. 303–315. ISSN: 1089795X. DOI: 10.1145/2628071.2628092.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." In: (2014), pp. 1–15. ISSN: 0147-006X. DOI: 10.1146/annurev.neuro.26.041002.131047. arXiv: 1409.0473. URL: http://arxiv.org/abs/1409.0473.

[5] Liang Bao, Xin Liu, and Weizhao Chen. "Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks." In: *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018* (2019), pp. 181–190. DOI: 10.1109/BigData.2018.8622018. arXiv: 1808.06008.

[6] Zhendong Bei, Zhibin Yu, Huiling Zhang, and Wen Xiong. "RFHOC : A Random-Forest Approach to Auto-Tuning Hadoop ' s Configuration." In: 27.5 (2016), pp. 1470–1483.

[7] Josep L. Berral, Chen Wang, and Alaa Youssef. "AI4DL: Mining Behaviors of Deep Learning Workloads for Resource Management." In: *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing*. 2020.

[8] Josep Ll. Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. "Towards Energy-aware Scheduling in Data Centers Using Machine Learning." In: *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. e-Energy '10. Passau, Germany: ACM, 2010, pp. 215–224. ISBN: 978-

1-4503-0042-1. DOI: 10.1145/1791314.1791349. URL: http://doi.acm.org/10.1145/1791314.1791349.

[9] Josep Ll Berral, Nicolas Poggi, David Carrera, Aaron Call, Rob Reinauer, and Daron Green. "ALOJA-ML: A framework for automating characterization and knowledge discovery in hadoop deployments." In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 2015-Augus (2015), pp. 1701–1710. DOI: 10.1145/2783258.2788600. arXiv: 1511.02030.

[10] Marcello M. Bersani, Francesco Marconi, Damian A. Tamburri, Andrea Nodari, and Pooyan Jamshidi. "Verifying big data topologies by-design: a semi-automated approach." In: *Journal of Big Data* 6.1 (2019). ISSN: 21961115. DOI: 10.1186/s40537-019-0199-y. URL: https://doi.org/10.1186/s40537-019-0199-y.

[11] Manuele Bicego, Vittorio Murino, and Mario A.T. Figueiredo. "Similarity-based clustering of sequences using hidden markov models." In: *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)* 2734 (2003), pp. 86–95. ISSN: 03029743. DOI: 10.1007/3-540-45065-3_8.

[12] David Buchaca, Joan Marcual, Josep LLuis Berral, and David Carrera. "Sequence-to-sequence models for workload interference prediction on batch processing datacenters." In: *Future Generation Computer Systems* 110 (2020), pp. 155–166. ISSN: 0167739X. DOI: 10.1016/j.future.2020.03.058. URL: https://doi.org/10.1016/j.future.2020.03.058.

[13] The Case and Energy-proportional Computing. "The Case for Energy-Proportional Computing." In: *Google* December (2007), pp. 33–37. ISSN: 0018-9162. DOI: 10.1109/MC.2007.443. URL: http://www.barroso.org/publications/ieee{_}computer07.pdf.

[14] Y Chen, P Goetsch, M A Hoque, J Lu, and S Tarkoma. "d-Simplexed: Adaptive Delaunay Triangulation for Performance Modeling and Prediction on Big Data Analytics." In: *IEEE Transactions on Big Data* (2019), p. 1. ISSN: 2332-7790. DOI: 10.1109/TBDATA.2019.2948338. URL: https://ieeexplore.ieee.org/document/8878273.

[15] T. Chiba and T. Onodera. "Workload characterization and optimization of TPC-H queries on Apache Spark." In: (2016), pp. 112–121. DOI: 10.1109/ISPASS.2016.7482079.

[16] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: (2014). ISSN: 09205691. DOI: 10.3115/v1/D14-1179. arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[17] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In: *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference* (2014), pp. 1724–1734. DOI: 10.3115/v1/d14-1179. arXiv: 1406.1078.

[18] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. "End-to-End Deep Learning of Optimization Heuristics." In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* 2017-Septe (2017), pp. 219–232. ISSN: 1089795X. DOI: 10.1109/PACT.2017.24.

[19] Christina Delimitrou and Christos Kozyrakis. "IBench: Quantifying interference for datacenter applications." In: *Proceedings - 2013 IEEE International Symposium on Workload Characterization, IISWC 2013* (2013), pp. 23–33. DOI: 10.1109/IISWC.2013.6704667.

[20] Christina Delimitrou and Christos Kozyrakis. "Paragon : QoS-Aware Scheduling for Heterogeneous Datacenters." In: (2013), pp. 77–88. ISSN: 0272-1732. DOI: 10.1145/2451116.2451125.

[21] Chen Ding, Sandhya Dwarkadas, Michael C. Huang, Kai Shen, and J. B. Carter. "Program phase detection and exploitation." In: *IPDPS*. IEEE, 2006. URL: http://dblp.uni-trier.de/db/conf/ipps/ipdps2006.html#DingDHSC06.

[22] Sina Esfandiarpoor, Ali Pahlavan, and Maziar Goudarzi. "Structure-aware online virtual machine consolidation for datacenter energy improvement in cloud computing." In: *Computers & Electrical Engineering* 42 (2015), pp. 74 –89. ISSN: 0045-7906. DOI: http://dx.doi.org/10.1016/j.compeleceng.2014.09.005. URL: http://www.sciencedirect.com/science/article/pii/S004579061400233X.

[23] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. "Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics." In: (2020), pp. 27–29. arXiv: 2001.08002. URL: http://arxiv.org/abs/2001.08002.

[24] G. D. Forney. "The Viterbi algorithm." In: *Proc. of the IEEE* 61 (1973), pp. 268 –278.

[25] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. "BigBench: Towards an Industry Standard Benchmark for Big Data Analytics." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 1197–1208. ISBN: 978-1-4503-2037-5.

[26] Pierre Hansen and Nenad Mladenovi? "J-Means: a new local search heuristic for minimum sum of squares clustering." In: *Pattern Recognition* (2001). ISSN: 0031-3203. DOI: 10.1016/S0031-3203(99)00216-2. URL: http://www.sciencedirect.com/science/article/pii/S0031320399002162.

[27] Álvaro Brandón Hernández, María S. Perez, Smrati Gupta, and Victor Muntés-Mulero. "Using machine learning to optimize parallelism in big data applications." In: *Future Generation Computer Systems* 86 (2018), pp. 1076–1092. ISSN: 0167739X. DOI: 10.1016/j.future.2017.07.003.

[28] Geoffrey E. Hinton. "Training Products of Experts by Minimizing Contrastive Divergence." In: *Neural Compututing* 14.8 (Aug. 2002), pp. 1771–1800. ISSN: 0899-7667. DOI: 10.1162/089976602760128018. URL: http://dx.doi.org/10.1162/089976602760128018.

[29] Geoffrey Hinton. "A Practical Guide to Training Restricted Boltzmann Machines A Practical Guide to Training Restricted Boltzmann Machines." In: *Computer* 9.3 (2010), p. 1. ISSN: 364235288X. DOI: 10.1007/978-3-642-35289-8_32. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.170.9573{\&}rep=rep1{\&}type=pdf.

[30] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* (1997). ISSN: 08997667. DOI: 10.1162/neco.1997.9.8.1735.

[31] Eugene Hsu, Kari Pulli, and Jovan Popović. "Style Translation for Human Motion." In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 1082–1089. ISSN: 0730-0301. DOI: 10.1145/1073204.1073315. URL: http://doi.acm.org/10.1145/1073204.1073315.

[32] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis." In: *Lecture Notes in Business Information Processing* 74 LNBIP.May (2011), pp. 209–228. ISSN: 18651348. DOI: 10.1007/978-3-642-19294-4_9.

[33] "Intel 64 R and IA-32 Architectures Software Developer's Manual vol. 3B: System Programming Guide, Part 2, Sep 2014." In: ().

[34] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. "Empirical prediction models for adaptive resource provisioning in the cloud." In: *Future Generation Computer Systems* 28.1 (2012), pp. 155–162. ISSN: 0167739X. DOI: 10.1016/j.future.2011.05.027. URL: http://dx.doi.org/10.1016/j.future.2011.05.027.

[35] T. Ivanov, R. Niemann, S. Izberovic, M. Rosselli, K. Tolle, and R. V. Zicari. "Performance Evaluation of Enterprise Big Data Platforms with HiBench." In: 2 (2015), pp. 120–127. DOI: 10.1109/Trustcom.2015.570.

[36] Pooyan Jamshidi and Giuliano Casale. "An uncertainty-aware approach to optimal configuration of stream processing systems." In: *Proceedings - 2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016* i (2016), pp. 39–48. DOI: 10.1109/MASCOTS.2016.17. arXiv: 1606.06543.

[37] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. "Transfer Learning for Improving Model Predictions in Highly Configurable Software." In: *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017* (2017), pp. 31–41. DOI: 10.1109/SEAMS.2017.11. arXiv: 1704.00234.

[38] Iñigo Jauregi Unanue, Ehsan Zare Borzeshi, and Massimo Piccardi. "Recurrent neural networks with specialized word embeddings for health-domain named-entity recognition." In: *Journal of Biomedical Informatics* 76 (2017), pp. 102–109. ISSN: 15320464. DOI: 10.1016/j.jbi.2017.11.007. arXiv: 1706.09569. URL: http://dx.doi.org/10.1016/j.jbi.2017.11.007.

[39] Jitendra Kumar and Ashutosh Kumar Singh. "Workload prediction in cloud using artificial neural network and adaptive differential evolution." In: *Future Generation Computer Systems* 81 (2018), pp. 41–52. ISSN: 0167-739X. DOI: 10.1016/j.future.2017.10.047. URL: https://doi.org/10.1016/j.future.2017.10.047.

[40] Mayuresh Kunjir and Shivnath Babu. "Black or White? How to Develop an AutoTuner for Memory-based Analytics." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1667–1683.

[41] Ping-Huan Kuo and Chiou-Jye Huang. "A High Precision Artificial Neural Networks Model for Short-Term Energy Load Forecasting." In: *Energies* 11.1 (2018), p. 213. ISSN: 1996-1073. DOI: 10.3390/en11010213. URL: http://www.mdpi.com/1996-1073/11/1/213.

[42] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, Nicholas Fuller, and MapReduce Applications. "MRONLINE: MapReduce Online Performance Tuning Figure 1: Current offline performance tuning approach used for." In: (2014). DOI: 10.1145/2600212.2600229. URL: http://dx.doi.org/10.1145/2600212.2600229..

[43] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. "SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark." In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. CF '15. Ischia, Italy: ACM, 2015, 53:1–53:8. ISBN: 978-1-4503-3358-0. DOI: 10.1145/2742854.2747283. URL: http://doi.acm.org/10.1145/2742854.2747283.

[44] Guangdeng Liao, Kushal Datta, and Theodore L Willke. "Gunther: Search-based Auto-tuning of Mapreduce." In: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 406–419. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_42. URL: http://dx.doi.org/10.1007/978-3-642-40047-6{_}42.

[45] Jessica Lin, Eamonn Keogh, and Wagner Truppel. "Clustering of streaming time series is meaningless." In: (2003), p. 56. DOI: 10.1145/882095.882096.

[46] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. "Heracles: Improving Resource Efficiency at Scale." In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15* (2015), pp. 450–462. ISSN: 10636897. DOI: 10.1145/2749469.2749475. URL: http://dl.acm.org/citation.cfm?doid=2749469.2749475{\%}5Cnhttp://doi.acm.org/10.1145/2749469.2749475{\%}5Cnhttp://dl.acm.org/ft{_}gateway.cfm?id=2749475{\&}type=pdf.

[47] Ni Luo, Zhibin Yu, Zhendong Bei, Chengzhong Xu, Chuntao Jiang, and Lingfeng Lin. "Performance modeling for spark using SVM." In: *Proceedings - 2016 7th International Conference on Cloud Computing and Big Data, CCBD 2016* (2017), pp. 127–131. DOI: 10.1109/CCBD.2016.034.

[48] Ying Luo, Hai Zhao, and Junlang Zhan. "Named Entity Recognition Only from Word Embeddings." In: (2019). arXiv: 1909.00164. URL: http://arxiv.org/abs/1909.00164.

[49] Deborah Magalhães, Rodrigo N. Calheiros, Rajkumar Buyya, and Danielo G. Gomes. "Workload Modeling for Resource Usage Analysis and Simulation in Cloud Computing." In: *Comput. Electr. Eng.* 47.C (Oct. 2015), pp. 69–81. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2015.08.016. URL: https://doi.org/10.1016/j.compeleceng.2015.08.016.

[50] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. "Improving Spark Application Throughput Via Memory Aware Task Co-location: A Mixture of Experts Approach." In: (2017). arXiv: 1710.00610. URL: http://arxiv.org/abs/1710.00610.

[51] Xiangrui Meng et al. "MLlib: Machine Learning in Apache Spark." In: 17 (2015), pp. 1–7. ISSN: 1532-4435. DOI: 10.1145/2882903.2912565. arXiv: 1505.06807. URL: http://arxiv.org/abs/1505.06807.

[52] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Distributed Representations of Words and Phrases and their Compositionality arXiv : 1310 . 4546v1 [cs . CL] 16 Oct 2013." In: *arXiv preprint arXiv:1310.4546* (2013), pp. 1–9. arXiv: arXiv:1310.4546v1.

[53] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. "ESP: A Machine Learning Approach to Predicting Application Interference." In: *Proceedings - 2017 IEEE International Conference on Autonomic Computing, ICAC 2017* (2017), pp. 125–134. DOI: 10.1109/ICAC.2017.29.

[54] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. "Online Phase Detection Algorithms." In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 111–123. ISBN: 0-7695-2499-0. DOI: 10.1109/CGO.2006.26. URL: http://dx.doi.org/10.1109/CGO.2006.26.

[55] Nhan Nguyen, Mohammad Maifi Hasan Khan, and Kewen Wang. "Towards Automatic Tuning of Apache Spark Configuration." In: *IEEE International Conference on Cloud Computing, CLOUD* 2018-July (2018), pp. 417–425. ISSN: 21596190. DOI: 10.1109/CLOUD.2018.00059.

[56] Nhan Nguyen, Mohammad Maifi Hasan Khan, Yusuf Albayram, and Kewen Wang. "Understanding the Influence of Configuration Settings: An Execution Model-Driven Framework for Apache Spark Platform." In: *IEEE International Conference on Cloud Computing, CLOUD* 2017-June (2017), pp. 802–807. ISSN: 21596190. DOI: 10.1109/CLOUD.2017.119.

[57] Michael Otte and Scott Richardson. "An HMM Applied to Semi-Online Program Phase Analysis University of Colorado at Boulder Technical Report CU-CS-1034-07 An HMM Applied to Semi-Online Program Phase Analysis." In: (2007).

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch." In: *NIPS-W*. 2017.

[59] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830.

[60] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. "Spark parameter tuning via trial-and-error." In: *Advances in Intelligent Systems and Computing* 529 (2017), pp. 226–237. ISSN: 21945357. DOI: 10.1007/978-3-319-47898-2_24. arXiv: 1607.07348.

[61] Pankaj Pipada, Achintya Kundu, K Gopinath, Chiranjib Bhattacharyya, Sai Susarla, and P C Nagesh. "LoadIQ: Learning to Identify Workload Phases from a Live Storage Trace." In: *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems* June (2012).

[62] Jorda Polo, Yolanda Becerra, David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. "Deadline-Based MapReduce Workload Management." In: *IEEE Trans. Network and Service Management* 10.2 (2013), pp. 231–244. DOI: 10.1109/TNSM.2012.122112.110163. URL: http://dx.doi.org/10.1109/TNSM.2012.122112.110163.

[63] David Buchaca Prats, Josep Lluis Berral, and David Carrera. "Automatic Generation of Workload Profiles Using Unsupervised Learning Pipelines." In: *IEEE Transactions on Network and Service Management* 15.1 (2018), pp. 142–155. ISSN: 19324537. DOI: 10.1109/TNSM.2017.2786047.

[64] Lawrence R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286. ISSN: 15582256. DOI: 10.1109/5.18626.

[65] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. "Stay-Away , protecting sensitive applications from performance interference." In: *Proceedings of the 15th International Middleware Conference on - Middleware '14* December (2014), pp. 301–312. DOI: 10.1145/2663165.2663327. URL: http://dl.acm.org/citation.cfm?doid=2663165.2663327.

[66] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and dynamicity of clouds at scale." In: *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12* (2012), pp. 1–13. ISSN: 0163-5964. DOI: 10.1145/2391229.2391236. URL: http://dl.acm.org/citation.cfm?doid=2391229.2391236.

[67] A. Rosà, W. Binder, L. Y. Chen, M. Gribaudo, and G. Serazzi. "ParSim: A Tool for Workload Modeling and Reproduction of Parallel Applications." In: *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*. 2014, pp. 494–497. DOI: 10.1109/MASCOTS.2014.71.

[68] Krzysztof Rzadca and et al. "Autopilot: Workload Autoscaling at Google." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ?20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387524. URL: https://doi.org/10.1145/3342195.3387524.

[69] Claude Sammut and Geoffrey I. Webb, eds. *Baum-Welch Algorithm. Encyclopedia of Machine Learning*. Springer, 2010, p. 74. ISBN: 978-0-387-30768-8. DOI: 10.1007/978-0-387-30164-8. URL: http://dx.doi.org/10.1007/978-0-387-30164-8.

[70] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando De Freitas. "Taking the human out of the loop: A review of Bayesian optimization." In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. ISSN: 00189219. DOI: 10.1109/JPROC.2015.2494218.

[71] Padhraic Smyth. "Clustering sequences with hidden Markov models." In: *Advances in Neural Information Processing Systems* (1997), pp. 648–654. ISSN: 10495258.

[72] *"Spark-perf: Performance tests for Apache Spark"*. 2020. URL: https://github.com/databricks/spark-perf.

[73] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks." In: (2014), pp. 1–9. arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.

[74] S. J. Tarsa, A. P. Kumar, and H. T. Kung. "Workload prediction for adaptive power scaling using deep learning." In: *2014 IEEE International Conference on IC Design Technology*. 2014, pp. 1–5. DOI: 10.1109/ICICDT.2014.6838580.

[75] Stephen J Tarsa, Amit P Kumar, and H T Kung. "Workload prediction for adaptive power scaling using deep learning." In: *2014 IEEE International Conference on IC Design & Technology* (2014), pp. 1–5. DOI: 10.1109/ICICDT.2014.6838580. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6838580.

[76] Graham W. Taylor and Geoffrey E. Hinton. "Factored conditional restricted Boltzmann Machines for modeling motion style." In: *Proceedings of the 26th International Conference on Machine Learning (ICML 09)* (2009), pp. 1025–1032. ISSN: 1520510X. DOI: 10.1145/1553374.1553505. URL: http://dl.acm.org/citation.cfm?id=1553374.1553505.

[77] Graham W. Taylor, Geoffrey E Hinton, and Sam T. Roweis. "Modeling Human Motion Using Binary Latent Variables." In: *Advances in Neural Information Processing Systems 19*. Ed. by P. B. Schölkopf, J. C. Platt, and T. Hoffman. MIT Press, 2007, pp. 1345–1352. URL: http://papers.nips.cc/paper/3078-modeling-human-motion-using-binary-latent-variables.pdf.

[78] R. Thonangi, V. Thummala, and S. Babu. "Finding Good Configurations in High-Dimensional Spaces: Doing More with Less." In: *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*. 2008, pp. 1–10. DOI: 10.1109/MASCOT.2008.4770581.

[79] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. "Data Warehousing and Analytics Infrastructure at Facebook." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 1013–1020. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807278. URL: http://doi.acm.org/10.1145/1807167.1807278.

[80] Tijmen Tieleman. "Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient." In: *Proceedings of the 25th International Conference on Machine Learning* 307 (2008), p. 7. DOI: 10.1145/1390156.1390290.

[81] Ruben Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguade, Jesus Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. "Spark deployment and performance evaluation on the MareNostrum supercomputer." In:

Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015 (2015), pp. 299–306. DOI: 10.1109/BigData.2015.7363768.

[82] Ruben Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguade, Jesus Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. "Spark deployment and performance evaluation on the MareNostrum supercomputer." In: *Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015* (2015), pp. 299–306. DOI: 10.1109/BigData.2015.7363768.

[83] Jérôme Tubiana, Simona Cocco, and Rémi Monasson. "Learning Compositional Representations of Interacting Systems with Restricted Boltzmann Machines: Comparative Study of Lattice Proteins." In: *Neural computation* 31.8 (2019), pp. 1671–1717. ISSN: 1530888X. DOI: 10.1162/neco_a_01210. arXiv: 1902.06495.

[84] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments." In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. Karlsruhe, Germany: ACM, 2011, pp. 235–244. ISBN: 978-1-4503-0607-2. DOI: 10.1145/1998582.1998637. URL: http://doi.acm.org/10.1145/1998582.1998637.

[85] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. "ARIA: Automatic resource inference and allocation for mapreduce environments." In: *Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011 and Co-located Workshops* (2011), pp. 235–244. DOI: 10.1145/1998582.1998637.

[86] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. "Show and tell: A neural image caption generator." In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 07-12-June (2015), pp. 3156–3164. ISSN: 10636919. DOI: 10.1109/CVPR.2015.7298935. arXiv: 1411.4555.

[87] Guolu Wang, Jungang Xu, and Ben He. "A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning." In: *Proceedings - 18th IEEE International Conference on High Performance Computing and Communications, 14th IEEE International Conference on Smart City and 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016*. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 586–593. ISBN: 9781509042968. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0088.

[88] Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen, and Swapna Gokhale. "Modeling interference for apache Spark jobs." In: *IEEE International Conference on Cloud Computing, CLOUD* (2017), pp. 423–431. ISSN: 21596190. DOI: 10.1109/CLOUD.2016.61.

[89] Michael R. Wyatt, Stephen Herbein, Todd Gamblin, Adam Moody, Dong H. Ahn, and Michela Taufer. "PRIONN: Predicting Runtime and IO using Neural Networks." In: *ICPP '18 Proceedings of the 47th International Conference on Parallel Processing* ls (2018), pp. 1–12. ISSN: 0022-3654. DOI: 10.1145/3225058.3225091. URL: http://dl.acm.org/citation.cfm?doid=3225058.3225091.

[90] Lexiang Ye and Eamonn Keogh. "Time series shapelets: A new primitive for data mining." In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2009), pp. 947–955. DOI: 10.1145/1557019.1557122.

[91] Tao Ye and Shivkumar Kalyanaraman. "A Recursive Random Search Algorithm For Large-Scale." In: (2003), pp. 196–205.

[92] Zhibin Yu, Zhendong Bei, and Xuehai Qian. "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing." In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 564–577. ISSN: 15232867. DOI: 10.1145/3173162.3173187.

[93] Matthew D Zeiler, Graham W Taylor, Nikolaus F Troje, and Geoffrey E Hinton. "Modeling pigeon behaviour using a Conditional Restricted Boltzmann Machine." In: *European Symposium on Artificial Neural Networks ESANN2009* (2009).

[94] Weishan Zhang, Bo Li, Dehai Zhao, Faming Gong, and Qinghua Lu. "Workload prediction for cloud cluster using a recurrent neural network." In: *Proceedings - 2016 International Conference on Identification, Information and Knowledge in the Internet of Things, IIKI 2016* 2018-Janua.66 (2018), pp. 104–109. DOI: 10.1109/IIKI.2016.39.

[95] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. "JustRunIt: Experiment-based Management of Virtualized Data Centers." In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USENIX'09. San Diego, California: USENIX Association, 2009, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1855807.1855825.

DECLARATION

Put your declaration here.

Barcelona, October 2020

David Buchaca Prats