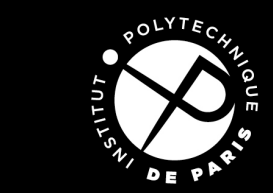
Neural-Based Modeling for Performance Tuning of Cloud Data Analytics

Khaled Zaouk



INSTITUT POLYTECHNIQUE DE PARIS

ÉCOLE POLYTECHNIQUE

IP PARIS

NNT : 2021IPPAX016

Thèse de doctorat

Neural-Based Modeling for Performance Tuning of Cloud Data Analytics

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'Ecole polytechnique

École doctorale n°626 Ecole Doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 11 mars 2021, par

KHALED ZAOUK

Composition du Jury :

Yanlei Diao Professeur, Ecole Polytechnique	Directeur
Alexandre Gramfort Directeur de recherche, INRIA	Examinateur
Florent Masseglia Directeur de recherche, INRIA	Rapporteur (Président)
Paolo Papotti Professeur aggrégé, EURECOM	Rapporteur
Marc Shapiro Directeur de recherche émerite, INRIA	Examinateur

RÉSUMÉ

Les avancées récentes des infrastructures cloud et l'omniprésence des offres de ressouces computationelles de genre "plug and play" ont abouti à un acroissement de la demande des ressources cloud par les entreprieses qui tournent des tâches analytiques sur des données massives. La plupart des fournisseures cloud, cependant, offrent les ressources computationnelles en mode "pay as you go" ou bien en mode prépayé, ce qui prend en compte la durée de la réservation des machines cloud sans aucune garantie sur les performances des jobs qui y tournent. Ceci oblige l'utilisateur à calibrer manuellement ces performances et aboutit souvent à une sous-utilisation des ressources réservées puisque les utilisateurs cloud ont toujours tendance à surprovisionner les ressources. D'où la nécessité de modifier le business model des fournisseurs cloud d'une facturation basée sur la disponibilité des machines vers une facturation qui se repose sur un accord sur les performances obtenues. Alors la modélisation des différentes objectives devient assez importante à la fois pour les utilisateurs ainsi que les fournisseurs cloud. Une bonne modélisation permettra aux fournisseurs cloud de colocaliser des jobs de plusieurs utilisateurs sur l'infrastructure cloud ce qui permettra d'offrir des machines computationnelles à des prix plus bas tout en satisfaisant les performances requises. Ceci aussi permettra de converger vers un mode d'usage du cloud plus écologique tout en maximisant l'utilisation des ressources des serveurs physiques au lieu de tourner de nouveaux serveurs physiques avec une utilisation réduite.

Ça reste cependant non évident de modéliser les performances des jobs qui tournent sur le cloud pour plusieurs raisons: (1) La diversités des jobs qui peuvent comprendre des requêtes de type SQL, des tâches de machine learning, des pipelines d'analytiques mixtes, etc. (2) La compléxité des systèmes d'execution distribués qui comprennent des comportements dynamiques divers en matière de CPU, IO, mémoire, shuffling, etc. (3) Une pléthore de noeuds de contrôle dans les systèmes d'execution distribués. Ces noeuds ont souvent des interactions qui sont complexes aboutissant à des effets différents sur les perfomances bout à bout. Dans cette thèse nous aborderons ces défis en proposant une approche "boîte noire" pour apprendre de façon automatique les modèles de performance à partir de métriques collectées durant l'exécution des jobs. On construit ainsi un modèle aussi complexe que nécessaire pour chaque environnement d'exécution et on le calibre pour pour cet environnement. On explore alors pour cette modélisation les techniques d'apprentissage les plus récentes, capables ainsi de modéliser l'interaction complexe entre les différents noeuds.

Plus précisément, dans cette thèse nous tirons parti des avancées dans les techniques d'apprentissage automatique et nous proposerons un optimiseur unifié qui automatise le tuning des jobs d'analyses de données sur Spark comme système distribué. On focalise principalement sur l'entraînement de modèles prédictifs qui prennent en compte les métriques collectées durant l'exécution des jobs sous forme d'une boîte noire. Ces modèles là sont utilisés par l'optimiseur qui va chercher dans l'espace des configurations possibles avant de recommander une configuration qui satisfait la meilleur performance. Nos méthodes couvrent principalement deux familles de techniques: (1) Les approches à bases de systèmes de recommendation (2) Les approches à base de l'apprentissage des représentations (Representation learning). Nous formulerons notre problème de tuning sous forme d'un système de recommendation qui recommande de meilleurs configurations pour les jobs dans le cloud, et ceci basé sur une

performance observée avec une configuration initiale. On présente une diversité de techniques de recommendations, et on focalise en particulier sur la factorisation matricielle qui modélise l'intéraction entre un job et une configuration particulière sous forme d'un produit scalaire. On présente aussi deux techniques d'embeddings qui peuvent modéliser cette intéraction avec un réseau de neurones. Les approches basés sur les systèmes de recommandation apprennent implicitement un vecteur qui représente chacun des jobs, uniquement à partir de l'objective qu'on souhaite optimiser. Ces méthodes aussi exigent un apprentissage de façon incrémental du modèle, une fois qu'un nouveau job est admis par l'optimiseur. Pour surmonter les limitations des approches basées sur les sytèmes de recommandation, on aborde une deuxième famille de techniques, celle à base du representation learning afin d'apprendre de façon explicite les embeddings des jobs tout en utilisant toutes les métriques collectées et non seulement l'objective à optimiser. Ceci permet des prédictions plus précises sous forme "zero-shot" (sans aucun apprentissage incrémental) et permet à l'optimiseur de recommender des configurations au delà de celles déjà explorées dans les données d'apprentissage.

En résumé, cette thèse apporte les contributions suivantes:

- On introduit un ensemble d'exigences réelles inspirées des services cloud pour guider la conception et l'architecture de notre optimiseur. On présente ensuite un ensemble de propriétés afin de pouvoir caractériser les différents jobs à partir des embeddings extraits: indépendence entre les facteurs génératifs, invariance par rapport à la configuration (et similarité en tant que relaxation de cette propriété), ainsi que la capacité de reconstruction des métriques à partir de l'embedding appris.
- Nous formulerons notre problème de tuning sous forme d'un système de recommendation et explore deux paradigmes de méthodes de recommendation: les architectures à base de filtrage collaboratif, et celles à base de contenu. On explique les limitations des différentes approches et présenterons des résultats au niveau de la modélisation qui comparent une architecture d'embedding à base de réseaux de neurones qu'on propose ainsi que Paragon [20] et Quasar [21] (comme outils de factorisation de matrices) de la litérature des systèmes.
- On introduit des architectures de representation learning de 3 familles de techniques: (1) architectures à base d'encodeurs/décodeurs (2) réseaux de neurones siamois (siamese neural networks) (3) une famille qui combine les deux premières dans des architectures hybrides. Ces techniques extraient de façon explicite des embeddings à partir des métriques collectées pour ensuite les mettre en entrée d'un réseau de neurone dédié à la tâche de regression sur la durée d'execution des jobs cloud. On couvre des auto-encodeurs deterministes ainsi que des auto-encodeurs génératifs et propose des extensions pour satisfaire les différentes propriétés désirées précedemment évoquées. On explique aussi pourquoi les réseaux de neurones siamois (siamese neural networks) sont particulièrement intéressant une fois qu'un job est admis avec une configuration arbitraire et on entraîne ces architectures en utilisant deux types de fonction de coût: une fonction de coût type "Triplet" et une fonction de coût type "soft nearest neighbor".
- On propose une extension d'un benchmark de workloads type streaming et on échantillonne des traces de ces workloads ainsi que d'autres workloads du benchmark TPCx-bb [59]. On utilise ces traces d'exécution dans l'évaluation des différentes techniques de modélisation. On fournit ces traces ainsi que notre code de modélisation sous le lien: `https://github.com/udao-modeling/code`
- On fournit un comparatif détaillé des différentes techniques de modélisation. Nos resultats

montre que les architectures à base systèmes de recommendation constituent un très bon choix de modélisation si on peut garantir que le nouveau workload a été profilé avec une configuration déjà utilisée avec d'autres workloads. Par contre, ces architectures restent incapables de recommander des configurations au delà de celles déjà vues dans les données d'apprentissage. Elles exigent aussi une phase d'apprentissage incrémental avant de pouvoir fournir des prédictions pour les nouveaux jobs. On montre d'autre part que les architectures types representation learning surmontent ces deux problèmes et modélisent le délai d'exécution des jobs Spark avec une erreur aux alentours de 10% pour les deux benchmarks.

- On fournit un comparatif bout à bout avec Ottertune [61], l'état de l'art dans la modélisation des performances des bases de données. On montre que les configurations recommandées par notre optimiseur sont meilleures que celles recommandées par Ottertune. En effet, notre optimiseur améliore le délai d'exécution des jobs Spark de 52.4% sur les deux benchmarks alors que Ottertune les améliore seulement de 35.96% par rapport au benchmark streaming et 43.19% par rapport au benchmark TPCx-BB quand les jobs sont admis avec une configuration arbitraire et que les ressources computationnelles à disposition de l'optimiseur sont élastiques. D'autre part, lorsque les ressources à disposition de l'optimiseur sont fixées, notre optimiseur améliore le délai d'exécution des jobs Spark par seulement 30.68% et 7.48% respectivement pour les benchmarks streaming et TPCx-BB alors que Ottertune reste incapable de fournir des meilleurs configurations en moyenne. En effet, les nouvelles configurations recommandées par Ottertune sont en moyenne 41.01% pire que les configurations initiales lorsqu'il s'agit du benchmark de streaming. Celles recommandées sur les jobs du benchmark TPCx-BB améliorent le délai d'exécution de seulement de 0.13%.

ABSTRACT

Recent advances in the cloud infrastructure and the widespread offering of plug and play cloud instances led to a growing demand for cloud resources from enterprise businesses that need to run critical analytical tasks on voluminous datasets. However, most cloud providers lend the computing infrastructure within a pay as you go, or prepaid scheme that accounts for the duration of the booking without any guarantees on the performance objectives of workloads running in the cloud. This usually leads to tedious manual performance tuning by the user, as well as underutilization of the resources as users tend to overprovision them in order to run their workloads without violating their performance goals. Hence, there's a growing need to shift the business model of cloud providers from machine availability to service level objectives. As such, modeling service level objectives of cloud applications becomes crucial to both the cloud providers as well as the users. It paves the way for cloud providers to colocate workloads of multiple users on the cloud infrastructure, and allows them to offer instances at reduced prices while meeting user performance goals. Such modeling can also help achieve "green computing" by maximizing the resource utilization of physical servers already turned on instead of running more physical servers at reduced utilization.

It remains however nontrivial to model the performance of cloud workloads due to (1) diversity of workloads, including SQL queries, machine learning tasks, etc. mixed in analytics pipelines; (2) complexity of the runtime systems, including diverse dynamic CPU, IO, memory, data shuffling behaviors; (3) a plethora of system knobs with complex interactions and different effects on the end performance. This thesis addresses these challenges by taking the following approaches. First, it takes a blackbox approach to automatically learn the performance models by leveraging runtime metrics. Second, it builds a model for each specific runtime environment, and tunes the model as complex as needed for that computing environment. Third, it further explores advanced machine learning techniques as a modeling tool due to their ability to model the complex interaction between the different knobs.

More specifically, in this thesis we leverage recent advances in machine learning and propose a unified data analytics optimizer that automates the tuning of Spark cloud workloads. We focus in particular on building accurate predictive models that leverage runtime metrics collected during the execution of Spark workloads in a blackbox manner. These models are used by the optimizer to search through the space of configurations before recommending a configuration that achieves the best performance objective. Our methods cover two main families of techniques: (1) Recommender based approaches (2) Representation learning based approaches. We first cast our modeling problem into a recommender systems framework that suggests better configurations for cloud workloads based on profiled performance with some initial configuration. We present a broad perspective of recommendation techniques and focus particularly on matrix factorization that models the interaction between a workload and a particular knob configuration using a dot product. We also introduce two embeddings approaches that model this interaction using a neural network. The recommender based approaches *implicitly* learn a vector representation of each workload using solely the objective to model and require incremental training upon the admission of a new workload. To overcome the limitations of recommender approaches, we employ a second family of techniques, namely,

representation learning techniques, to *explicitly* learn different workload embeddings by leveraging the full runtime metrics. This enables more accurate predictions in a zero-shot learning scheme (i.e., without incremental training) and supports recommending configurations beyond those already observed within training data.

In summary, this thesis makes the following contributions:

- We introduce a set of real world requirements from cloud services that inspire the design of our performance tuning system architecture. We then outline a set of properties for encodings we extract from runtime traces in order to characterize the different workloads: *independence* between generating factors, *invariance* with respect to the generating configuration (and *similarity* as a relaxation of this property), and the *reconstruction* ability of the learned encoding.
- We cast our tuning problem into a recommender systems framework and cover two paradigms of collaborative and content based recommender architectures. We also explain the limitations of the different existing approaches and provide comparative modeling results between a neural embedding architecture we propose as well as Paragon [20] and Quasar [21] (as matrix factorization tools) from the systems literature.
- We introduce representation learning architectures from three families of techniques: (1) encoder/decoder architectures; (2) siamese neural networks; and (3) a new family that combines the first two in hybrid architectures. These representation learning based techniques explicitly extract encodings from runtime traces before feeding them to a neural network dedicated for the end regression task on the runtime latency. We cover deterministic and generative auto-encoders and propose extensions of them in order to satisfy different desired encoding properties. We also explain why the siamese neural networks are particularly interesting when a job is admitted with an arbitrary configuration and we train these architectures using two types of losses: a triplet loss and a soft nearest neighbor loss.
- We extend a previous benchmark of streaming workloads and sample traces from these workloads as well as workloads from the TPCx-BB [59] benchmark. We use these traces in order to evaluate the different modeling techniques. We make these traces available at `https://github.com/udao-modeling/code`.
- We provide comparative results between different modeling techniques. Our results show that recommender architectures are a good modeling choice if workloads are profiled with a previously seen configuration. They remain incapable of generalizing to predict the performances over unseen configurations and require incremental training prior to prediction. On the other hand, representation learning based techniques overcome these two main problems and achieve errors around 10% on both benchmarks.
- We provide end-to-end comparative results with Ottertune [61], a state of the art tuning tool, and we demonstrate the superiority of the performance of configurations recommended by our optimizer over those recommended by Ottertune. For instance, our optimizer achieves latency improvements around 52.4% on both benchmarks while Ottertune achieves latency improvements of 35.96% on the streaming benchmark and 43.19% on the TPCx-bb benchmark when jobs are admitted with an arbitrary configuration and our optimizer is allowed to scale out resources. In addition, in the setting of limited resources, our optimizer achieves a latency improvement of 30.68% and 7.48% on the streaming and TPCx-bb benchmarks respectively while Ottertune fails to provide better configurations.

For instance, the configurations recommended by Ottertune yield latencies that are 41.01% worse on average than the initial configurations (negative average latency improvement) on the streaming benchmark, and yield a latency improvement of only 0.13% on the TPCx-bb benchmark.

ACKNOWLEDGMENTS

Now It's been already a while since I've defended my thesis that I could gather my thoughts to write this page... I would like to start by thanking my advisor Yanlei for having given me the opportunity to pursue this Ph.D. right after my master thesis and for her time advising this Ph.D. work.

I would like to deeply thank Florent Masseglia and Paolo Papotti for accepting to be the rapporteurs of my thesis manuscript, for taking the time to read it and give very postive comments on its content. I would also like to deeply thank Alexandre Gramfort and Marc Shapiro for the time they provided for both the mid-term discussion of my Ph.D. as well as for accepting to be examiners during the day of the thesis defense. Finally, a sincere thanks goes to Ioana Manolescu and Benjamin Doerr for taking the time to listen and provide the required support towards the successful completion of this work.

Throughout my Ph.D. years I have also met amazing people that reshaped my thinking and view of the world. I start with the people with whom I spent most of my time within our research team, the *Cedar* team. I'd like to thank in particular Tayeb, Mirjana and Maxime for lifting up the spirits in difficult days and for always providing encouragement throughout this journey. I'm also thinking of the amazing people with whom I shared the same office and with whom I'd loved to spend more time: Alexandre and our discussions while speed writing, Arnaud and the pauses café sans café, Vincent and the discussions about sports and Qi and the discussions about life in China. I'd like to thank Pawel and Arnab for the discussions that we had around lunch about cultures, languages, immigration, and other... I'm also thinking of Luciano and Enhui with whom we had less frequent but very interesting discussions around lunch and Félix with his very energetic morning brainstormings and discussions. I'd like to thank Fei for the early discussions in the beginning of my Ph.D, and Chenghao who helped in generating trace datasets for this project. I'm very happy for having met Oana who taught me about statistical tests and with whom I had discussions about their usage within the research community. I would have loved to spend more time and learn more from the new team members: Angelos, Nelly, Yamen and Fanzhi. I'd also like to thank the people from *CSX* with whom I enjoyed playing volleyball during lunch breaks: Didier, Michel, Mathias, Thomas, Vlad, Sylvie... Finally, a sincere thanks goes to Jessica for for providing the administrative support helping with all paperwork throughout my Ph.D. years.

This Ph.D. work has been possible thanks to all of my teachers starting with those who taught me in first place how to write at my high school *Rawat El Fayhaa*, to other teachers I met throughout my Bachelor at the *Lebanese University*, up to my masters teachers at both *Télécom Paris* and *Ecole Polytechnique.* To all of them, a big thank you!

Finally, I'd like to finish by thanking both of my parents for their invaluable support and to whom I dedicate this work. I would also like to thank my beloved sisters for always trusting in my abilities and for the care they brought to me before I left *Lebanon.* A sincere thanks goes to my friends in Paris who always encouraged me throughout this journey... I'd finally like to thank my uncle Rachid who first taught me how to write my very first *program* at the age of 10, and my cousin Kareem at whom I fired a bunch of programming questions every time we met while I was still a teenager...

Contents

List of Figures

List of Tables

1 Introduction

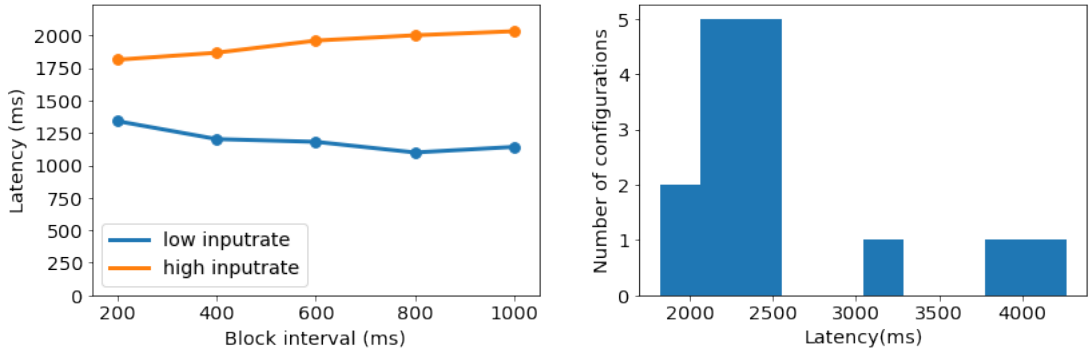
Today's big data analytics systems are best effort only. While scalable analytics systems such as Hadoop, Spark [66], Flink [22], Google Dataflow [9], and Scope [70] have gained wide adoption, most of them lack the ability to take user performance goals and budgetary constraints, collectively referred to as "objectives", and automatically configure an analytic job to achieve those objectives. Most cloud providers lend the computing infrastructure within a pay as you go or an annual prepaid scheme that accounts for the duration of the booking, without any guarantees on the performance objectives of user workloads. In practice this often leads to tedious manual performance tuning by the user.

Consider an analytics user who aims to run a mix of SQL queries and machine learning tasks using Spark over instances from AWS. The user needs to first choose from about 60 EC2 instance types that differ in the number of cores and memory available. This choice significantly affects the performances achieved in analytics and depends on the workload, performance goals, and budgetary constraints of the user, and is often made as a guess by the user. After the cloud instance is chosen and to gain good performance, the user may still need to tune many parameters of the runtime system, collectively referred to as a "job configuration", for each analytic task. Take Spark for example. The runtime parameters include *parallelism* (for reduce-style transformations), *Rdd compression* (boolean), *Memory per executor*, *Memory fraction* (of heap space), *Batch interval* (the size of each minibatch) and *Block interval* (the size of data handled by each map process) in the streaming setting, to name a few.

Choosing the right job configuration to meet user performance goals is a difficult task. For example, Figure 1(a) shows that in Spark streaming, the effect of the block interval on latency exhibits opposite trends under low and high input rates: a larger block interval size reduces latency when the input rate is low, and increases latency when the input rate is high. Figure 1(b) shows that under the fixed input rate and batch size, good and bad configurations of parallelism give very different latencies. Although Spark recommends setting parallelism to be 2-3 times the total number of cores, parallelism equals to the number of cores achieves the lowest latency (2000ms) in this cluster settings, while a higher value in the recommended range achieves a worse latency (4000ms). The complexity of finding the right configuration grows quickly, e.g., given 10 or more parameters, due to joint effect and correlation between the different parameters. Our observation is consistent with a recent study [49] that shows that for HiveQL queries alone, even expert engineers most time could not make the right choice between two cluster options and their estimated runtime ranged from 20x under-estimation to 5x over-estimation.

Searching for the right configuration that suits the user objectives is largely a trial-and-error process today, sometimes involving the change to a larger cloud instance. Even if the user could find the right cluster and configuration through manual tuning, one day the user might decide to slightly change his workload, e.g., by operating on a different window size for stream analytics. The previous configuration that the user manually found might not achieve as low latency as that of the previous window size. In this case, the user is left with no choice other than repeating the tuning manually.

As can be seen, the lack of system support for configuring cloud analytics to meet user objectives leads to tedious manual tuning by the user. In practice, it also leads to underuti-



(a) Opposite trends in latency under different input rates

(b) Wide range of latency values

Figure 1: Complexity of parameter tuning.

lization of the resources as users tend to overprovision them in order to run their workloads without violating their performance goals. Such overprovisioning comes at the cost of the user, and also incurs a waste of overall system resources. Shifting the business model of cloud providers from machine availability to service level objectives will allow service providers to colocate workloads of multiple users on the cloud infrastructure, and offer instances at reduced prices while meeting user performance goals. This shift can also help achieve the broad vision of "green computing" by maximizing the resource utilization of physical servers already turned on instead of running more physical servers at reduced utilization.

In this work, we aim to take a step further towards building a **unified data analytics optimizer** that can determine the job configuration in an automated manner based on user objectives regarding latency, throughput, processing cost, etc. This new optimizer aims to support broad analytics tasks including SQL queries, machine learning tasks, graph analytics, etc., in the general paradigm of dataflow programs. Towards this vision, we put more focus in this thesis on **modeling service level objectives**, i.e., building and serving accurate models that the optimizer can use for recommending a better configuration for the submitted workload. We detail in the next subsection the technical challenges for building accurate models.

1.1 Technical challenges

It is nontrivial to model the performance of cloud workloads due to (1) diversity of workloads, including SQL queries, machine learning (ML) tasks, etc. mixed in analytics pipelines; (2) complexity of the runtime systems, including diverse dynamic CPU, IO, memory, data shuffling behaviors; (3) a plethora of system knobs with complex interactions and different effects on the end performance. We detail below each of these three challenges.

1. Diversity of workloads. Recent work in the database community [41, 43, 49] has addressed the problem of performance modeling of SQL queries in Relational Database Management Systems (RDBMS) by taking into account SQL query plans. Our work takes a step towards modeling a more diverse set of workloads including native SQL queries, SQL queries that make intensive use of user-defined functions (UDFs) and ML queries. Supporting

these diverse workloads is challenging because there isn't a fixed set of operators within the distributed analytics system in contrast to SQL workloads that rely on relational algebra. For instance, the cloud practitioner can import an arbitrary library into the distributed system, and this library can be written in any programming language. Hence, we rely in this work on runtime metrics collected during the execution of workloads so that we model the performance in a blackbox manner. In order to characterize the workloads, we leverage recent advances in deep neural networks, which are known for the power of representation learning. This allows us to extract a numerical representation of each workload from runtime traces to describe its characteristics even if the dataflow program is given as a blackbox.

2. Complexity of the runtime systems. The cost model has to characterize any objective of interest to the user in a dynamic cluster computing environment where the observed performance depends on the hardware, the software, and system behaviors involving CPU, IO, shuffling, stragglers, fault tolerance, etc. Static models often fail to adapt to new user objectives, new workloads, a different runtime system (or a new version of it), etc. To do so, our work takes a new approach that learns a model for each user objective in the same computing environment as the user job is being executed, which we refer to as *in-situ modeling*. Given the complexity of system behaviors in scalable analytics, we explore neural networks as a modeling tool for their expressiveness power: when the cost model of each user objective is viewed as a continuous function, every such function can be approximated arbitrarily closely by a three-layer neural network [30]. Such expressive power frees us from worrying about if assumptions such as linear functions hold or not. We can further adapt the expressive power, through hyper-parameter tuning, to learn models as complex as necessary for a particular computing environment.

3. A plethora of knobs with complex interactions. The distributed analytics systems usually have a large set of knobs that have complex interactions and that jointly affect the performance of the distributed job. Without proper tuning, these knobs usually yield a poor performance of the deployed job on cloud infrastructure. In our work, we first select the most important knobs to tune, and then model the performance objective as a function of the selected knobs alongside the extracted numerical representation of each workload. We use ML methods to automatically capture this interaction while modeling the performance objective.

1.2 Contributions and most significant results achieved

In this thesis, we address the aforementioned challenges by bringing advanced machine learning (ML) techniques to bear on the process of performance modeling of cloud analytics. In particular, we propose modeling methods from two families of ML techniques: (1) Recommender based approaches and (2) Representation learning based approaches.

We first cast our modeling problem into a recommender systems framework that suggests better configurations for cloud workloads based on profiled performance with some initial configuration. We present a broad perspective of recommendation techniques and focus particularly on matrix factorization that models the interaction between a workload and a particular knob configuration using a dot product. We also introduce two embeddings approaches that model this interaction using a neural network. The recommender based approaches *implicitly* learn a vector representation of each workload using solely the objective

to model and require incremental training upon the admission of a new workload.

To overcome the limitations of recommender approaches, we explore a second family of techniques, namely, recent representation learning techniques, to *explicitly* learn different workload embeddings by leveraging the full runtime metrics. This enables more accurate predictions in a zero-shot learning scheme (i.e., without incremental training) and supports recommending configurations beyond those already observed within training data.

More specifically, this thesis makes the following contributions:

- We introduce a set of real world requirements from cloud services that inspire the design of our performance tuning system architecture. We then outline a set of properties for encodings we extract from runtime traces in order to characterize the different workloads: *independence* between generating factors, *invariance* with respect to the generating configuration (and *similarity* as a relaxation of this property), and the *reconstruction* ability of the learned encoding.
- We cast our tuning problem into a recommender systems framework and cover two paradigms of collaborative and content based recommender architectures. We also explain the limitations of the different existing approaches and provide comparative modeling results between a neural embedding architecture we propose as well as Paragon [20] and Quasar [21] (as matrix factorization tools) from the systems literature.
- We introduce representation learning architectures from three families of techniques: (1) encoder/decoder architectures; (2) siamese neural networks; and (3) a new family that combines the first two in hybrid architectures. These representation learning based techniques explicitly extract encodings from runtime traces before feeding them to a neural network dedicated for the end regression task on the runtime latency. We cover deterministic and generative auto-encoders and propose extensions of them in order to satisfy different desired encoding properties. We also explain why the siamese neural networks are particularly interesting when a job is admitted with an arbitrary configuration and we train these architectures using two types of losses: a triplet loss and a soft nearest neighbor loss.
- We extend a previous benchmark of streaming workloads and sample traces from these workloads as well as workloads from the TPCx-BB [59] benchmark. We use these traces in order to evaluate the different modeling techniques. We make these traces available at `https://github.com/udao-modeling/code`.
- We provide comparative results between different modeling techniques. Our results show that recommender architectures are a good modeling choice if workloads are profiled with a previously seen configuration. They remain incapable of generalizing to predict the performances over unseen configurations and require incremental training prior to prediction. On the other hand, representation learning based techniques overcome these two main problems and achieve errors around 10% on both benchmarks.
- We provide end-to-end comparative results with Ottertune [61], a state of the art tuning tool, and we demonstrate the superiority of the performance of configurations recommended by our optimizer over those recommended by Ottertune. For instance, our optimizer achieves latency improvements around 52.4% on both benchmarks while Ottertune achieves latency improvements of 35.96% on the streaming benchmark and 43.19% on the TPCx-bb benchmark when jobs are admitted with an arbitrary configuration and our optimizer is allowed to scale out resources. In addition, in the setting of limited resources, our

optimizer achieves a latency improvement of 30.68% and 7.48% on the streaming and TPCx-bb benchmarks respectively while Ottertune fails to provide better configurations. For instance, the configurations recommended by Ottertune yield latencies that are 41.01% worse on average than the initial configurations (negative average latency improvement) on the streaming benchmark, and yield a latency improvement of only 0.13% on the TPCx-bb benchmark.

1.3 Outline of the thesis

The remainder of the thesis is organized as follows:

Chapter 2.: This chapter outlines some of the recent related work that has had an impact in terms of tuning cloud workloads.

Chapter 3.: This chapter presents real world requirements and challenges for building a cloud optimizer. These requirements have guided the design of our system which we detail and describe its different components. We also provide in this chapter the notation that we adopt throughout the thesis alongside the formal description of the tuning problem that we address. Finally, we list a set of properties that we find important to assess the quality of different modeling techniques that we survey in this thesis.

Chapter 4.: This chapter explains how configuration recommendation for cloud workloads can be casted into a recommender system framework. It surveys existing techniques widely adopted within recommender systems covering both collaborative based and content based methods and discusses the limitations and potentials of each of these techniques. It also introduces a neural network alternative to dot-product based matrix factorization techniques for modeling the runtime latency and also proposes a hybrid embedding architecture that can be used to recommend configurations beyond those already existing within training data.

Chapter 5.: In this chapter, we first motivate the need for representation learning methods before we introduce another paradigm of modeling methods that make use of a full runtime metrics collected from a submitted workload before recommending a new configuration. We survey auto-encoder based architectures explaining why traditional auto-encoders usually fail to model accurately the runtime latency in our settings. We then introduce a custom architecture that can help in extracting encodings with better disentanglement properties and give a brief overview of variational auto-encoders which perform a similar task. We finally propose to use siamese neural networks for solving our problem, explain two types of loss functions that can be used to train such an architecture and describe some hybrid ways to combine siamese neural networks with auto-encoders.

Chapter 6.: This chapter is dedicated to describing the workloads and traces that are used for evaluating the different modeling techniques from Chapters 4 and 5. We cover in this chapter our contribution which consists of an extension of streaming workloads from a prior work. We then briefly describe batch workloads from the TPCx-BB benchmark on top of which we evaluate this work as well. We also provide in this chapter details regarding the heuristics that we have used for sampling traces from these workloads on our computing infrastructure. We motivate the choice of the different knobs that we select to tune for Spark, and give a brief overview of the metrics that we collect within each runtime trace.

Chapter 7.: This chapter is dedicated for the experiments we conducted in order to compare different modeling techniques. We first outline the experimental setup that is adopted in terms of trace preprocessing, hyper-parameter tuning and implementation details. We start by comparing between the most promising recommender approaches which we introduce in Chapter 4 and we recall the need for representation learning approaches. We then provide comparative results between different representation learning based techniques introduced in Chapter 5 as well as a state of the art tuning tool, baseline methods and a hybrid embedding approach we suggest at the end of Chapter 4. We conduct an ablation study in order to understand how many configurations the cloud optimizer needs to afford to sample during an offline runtime session period. We also conduct a scalability study which reveals interesting insights regarding business critical decisions in terms of training time budget. Finally, we conduct an end-to-end comparison between our best performing technique (triplet) and a state of the art tuning tool.

Chapter 8.: This chapter provides an extended discussion of some of the technical issues encountered throughout the thesis.

Chapter 9.: This chapter summarizes the contributions of this thesis and states some research directions for future work.

So far, this work has led to the following contributions:

- *Zaouk et al., 2020*: **Neural-based Modeling for Performance Tuning of Spark Data Analytics**. Khaled Zaouk, Fei Song, Chenghao Lyu, Yanlei Diao. (To be submitted)
- *Song et al., 2020*: **Spark-based Cloud Data Analytics using Multi-Objective Optimization**. Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, Prashant Shenoy. *ICDE2021*, accepted.
- *Zaouk et al., 2019*: **UDAO: A Next-generation Unified Data Analytics Optimizer**. Khaled Zaouk, Fei Song, Chenghao Lyu, Arnab Sinha, Yanlei Diao, Prashant Shenoy. *PVLDB* 2019, accepted.

2 Related Work

In this chapter we survey related systems that solve similar problems to our problem of tuning cloud based workloads. In §2.1 we first introduce relevant work from the cloud resource management community. Then in §2.2 we explain related work that uses representation learning to characterize workloads. Finally, in §2.3 we cover three related systems from the machine learning (ML) community that solve a similar problem when tuning hyper-parameters of ML models. We discuss in the next subsections the similarities and differences between these systems and our work.

2.1 Learning based cloud resource management systems

Ottertune. Ottertune [61, 68] is a state of the art tuning tool for database management systems (DBMs) published in 2017. It offers a range of ML techniques with the purpose of automating the tedious task of tuning the large database knobs and cutting the expensive cost of manual tuning. We give a detailed description of Ottertune in this section, because it's the main relevant system to which we compare quantitatively our work on both modeling performance and in an end-to-end manner in Chapter 7 (in particular under §7.6).

Ottertune goes beyond previous work in the literature that consists of heuristics and practices for tuning and usually yields non-reusable configurations. The core idea in Ottertune is centered around leveraging runtime traces from past workloads submitted to the DBMS, and training ML models that allow better predictions on new workloads. It combines a set of supervised and unsupervised machine learning techniques in its processing pipeline to serve different purposes ranging from (1) selecting the important knobs it wants to tune, to (2) deciding which metrics are relevant for characterizing the workloads, (3) training predictive models for these retained metrics, and (4) mapping a new workload to a past workload and then training a Gaussian Process (GP) model using profiling data from both the new workload and the mapped workload. Finally, it uses its trained GP model to recommend a better configuration after minimizing a lower-confidence-bound (LCB) acquisition function. In the following we discuss these 4 main modeling components that are at the core of Ottertune's configuration recommendation pipeline:

1. Knob selection In order to determine which knobs to tune, Ottertune applies a Lasso regression [1] on polynomial features generated from different knobs (so as to account for the interaction between different knobs). Then, it applies the Lasso path algorithm on the regression result. We have found that this approach is not convenient for selecting the knobs for several reasons:

(a) *Model discrepancy*: There is a discrepancy between the type of the models used for training the Lasso path algorithm and the type of the final model used for making predictions. For instance, the final regression model that is used within Ottertune is not a linear regression on top of polynomial features, but rather a GP model where the input features are the different knobs (without any polynomial features). This discrepancy also makes it difficult to interpret which features we need to keep or discard if the polynomial combination of features has

[1] Sections 3.4 and 3.8 of Chapter 3 of the book "Elements of statistical learning" [26] give a good explanation of both the Lasso regression and the Lasso path algorithm

for example a good ranking amongst all features while the two features from which this polynomial feature was generated have separately lower contributions. In short, the feature importance should be done and analyzed on the final algorithm used for the regression task not using a different algorithm.
(b) *Single vs global model training*: Ottertune trains a global model when it performs the Lasso knob selection, but when it comes to regression on the end target it trains separate GP models, one per workload. If for some few workloads a particular subset of knobs have an important effect on the regression target, but do not seem to affect the other remaining majority of workloads, then these knobs won't be selected by the algorithm. Hence, a tuning session for a new workload similar to one of the workloads within the particular set, will probably disregard important knobs that may hugely affect its performances.
(c) *Characteristics of sampled traces*: The knob selection procedure depends heavily on the characteristics of the data collected for training such as how many knobs are tuned in total, and how many distinct values per knob are sampled. For instance, increasing the number of knobs to be tuned requires exponentially increasing the number of sampled traces in order to cover the space of knobs in a fair manner. Since it's difficult and expensive to afford for sampling a lot of points for a particular workload, it is unlikely that the sampling procedure will cover the different knobs in a fair manner as soon as the number of knobs increase. The data skew will be directly reflected in an unfair feature selection which will immediately drop the knobs for which only few values have been sampled across the trace dataset

2. Metric selection. Instead of using the full set of metrics collected while profiling workloads, Ottertune first runs factor analysis to get factors representing the contribution of each metric. Then in the space defined by the different factors, it tries to detect the clusters that can be found among the different metrics by applying the k-means clustering algorithm as shown in the diagram within Figure 2. Since the k-means algorithm requires setting the number of clusters k, Ottertune resorts to a heuristic to automatically set this value. Then, after the clustering is done, Ottertune retains one metric per cluster, and this metric is chosen to be the closest to the cluster's centroid. Thus, Ottertune retains in total a number metrics equal to the number of clusters k.

3. Training models Using each of the different k retained metrics, and on top of each training workload, Ottertune trains a GP model using data from available configurations. Hence, if we have a total number n of training workloads and k retained metrics, the total number of models trained by Ottertune is $n * k$. This directly implies a scalability issue as soon as the total number of workloads increases, because in order to maintain good predictive power for models for which it has collected additional traces, Ottertune will have to periodically retrain them. We show later in §4.2 of Chapter 4 how by training different models, Ottertune can be seen as a workload-centered content based recommendation approach.

4. Workload mapping and tuning Upon the admission of a new workload under some initial configurations, Ottertune tries to use the previously trained $n * k$ models in order to predict the k retained metrics for each of the n training workloads with the same input configurations observed for the new workload. It then tries to discretize the values of these predictions (also called binning) before proceeding with mapping the new workload to its nearest neighbor workload. We also explain in Chapter 4 that this mapping step allows us to consider Ottertune as a memory based collaborative approach as well. Once the new workload has been mapped to its nearest neighbor, Ottertune trains a GP model using traces

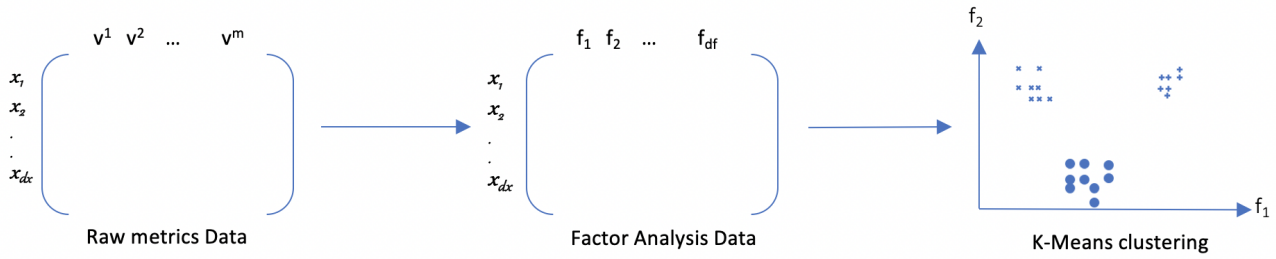


Figure 2: Ottertune's metric selection process.

collected for the mapped workload alongside the new traces observed for the new workload. This GP model defines a predictive distribution that Ottertune uses for recommending a configuration vector $\hat{\mathbf{v}}$ that minimizes a lower-confidence-bound (LCB) acquisition function given by:

$$\hat{\mathbf{v}} = \underset{\mathbf{v}}{\operatorname{argmin}}\, \lambda\tilde{\mu}(\mathbf{v}) - \gamma\tilde{\sigma}(\mathbf{v}) \tag{1}$$

where $\tilde{\mu}(\mathbf{v})$ is the mean of the predicted distribution of target objective (for example runtime latency) over the knobs $\mathbf{v}$ and $\tilde{\sigma}(\mathbf{v})$ is the square root of the variance of the predicted distribution. λ and γ are multiplier coefficients that are constants within Ottertune with default values: $\lambda = 1$ and $\gamma = 3$. These parameters usually balance the exploration against the exploitation over the course of optimization. Ottertune resorts to a simple gradient descent in order to minimize this acquisition function.

This mapping scheme suffers from a major drawback of thrashing the precious information other models built from training workloads maintain. We introduce in Chapter 5 another paradigm for training/inference that makes use of recent advances in representation learning and that focuses on training global models so that all traces from past training workloads can be leveraged while doing inference. We then show later in our experiments in Chapter 7 of this manuscript how our paradigm achieves lower error rates on model prediction than Ottertune. We also show how these better modeling results translate into better latency improvements when it comes to end-to-end performance.

Paragon and Quasar. Paragon [20] and Quasar [21] cast the tuning problem into a recommender system that uses matrix factorization techniques. These systems do not make extensive use of the full trace of runtime metrics, and instead record the runtime latency as the only *interaction* between configurations and different workloads. Upon the submission of a new workload to the system, such techniques are not able to recommend configurations beyond those seen within training data. The reason is that the underlying model learns an embedding vector for the knob configuration alongside learning an embedding vector for the workload, and thus does not allow to predict performances for a custom configuration not present within training data. In our work, we propose instead a neural network based recommender approach that allows exploring configurations beyond those observed within training data by only learning an embedding vector for the workloads while representing configurations by a vector of knob values.

CDBTune. CDBTune [69] was the pioneer in casting the tuning problem into a reinforcement learning framework. Although it leverages the runtime traces of the running workloads in order to tune their knobs in a blackbox manner, it couples both the modeling and optimization steps while formalizing the tuning problem. This is in contrast to our work that has these two steps separated.

It considers the database system as the reinforcement learning agent and defines the state of the agent by the runtime metrics collected for a past configuration. It also defines the actions as a vector describing the amount by which each knob value should be increased or decreased (if the knob value is numerical). The policy that consists of mapping from a state to a particular action is modeled using a neural network. This type of modeling is effective in multi-tenant environments in which an action consisting of setting a particular value of knobs depends on the actual state of the system. In our use case of tuning workloads alone, we don't need to have a state, because no matter what is the current state of the system, setting the knobs to a particular value will yield the same results. Finally, CDBTune doesn't leverage past workload information when a new job is submitted for a tuning session. This makes this approach miss an opportunity of leveraging past traces when a new workload that bears similarity to previously tuned workloads is admitted by the optimizer.

Resource Central. Resource Central [19] is a new system that aims at predicting the runtime latency of virtual machines (VMs) running in private or public clouds. Accurately predicting the lifetime of a VM helps in increasing utilization of physical machines and preventing the exhaustion of physical resources by colocating VMs that are likely to terminate at the same time. Such a modeling can also help the health management system to schedule non-urgent server maintenance without producing VM unavailability or requiring live migration. The core problem that this paper addresses is the scheduling of virtual machines. It consists of a first step of assigning that requires making predictions before the VM can be profiled. The early predictions can be made by taking as input information collected from the VM booking such as the service name, the deployment time, the operating system, the VM size, the VM roles (IaaS or PaaS), etc. This approach tries to classify workloads into interactive or non-interactive workloads, then to further characterize the workloads it runs Fast Fourier Transform (FFT) to extract features that account for periodicity and other temporal features. We do not keep in our work any temporal information since we are interested in modeling the average performances of Spark workloads.

PerfOrator. PerfOrator [49] is a system that focuses on finding the best hardware resources for a given execution plan of a cloud query. It is a whitebox approach that builds a resource-to-performance model after analyzing the query plan through operator overloading and query rewriting. For instance, it collects data sizes at the input and output of each stage of the query execution plan in order to train a non-linear regression model that relates the input to output data size at each stage, and thus enables resource optimization. This is in sharp contrast to our work which resorts to a blackbox modeling of the cloud workloads.

WiseDB. WiseDB [42,44] proposes learning-based techniques for cloud resource management. A decision tree is trained on a set of performance and cost related features collected from minimum cost schedules of sample workloads. Such minimum cost schedules are not available in our problem setting.

Other work. Recent work has used neural networks to predict latency of SQL query

plans [43] or learn from existing query plans to generate better query plans [41]. These methods, however, are applicable to SQL queries only. Finally, performance modeling tools [37, 64] use handcrafted models, hence remain hard to generalize.

2.2 Workload characterization using representation learning

The Dacapo benchmarks. The work in [16] has used PCA as a way to validate diversity in benchmarks for the Java programming language. The main contribution in this work is introducing Java benchmarks and not characterization of different workloads introduced. Nevertheless, this work uses PCA as a sanity check to visualize different modules within the benchmark.

This work outlines that diversity in workloads is achieved because the workloads' representations are scattered in the 2-dimensional space spanned by the first two components of the PCA. Hence the benchmark can be adopted by researchers wishing to analyze Java programs. Our work, in contrast, introduces representation learning methods in Chapter 5 in order to learn meaningful encodings for an end regression task that serves to predict the runtime latency of workloads. We also provide comparative results with PCA in Chapter 7.

The study in [16] reveals that trace results are not only sensitive to the benchmarks but also to the underlying architecture, the choice of heap size, and the virtual machine that is used for running the workloads. In our work, we don't need to worry about the architecture because we are running our workloads on identical hardware, and we are using the same operating system and same Spark versions for our experiments.

Characterizing and subsetting big data workloads. A more recent study [32] on characterizing big data workloads focused on inferring workloads properties from traces collected from BigDataBench benchmark. The purpose of this study was twofold: to come up with a way to do workload subsetting (selecting representative workloads from a set of workloads) as well as to guide the design of future distributed systems architectures. The study consists of collecting traces from different workloads on two big data software stacks (Hadoop and Spark), and then it uses these collected traces in order to analyze the workloads after doing PCA followed by a k-means clustering.

An interesting finding within this study is that PCA was unable to disentangle the algorithm (the workload) from the softare stacks on which it was run. For instance, representations learned from the same algorithm but on two different software stacks are not usually clusterd together, while representations learned from two different algorithms on the same software stack (either Spark or Hadoop) are close to each other. While in our study we focus on modeling workloads on top of a single software stack (Spark), we have run our workloads with different configurations on the same distributed infrastructure in order to study the impact of the configuration on different workloads. We had a similar finding in our work regarding PCA as a representation learning technique: PCA couldn't disentangle the workload characteristics (the algorithm) from the configuration with which the workload was submitted (the configuration is determined by the number of cores and memory allocated for the workload, the degree of parallelism, and other knobs introduced later in Table 2 of Chapter 6)

Hence, our work outlines in Chapter 3 different properties that could be satisfied within workload encodings and that could help in the task of performance regression tuning. We

also introduce in Chapter 5 more advanced representation learning techniques that go beyond a simple dimensionality reduction (as is the case with PCA) and that add a little supervision on encodings while training them from traces.

2.3 AutoML

Promising systems have emerged from the machine learning community to address a similar problem: that of tuning the hyper-parameters of the machine learning algorithms. Among the most successful systems were Hyperband [38], Spearmint [55], and then BOHB [23]. The underlying ideas governing these systems can be directly cast to tune Spark workloads instead of tuning hyper-parameters of machine learning models. These methods are however invasive and require several iterations of tuning before proposing a good configuration. In the following, we discuss these different approaches in more detail.

Spearmint. Spearmint [25, 55, 56, 58] is a tuning package that emerged within the ML community in 2012 as part of an effort to make ML accessible for everyone and is now part of a large-scale project called *AutoML*. It focuses on automating the process of tuning hyperparameters of ML models instead of leaving the tedious tuning task to the user and has outperformed expert level performance while tuning the hyper-parameters of diverse set of ML problems. Thus it became a widespread substitute for the naive bruteforce hyper-parameter search (or so called grid search). Spearmint uses a Bayesian Optimization framework to solve the tuning problem and models the performance of a ML algorithm using Gaussian Processes. Bayesian Optimization is good at finding minima of blackbox functions (which are most probably non-convex) while starting with few available evaluations. The problem we're trying to address in this thesis, automatically tuning Spark knobs, is similar to the problem of tuning hyper-parameters of ML algorithms that Spearmint addresses. Spearmint resorts to a proxy optimization algorithm since it can't directly minimize the unknown blackbox function $f(\mathbf{v})$ (function of a hyper-parameter knobs vector $\mathbf{v}$) that models the predictive performance of ML algorithms. Hence, Spearmint maximizes the expected improvement acquisition function which has a closed form under the GP prior and is given by:

$$a_{EI}(\mathbf{v}) = \tilde{\sigma}(\mathbf{v})(\gamma(\mathbf{v})\ \Phi(\gamma(\mathbf{v})) + \mathcal{N}(\gamma(\mathbf{v}); 0, 1)) \tag{2}$$

where Φ denotes the cumulative distribution function of the standard normal, $\gamma(\mathbf{v})$ is given by: $\gamma(\mathbf{v}) = \dfrac{f(\mathbf{v}_{best}) - \tilde{\mu}(\mathbf{v})}{\tilde{\sigma}(\mathbf{v})}$, and $\mathbf{v}_{best} = \underset{\mathbf{v}_n}{\operatorname{argmin}}\, f(\mathbf{v}_n)$

The expected improvement (EI) acquisition function does not require to tune its own parameters in contrast to the LCB introduced earlier in equation 1 and which Ottertune uses before recommending a configuration. That being said, the expected improvement expression still has two components that implicitly capture the tradeoff between exploitation (evaluating at points with low mean) and exploration (evaluating at points with high uncertainty) under a GP prior. The expected improvement can be seen as a greedy procedure that always tries to minimize further the objective in the next function evaluation.

While Spearmint focuses mainly on optimizing the predictive performance of ML algorithms by minimizing the EI, we focus in this thesis on optimizing the runtime latency of workloads deployed on Spark. Nevertheless, Spearmint has also introduced in the paper [55] another

acquisition function called *expected improvement per second* which accounts for the wallclock time while optimizing the predictive performance since different hyper-parameters within different tuning sessions of the same ML algorithm yield different runtime latencies.

On the other side, the package code suffers from the cold start problem and doesn't support without any modification an assisted warmup from some initial traces. We have nevertheless forked[2] its implementation and modified it so that we can use it for sampling offline traces with a focus on a single target objective.

Finally, Spearmint has already gained the attention of systems community, and Cherrypick [11], a whole tuning system for cloud workloads, was built on top of it.

Hyperband. Hyperband [38] appeared in 2018 as a bandit based approach to hyper-parameter optimization. This algorithm was intended to solve the same problem as Spearmint: tuning the hyper-parameters of ML models with the aim of achieving the best predictive performance. In contrast to Spearmint which uses a Bayesian Optimization framework for addressing the problem, Hyperband uses a bandit based framework and focuses mainly on speeding up random hyper-parameter search by exponentially allocating more resources to more promising configurations. Thus it can be seen as a racing approach between different candidate hyper-parameters with an early stopping mechanism that tracks a predefined resource. It proceeds with sampling candidate hyper-parameters at random, and then runs them with a small budget of resources (with a small number of iterations or a small number of data samples or a small number of features, etc...) Then, the most promising candidate hyper-parameters are rerun with a bigger budget. This repeats over many *successive halving* [31] iterations that prune the less promising configurations and finally increase the fidelity of the most promising configurations by allocating more resources for running them.

If we cast this approach to our use case of tuning Spark knobs, we have to keep in mind that Hyperband does not make use of a pre-trained model that can quickly recommend a better configurations for a submitted workload, and that such an approach will rely heavily on exploring many configurations in order to yield better results. This approach will have to use the computation time of the cloud user for running his workload with multiple configurations before being able to recommend a new configuration that has the potential of achieving a latency improvement with respect to the initial configuration with which the workload was submitted. The tuning approach that we later discuss in this thesis does not make use of the cloud computation time of the user before recommending a better configuration.

However, if we suppose that the cloud user can afford to have a fixed time budget for tuning, then we can use Hyperband to tune Spark workloads, but we should choose first the resource over which we fix a computation budget in the beginning. The duration of the streaming workload seems a good choice. This is not to be confused with the runtime latency (the time it takes to run analytics computation on a batch of streaming) which is the objective that we'd like to optimize. Hence, we can use Hyperband to launch a few configurations on a particular Spark Streaming workload for a short streaming period, and then exponentially increase this streaming period for more promising configurations. So we need to come up with a geometric distribution with an increasing average budget per configuration. However, the maximum budget is 180 seconds (3 mins) because the trace data we sampled on our workloads and which we later introduce in Section 6 have a streaming

[2]Our fork is available publicly at this URL: `https://github.com/zaouk/Spearmint`

duration of 3 mins. Using this custom geometric distribution: $\{30, 60, 120, 180\}$ (with $\eta = 2$ except for the last bracket) reveals a poor design choice for two reasons. First, according to Hyperband's algorithm this would allow us to start with only 6 configurations to run with the shortest streaming duration (30seconds). It's not very interesting to let Hyperband explore only 6 configurations while it's not a model based approach that can make use of previous traces from other submitted workloads. Second, having a minimum budget equals to 30 seconds is problematic since it does not yield predictable behavior under the same configuration. In other words, if we run the streaming workload with the same configuration but we limit the duration of streaming to 30 seconds then we don't get consistent runtime latency across the different runs. This observation is mainly due to the randomness in the distributed workload behavior during the warmup phase. Hence, for these two reasons, we don't provide any end-to-end comparative results with Hyperband.

BOHB. BOHB [23] is a hybrid approach between Bayesian Optimization (BO) and bandit based methods that aims to address the limitations of both worlds. Hyperband [38], the bandit based method introduced earlier suffers from the lack of guidance problem because it samples random configurations without learning from previously sampled configurations. On the other hand, GPs dominate the BO literature (in particular Spearmint [55] which we introduced earlier) and lead to computational infeasibility as soon as the number of data points increase because of its cubic-time complexity. BOHB uses a model-based search at the beginning of each iteration of Hyperband [38] instead of starting with random configurations. The Bayesian Optimization part of BOHB also addresses the problem of computational infeasibility by using a variant of a Tree Parzen Estimator (TPE) instead of using a GP. Although this approach is interesting, we don't consider it in our comparison since it requires a budget for running configurations on the cloud infrastructure reserved by the cloud user before recommending a better configuration for his/her Spark workloads.

2.4 Existing cloud offerings

Automatic tuning of resource allocation knobs has been part of what cloud service providers started to offer. For instance, as part of their Elastic MapReduce (EMR) offerings, AWS has always supported automatic scaling of the resources for jobs which fail for example when Spark is launched with insufficient resources. AWS also supported [3] two profiling tools (Dr. Elephant [39] and Sparklens [2]) for tuning Spark workloads, both of which are not learning based tools but rather rule-based heuristics for tuning. They didn't however address the tuning of other important Spark knobs until recently as part of EMR 5.30 whose release date coincides with the starting time of writing this thesis manuscript. The new version addresses mainly Spark SQL related knobs while our contribution in this thesis covers mainly Streaming workloads. That being said, whether we're speaking of the new optimizations or of the previous ones, the cloud service providers do not support a sort of repository of jobs profiling data that can leverage information collected from past jobs runtime to recommend better configurations for similar jobs, while this thesis does.

3 Problem Settings, Environment and System Design

In this chapter, we first motivate the challenges for building tuning systems by looking at real world requirements in §3.1, then in §3.2 we present the design of our system and describe the different components that fulfill these requirements. Then, in §3.3 we provide the formal description of our problem, and in §3.4 we introduce properties that deem to be essential for learnining numerical vectors describing our workloads.

3.1 Real world requirements

Support for mixed workloads Analytics workloads running on production systems cover today a broad range of tasks such as SQL queries, graph analytics, ML analytics, etc. Cloud practitioners (developers and engineers) rarely write workloads using different distributed systems when they want to analyze data at their disposal because it is both time consuming and technically challenging. One has to be proficient in the programming framework for each of these systems, understand the best practices for tuning its important knobs and then spend a technical budget implementing workloads using each of these systems. The technical debt grows even more if one has to implement his own tuner for the different systems. On the other side, few are the distributed systems that natively support a mixed type of workloads within the same framework. In particular, Spark [66] which emerged from AmpLab at Berkley as a university project and is now maintained and supported by DataBricks, has the benefits of supporting mixed workloads at these 2 levels: (1) *streaming* vs *batch*: Spark has a *lambda* architecture, which means that the same instructions and operations can be applied to both batch and streaming data ingested into its unified engine. (2) *SQL-like* vs *non SQL* queries: Spark can process the database like type of analytics written in SQL as well as more recent types of analytics that rely on data mining and machine learning algorithms. Hence, it is natural to choose Spark as a target distributed system when building a cloud optimizer with support for mixed workloads.

Non-invasive modeling Cloud users at startups and tech companies are more concerned about privacy regarding the data and the workloads they run on a foreign cloud provider's infrastructure. Recent regulations such as the GDPR have been issued to protect the data and the privacy of people in Europe in the age of the Big Data. Technology providers have since started adapting their technologies to more privacy preserving paradigms. Hence if a cloud provider is willing to develop a technology for tuning the distributed workloads of its clients, privacy has to be a first class citizen within this technology. With this in mind, our approach to modeling and tuning workloads should not be invasive with respect to the cloud user's data. So it should not have any access to the workload code written in plain text nor to the datasets (or data streams) on top of which the workload is running. This implies that the tuning and the optimization should be done in a blackbox manner. For instance, Spark workloads written in *Scala* are usually submitted via the *spark-submit* command which takes as arguments different configuration knobs as well as a jar file compiled by the cloud user. Our tuning tool should not expect access to the code that was used for building the jar file, but rather the only thing that it should be able to access is the meta-data used when submitting the workload to Spark (containing the name of the jar file, the initial arguments

chosen by the user, the class name of the workload that is submitted and that can be a unique identifier for the workload). Nevertheless, with the consent of the cloud user telemetry data can be collected from the infrastructure on top of which the workloads are run. Also, key metrics from the distributed system (Spark in our case) are also collected and aggregated during the runtime of the workload.

Online and offline runtime sessions We consider two types of runtime sessions for the workloads: *(1) online runtime sessions* that start upon the submission of the workload to our tuning system and which usually consists of a runtime with either a default initial configuration or an intial arbitrary configuration selected by the user. We make the assumption that the user can only afford to tune few traces (not more than 5) within the online runtime session. This assumption is realistic because the cloud users usually pay for the uptime of the cloud instances they're booking and because the manual tuning process consumes machine time and cpu cycles which are considered an overhead to pay on top of the actual cost of running the workloads. *(2) offline runtime sessions:* These are sessions that should be launched by the tuning system overnight in order to enrich the training with more data points. We can expect for past training workloads to have them sampled for more than 5 different configurations. The tuning system can sample these configurations independently using either a random search assisted with best practices and heuristics from Spark, or it can use a task oriented sampling using Spearmint [55], Hyperband [38] or BOHB [23] which we previously introduced in §2.3. The former method allows the collection of general-purpose traces while the latter can enrich the trace dataset for only a specific downstream task (such as minimizing the runtime latency or maximizing the throughput.) Sampling more configurations overnight for each training workload can help improve the performances of the models used for online tuning up to a certain point as we later explain in our ablation study in §7.5.

Repeated workloads. Most workloads that are submitted to big data analytics platforms today are workloads that exhibit similarities. For example, in some cases the workload is repeatedly run with different ingested data points. Or in other cases, workloads are run by modifying the selectivity of the filter at the end of the pipeline. We leverage this fact to design a system that models the performances of Spark by solely taking into consideration their runtime traces. This important property of workloads has also influenced the way we have designed and extended benchmarks for evaluating our tuning system and which we detail in Chapter 6.

3.2 System Design

The presented real-world requirements led to the design of our unified data analytics optimizer UDAO shown within Figure 3. The left part of the diagram shows the online flow upon the submission of the workload to the tuning system. The right part shows modules that are active during the offline phase. These modules sample additional traces for past workloads and use them to retrain the predictive ML models. We describe below the different components of our system.

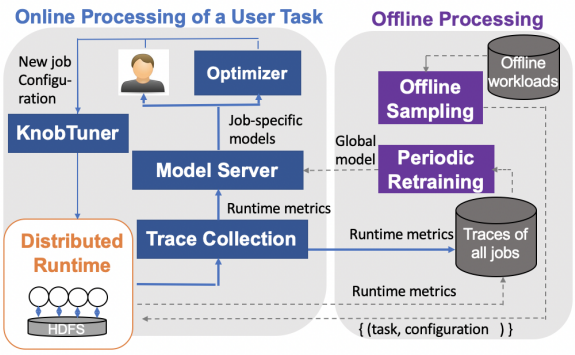


Figure 3: UDAO's System Design.

The controller (KnobTuner) We assume that the workload can be submitted to our tuning system via an alternative command rather than *spark-submit* command. The controller will receive the arguments of this command, and can in turn submit the workload to spark distributed runtime via the *spark-submit* command. If the user has provided an initial configuration when submitting his workload to the optimizer, then this configuration is mirrored within the *spark-submit* command, otherwise the controller chooses an arbitrary configuration for running the workload.

Trace collection module While the job is running, the trace collection module monitors real-time metrics from both spark and the operating system. Once the workload execution is done, this module first aggregates the trace over the runtime period and computes the average metrics. It concatenates both types of metrics (os and distributed engine metrics) into a single trace. It then feeds the trace to a model server and saves a copy within a database of traces. Chapter 6 is dedicated for detailing the workloads used for benchmarking and traces collected from these workloads.

Model Server The model server maintains global regression models trained using all workloads. It uses these regression models at inference time upon the admission of a new workload for which a prediction is needed quickly in order to recommend a better configuration. We distinguish between two types of models: (1) *implicit workload representation models*: these are regression models in which each workload is implicitly represented by an embedding vector $\mathbf{z}$ that is learned while the regression model is being trained on predicting the target objective using a knob configuration $\mathbf{v}$ (2) *explicit workload representation models*: these are models with two main steps: (a) a representation learning step that extracts workload encodings $\mathbf{z}$ out of runtime metrics $\mathbf{x}$ (b) a regression step that uses the extracted $\mathbf{z}$ alongside knob configurations $\mathbf{v}$ to predict the target objective. This thesis focuses on studying effects of different modeling choices on performance prediction from both families. The first family of models is introduced in Chapter 4 while the second family of models is introduced in Chapter 5.

Symbol	Description
$\mathbf{v}_j^i$	ith knob configuration vector (of size s_v) set for job j
$\tilde{\mathbf{v}}_j^i$	approximation of ith knob configuration vector (of size s_v) set for job j using encoder
$\mathbf{x}_\mathbf{j}^\mathbf{i}$	runtime metrics vector observed (of size s_x) for with ith configuration of job j
$\tilde{\mathbf{x}}_j^i$	approximation of runtime metrics x_j^i using auto-encoder
$\mathbf{z}_j^i$	partial latent encoding vector obtained (of size s_z) for job j using configuration i
$\mathbf{z}_j$	latent encoding vector (of size s_z) for job j
y_j^i	runtime latency (scalar) observed with ith configuration of job j (regression target)
f	regression function that maps knob configuration v_j^i to an approximation of latency y_j^i
F	Multi-output regression function that maps knob configuration v_j^i to an approximation of runtime metrics x_j^i
e	encoder function used within encoding part of auto-encoder
d	decoder function used within decoding part of auto-encoder
N	Total number of training points for all workloads altogether.
n	Total number of training workloads
m	Total number of training configurations

Table 1: Notation.

Optimizer The optimizer is the driving component of the tuning system. It is responsible for admitting the workload and then invoking the model server in order to probe it at different points before recommending a configuration for the submitted workload. The optimizer can be a single objective optimizer that focuses on a single performance objective or a multi-objective optimizer that recommends configurations which achieve a certain tradeoff between multiple service level objectives. Nevertheless, in the scope of this thesis, we focus on optimizing a single objective (the runtime latency of distributed jobs).

Offline sampler Today's cloud service providers put unused resources at very affordable prices compared to traditional on-demand instances. These resources are provided as part of some offers under the name of *spot* instances. The offline sampler can benefit from these cheap resources in order to launch offline tuning sessions and boost the predictive performance of the global models. It prioritizes workloads for which only few traces are available in the training set. It then enriches the training data by sampling either general purpose traces or traces dedicated for a particular downstream task (minimizing the latency, maximizing throughput, etc...). In Chapter 6, we discuss in details the heuristics and best practices from Spark that we used for sampling our general purpose traces.

Periodic retraining module After each session of offline sampling, the system periodically retrains the global model maintained within the model server after incorporating the new traces into training data. It then makes new models directly available for inference and probing by the optimizer.

3.3 Problem Statement

We focus in this thesis on modeling the runtime latency of the jobs as a single service level objective to tune. In other words, we are interested in predicting and then tuning how

much time it takes for a workload to execute on Spark if it's a batch workload, or how much time it takes to execute a window of computation if the workload at hand is a streaming workload. We model the runtime latency objective y_j^i as a function of some latent workload characteristics $\mathbf{z}_j$ (to be learned) and a given configuration $\mathbf{v}^i$ of the workload submitted to cloud. Thus, our tuning problem boils down to finding both f^* and $\{\mathbf{z}_j\}$ such that:

$$f^* = \underset{f}{\operatorname{argmin}} \frac{1}{N} \sum_{i,j} \left(f(\mathbf{z}_j, \mathbf{v}^i) - y_j^i \right)^2 \tag{3}$$

where N is the number of training points, y_j^i is the observed latency under configuration vector $\mathbf{v}^i$, and $\mathbf{z}_j$ is a latent embedding vector that needs to be learned from the observed runtime metrics vector $\mathbf{x}_j^i$. We provide in Table 1 the notation adopted throughout this manuscript.

The same logic applies if we replace the runtime latency with another service level objective. For example, if we are interested in modeling the throughput of workloads (in terms of records processed per second) instead of the runtime latency, we would opt for a similar optimization problem where the target y_j^i represents the throughput instead of the runtime latency. If we are less interested in the exact value of the throughput, but rather interested in predicting whether the configuration i alllows a low, medium or high throughput for the particular job j then the mean squared error loss should be simply replaced by a cross-entropy loss and the problem becomes a classification problem instead of a regression problem.

The main challenge that arises in our problem, whether being a regression on the runtime latency or throughput, or a simple classification on throughput, is that workload characteristics $\mathbf{z}_j$ are unknown and we have to extract them from the runtime traces observed for the workloads under past configurations. We explain in the next section some properties that are inspired from the domain knowledge about the modeling problem and that help us extract meaningful workload characteristics.

We decide to run our workloads on identical hardware types (see §6.2 within Chapter 6 for more details). Thus, we ignore the hardware characteristics when we introduce our modeling function f. Nevertheless, this formalism allows an extension to include an additional representation of the hardware parameters but we leave the modeling of the hardware characteristics beyond the scope of this thesis. In the more general case, the modeling function f would be a function of: (1) workload characteristics (2) resource allocation related parameters (number of cores, memory size, etc..) (3) distributed engine (Spark in our case) related parameters) (4) hardware specific parameters (number of cpus, cpu brand, number of gpus, gpu brand, etc...). Our main contribution considers (1), (2) and (3), while (4) is left out of the scope of this thesis because we choose identical hardware.

3.4 Workload embeddings properties

In this section, we introduce different properties that we would like to use in order to drive the design of different modeling architectures and then assess the quality of embeddings vector $\mathbf{z}_j$ we would like to learn for each workload.

- **Independence**: This property describes the independence between the different factors that contribute to the generative process of the observed runtime traces $\mathbf{x}_j^i$. We distinguish

between two types of independence: (1) Independence between the workload descriptors to be extracted (so called encodings or embeddings) and the knob configuration: Since the runtime latency is affected by both the configuration of the distributed system $\mathbf{v}^i$ and the workload submitted to the cloud, then it is reasonable to assume that the workload encodings $\mathbf{z}_j$ we are trying to learn and the system knobs $\mathbf{v}^i$ should be orthogonal and independent of each other. (2) Independence between different workload descriptors themselves: This type of independence is optional but can be seen as a way to regularize encodings by ensuring that we don't have a dependency between different learned components for the workload encoding vector $\mathbf{z}_j$. It can also help us simplify complex mathematical modeling assumptions (for example within variational auto-encoders later introduced in §5.3).
The independence property is highly dependent on the ability of the representation learning technique to disentangle the factors that are responsible for the generation of the observed runtime metrics (such as parametrization of the workload, knob configuration) or the design of the architecture that can force such a separation.

- **Invariance**: This property implies that the modeling tool should provide one encoding vector $\mathbf{z}_j$ that is invariant with respect to how many and which are the configurations whose traces were used for its generation. In other words, no matter which configuration(s) generated some given trace(s), the extracted encoding $\mathbf{z}_j$ vector for a particular workload j should be unique. On the other hand, usually running a workload on a distributed system does not yield deterministic behavior even if the same configuration is run twice unless we assume fixed data characteristics. Hence, it becomes more reasonable to speak of invariance assuming that the data characteristics of a particular workload are fixed across different runs and are inherent to the workload.
- **Similarity**: This property is a relaxation of the invariance property and is meant to tighten together encodings extracted from traces originating from different configurations run over the same workload. It implies that the encoding function should map semantically similar runtime metrics vectors $\mathbf{x}_j^i$ from the runtime trace data manifold onto metrically close embedding points $\mathbf{z}_j^i$. We consider two points to be semantically similar if they come from the same job.
- **Reconstruction**: The reconstruction property reflects the ability of the workload encoding $\mathbf{z}_j$ of reconstructing runtime metrics $\mathbf{x}_j^i$ given a particular configuration $\mathbf{v}^i$. This is a defacto property that is automatically ensured within auto-encoder architectures we later introduce in Chapter 5. It implies that the bottleneck layer from which the workload encoding should be extracted has this reconstruction ability.
  The reconstruction property is especially helpful when it comes to learning a generic encoding that can be used for modeling multiple performance objectives provided as separate components of runtime vectors $\mathbf{x}$. In other words, this property is useful as soon as the encoding needs to be used with multiple downstream tasks.

Throughout Chapters 4 and 5, we will introduce different modeling architectures and we will explain how difficult it is to satisfy simultaneously all of these properties due to either (1) architectural limitations or (2) tradeoffs between the different properties when they are reflected within separate terms in a particular loss function.

3.5 Summary

In this chapter, we have covered real world requirements from cloud services that led to the design of our tuning system and then described the different components of our system. We gave a particular importance to the model server component which we study in detail throughout this thesis. We have then formalized the problem of modeling runtime latency of distributed workloads and explained that this problem is a non trivial one because of the lack of numerical descriptors for workload embeddings. Finally, we have listed a set of properties that we would like to use in order to compare different architectural choices for learning meaningful workload embeddings and see their impact on modeling accurately the runtime latency.

4 Recommender based architectures

In this chapter, we show how it's possible to cast the workload tuning problem into a recommender systems framework after we notice that our tuning problem exhibits similarity with the recommendation problem. Indeed, recommender systems have gained widespread adoption within e-commerce portals and online streaming services in order to increase sales by recommending new items for a user based on a past purchase/watching list. In the context of tuning workloads, a configuration recommender system is a system that recommends better configurations for a particular workload based on performance observed for this workload under some intial configuration(s). Thus, if we make the analogy to existing recommender systems used in e-portals, the workload here takes the role of the *user* while the configuration takes the role of the *item* that needs to be recommended. However, the overall objective of the recommendation in our use case is to reduce the runtime latency of a given job.

There exists three main families of recommender systems within the literature: (1) *collaborative filtering methods*, (2) *content based methods* and more recently (3) *hybrid methods* that mix the first two. While the collaborative filtering methods rely heavily on the interactions between users and items, content based methods focus on recorded features for both the users and the items. Throughout this chapter, we survey state-of-the-art techniques from these families while listing existing work in the database and systems literature that make use of these techniques in the context of tuning workload configurations. We start with collaborative filtering methods in §4.1 and go over both memory and model based methods. We explain in §4.1.2 how matrix factorization techniques are used as tuning tools within Paragon [20] and Quasar [21] which we briefly introduced earlier in the related work section in §2.1. Our contribution consists of providing a formulation of the tuning problem that takes into account both workload and configuration biases in §4.1.2 as well as another trick that allows workload encodings and configuration encodings to have different latent dimensions. We also propose in the same paragraph a neural network based architecture which we call *Dual Embeddings* and explain why it is interesting to explore it in our problem settings. We next briefly introduce in §4.2 the family of content based methods and explain how Ottertune [61] can be seen as such a technique. Finally, we give examples of hybrid methods in §4.3 and propose a *Hybrid Embeddings* architecture which is obtained from a slight modification to the *Dual Embeddings* approach in such a way that we provide the configuration vector as input instead of learning latent embedding for it. This latter architecture enables making predictions for arbitrary configuration (not necessarily seen previously for some of the existing workloads) and hence allows the recommendation to go beyond existing configurations within training data.

4.1 Collaborative filtering methods

The basic building block of collaborative based methods, widely used in the context of item recommendation for users, is having an *interaction* matrix that contains *ratings* users gave to previously watched movies. This matrix is usually sparse, as users can't watch all movies. Similarly, in the context of tuning workloads, the *interaction matrix* should contain workload runtime latencies under different knob configurations. This matrix is also sparse because it's impossible to evaluate each workload on every single knob configuration. Solving the recommendation problem boils down to first providing estimates for the runtime latency

across all missing fields within the interaction matrix, then recommending configurations that yield the minimum estimated runtime latency for the current workload.

Within the collaborative filtering family, there exists two types of methods for recommendation: (1) *memory* based methods and (2) *model* based methods.

4.1.1 Memory based methods

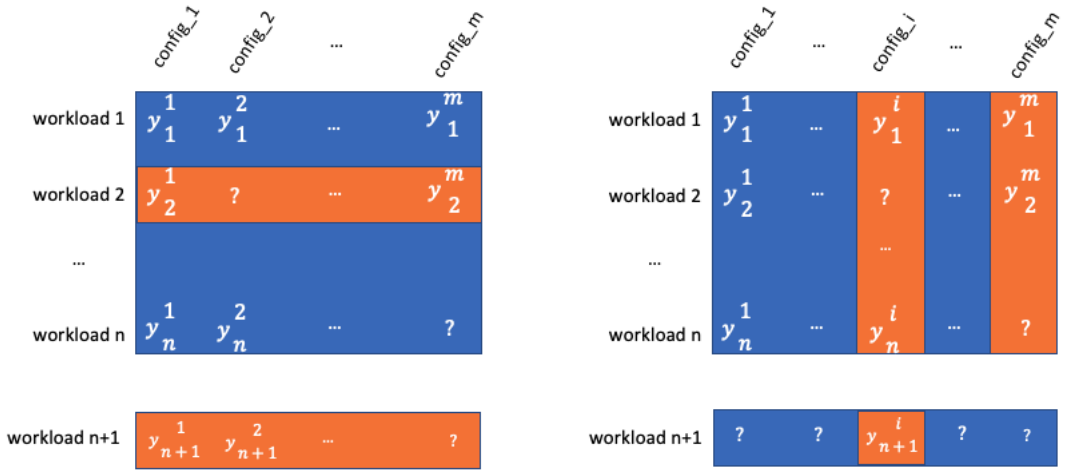


Figure 4: Memory based nearest neighbor mapping.

Memory based methods apply a nearest neighbor strategy before recommending a new configuration to a workload. There are two ways for doing so: an example of *workload-workload* mapping is provided in the left part of the diagram in Figure 4 and a *configuration-configuration* mapping is provided to the right. We further explain these two ways of mapping below:

- 1) *workload-workload* A newly admitted workload $n+1$, described by a sparse vector of runtime latency values over m configurations $\mathbf{y}_{n+1} = (y^1_{n+1}, y^2_{n+1}, y^3_{n+1}, ..., ?, y^i_{n+1}, y^{i+1}_{n+1}, ?, ...y^m_{n+1})$ is first mapped to the most similar workload represented by a row within the interaction matrix (after imputing the missing values within both the matrix and the new vector). Then, the most performing configuration of the mapped workload (the configuration that yields the lowest runtime latency on the mapped workload) is generally recommended to the original workload.
- 2) *configuration-configuration* When a new job is admitted with some previously seen configuration(s) within training data, then the most performing configuration i among those observed, and which can be described by a column within the interaction matrix (as a vector of runtime objectives across the different jobs $\mathbf{y}^i = (y^i_1, y^i_2, ?, ...y^i_j, ...y^i_n)^T$ is first mapped to its nearest neighbor configuration within the interaction matrix. This nearest neighbor configuration is then recommended to the new workload. The configuration-configuration memory based mapping does not take into consideration an order relationship and may hence return a configuration that is worse in terms of runtime latency.

There are mainly three issues with memory-based methods: (1) They require the recommender system to sample a lot of shared configurations and run them across all workloads so that

nearest neighbor mappings become meaningful. (2) They require mapping a vector to its nearest neighbor in a high dimensional space because both n and m are on the order of thousands in real-world datasets. The mapping step hence unfortunately suffers from the curse of dimensionality and won't work properly as explained in [28]. (3) They have a scalability issue: the nearest neighbor search (whose complexity is $O(nmk)$ with n:#of workloads m:# of configurations and k:#neighbors) becomes intractable as soon as the number of workloads or configurations increase.

4.1.2 Model based methods

The model based methods rely on the assumption that a latent model for workloads and configurations exist and such a model can explain the interactions between workloads and configurations. There are several choices for modeling the interaction between the workload and the configuration. The dot product between two vectors is a common way for representing this interaction, and we explain it under the **Matrix factorization** paragraph. Then we explain under **Advanced Matrix Factorization** paragraph how we introduce another matrix variable in order to allow more degrees of freedom while training the matrix factorization loss function. Finally, within the last paragraph of this section (§ on **Dual Embeddings**) we propose yet another alternative. Instead of representing the interaction between a workload and a particular configuration by a dot product, the two learned representations are concatenated and then fed to several fully connected layers that estimate the runtime latency.

Matrix factorization The matrix factorization technique, also known as QP factorization, became popular after it won the Netflix Prize competition [35], demonstrating the superiority of model based methods on memory based methods for building recommender systems. Under this framework, we assume that the sparse workload-configuration interaction matrix Y can be approximated by a product of 2 dense matrices[3] whose representations are in a D-dimensional latent space: (1) a *workload-factor* matrix Z and (2) a *factor-configuration* matrix V.

Consider J the set of the jobs that we have in our training set, I the set of configurations sampled across the different workloads. For each couple (j, i), either the job j hasn't been run on configuration i, so we don't have a data point for it, or, we have recorded the average runtime latency y_j^i for running this job under the configuration i. Y denotes the interaction matrix where an element of this matrix Y_{ji} at the j-th row and i-th column (thus corresponding to the j-th workload and i-th configuration) corresponds to the recorded runtime latency y_j^i. In other words, $Y_{ji} \equiv y_j^i$

We hypothesize that there exists a $D-$dimensional latent space of joint characteristics of workloads and configurations such that the *interactions* between the workloads and the configurations are dot-products within this space. According to this model, we should have:

$$Y_{ji} \approx \sum_{d \in D} Z_{jd} V_{di} = \mathbf{z}_j^T \mathbf{v}^i \tag{4}$$

[3]We use Z and V instead of Q and P in order to avoid confusing with earlier introduced notation.

where $\mathbf{z}_j^T = Z_{j,:}$ (j-th row of the Z matrix) is a representation of job j in the $D-$dimensional latent space and $\mathbf{v}^i = V_{:,i}$ (i-th column of the V matrix) is a representation of the configuration i in this same dimensional space.

The factor matrices Z and V are first initialized by running an SVD decomposition of the initial interaction matrix Y. Then, Z and V are updated by minimizing a regularized least squares loss function:

$$(\hat{Z}, \hat{V}) = \underset{Z,V}{\operatorname{argmin}} \frac{1}{2} \sum_{(j,i)\in K} (Y_{ji} - \sum_{d\in D} Z_{jd} V_{di})^2 + \frac{\lambda}{2} (\sum_{j\in J, d\in D} Z_{jd}^2 + \sum_{i\in I, d\in D} V_{di}^2)$$

$$(\hat{Z}, \hat{V}) = \underset{Z,V}{\operatorname{argmin}} \frac{1}{2} ||1_K \circ (Y - ZV)||_F^2 + \frac{\lambda}{2} ||Z||_F^2 + \frac{\lambda}{2} ||V||_F^2 \tag{5}$$

where K is the set of couples for which $L_{j,i}$ is known. $||.||_F$ is the Frobenius norm, $(1_K)_{j,i} = 0$ if $(j,i) \notin K$ and $(1_K)_{j,i} = 1$ if $(j,i) \in K$. $(X \circ Y)_{j,i} = X_{ji} Y_{ji}$ and $\lambda > 0$ is a regularization parameter.

Paragon [20] and Quasar [21] from the system literature use the same matrix factorization techniques in order to recommend better configurations for cloud workloads. Paragon [20], in particular, minimizes the same loss function that we provided in Eq. 5. The authors provide the iterative formulas for learning and updating the encodings for both the workloads and configurations. They rely mainly on optimizing the loss function using alternating least squares which fixes one variable while minimizing the other. This allows having a convex function and thus guarantees convergence towards a minimum. The derivations from the loss function we provided in Eq. 5 and that lead to the iterative updates mentioned within Paragon paper are straightforward. We defer them to §A within the appendix for the interested reader.

Instead of manually implementing the gradient descent updates on vectors $\mathbf{z}_j$ and $\mathbf{v}^i$, we have resorted to automatic differentiation tools and thus implemented a solver for the loss function in Eq 5 in Tensorflow 2.0 [7, 8].

Quasar [21], on the other side, extends this loss function to account for the *user bias - workload bias* in our context. The matrix factorization original paper [35] goes beyond this extension and also accounts for the *item* bias - bias that is due to *configurations* in our context. We therefore implement a more general version that covers both biases. The approximation of runtime latency for a job j under configuration i is not anymore given by a simple dot product between $\mathbf{z}_j$ and $\mathbf{v}^i$ as in Eq.4, but is rather given by:

$$Y_{ji} \approx \mathbf{z}_j^T \mathbf{v}^i - \mu - a_j - b_i \tag{6}$$

where μ is the average runtime latency across all collected data, a_j is the intercept (bias) for job j and b_i is the intercept for configuration i.

μ, a_j and b_i are given by:

$$\mu = \frac{1}{|K|} \sum_{(j,i)\in K} (1_K \circ Y)_{ji}$$

$$a_j = \frac{\sum_{i=1}^m (1_K \circ Y)_{ji}}{\sum_{i=1}^m (1_K)_{ji}}$$

$$b_i = \frac{\sum_{j=1}^{n}(1_K \circ Y)_{ji}}{\sum_{j=1}^{n}(1_K)_{ji}}$$

The extension we brought on top of Quasar can be written in its matrix form as:

$$\mathcal{L} = \frac{1}{2}||1_K \circ (Y - ZV + M + A + B)||_F^2 + \frac{\lambda}{2}||Z||_F^2 + \frac{\lambda}{2}||V||_F^2 \tag{7}$$

The matrices M, A and B are given by:

$$M = \mu * \mathbb{1}_{n,m}$$

$$A = \mathbf{a}\mathbb{1}_m^T$$

$$B = \mathbb{1}_{\mathbb{n}}\mathbf{b}^T$$

$\mathbb{1}_{n,m}$ is a matrix of ones with n rows and m columns, $\mathbb{1}_n$ and $\mathbb{1}_m$ are column vectors of ones of size n and m respectively.

$\mathbf{a}$ and $\mathbf{b}$ are vectors representing jobs and configurations biases and are defined by:

$$\mathbf{a} = (a_1, a_2, ...a_j, ..., a_n)^T$$
$$\mathbf{b} = (b_1, b_2, ..., b_i, ..., b_m)^T$$

Advanced Matrix factorization The collaborative filtering approach proposed so far can be modified to go beyond a simple matrix factorization. Indeed, instead of representing the interaction between workload j and configuration i as a simple dot product between $\mathbf{z}_j$ and $\mathbf{v}^i$ $\mathbf{z}_j^T\mathbf{v}^i$, we can think of introducing a matrix S in the middle, so that the interaction is approximated by $\mathbf{z}_j^T S\mathbf{v}^i$. Although the introduced matrix S can be seen as a simple linear transformation of the configurations, its elements remain as degrees of freedoms while learning the factorization. Hence, Y can be approximated by $Y \approx ZSV$. This allows us to have a different latent dimension for Z and V so that the shapes become (n, d_z) for Z, (d_z, d_v) for S and (d_v, m) for V.

$$Y_{ji} \approx \sum_{k=1}^{d_v}\sum_{t=1}^{d_z} Z_{jt}S_{tk}V_{ki}$$

The loss function we minimize in this case is given by:

$$\mathcal{L} = \frac{1}{2}||1_K \circ (Y - ZSV + \gamma(M + A + B))||_F^2 + \frac{\lambda}{2}||Z||_F^2 + \frac{\lambda}{2}||V||_F^2 \tag{8}$$

where γ is a hyper-parameter that takes values in $\{0, 1\}$ and is used in order to take into consideration or not the terms that account for bias.

In contrast to the normal matrix factorization where Z and V are initialized using the results of the SVD decomposition, we randomly initialize all three matrices Z, S and V before solving the factorization problem.

Upon the admission of a new workload profiled with few previously seen configurations, a new randomly initialized row $\mathbf{z}_{n+1}$ is inserted within the matrix Z and incremental training is trigerred to update this row. Both matrices S and V are frozen during the incremental training step.

We leave as part of our future work exploring another generalization of the matrix factorization approach that leverages the kernel trick instead of the dot product and we briefly explain it later in section 9.2. The next paragraph introduces an alternative to the dot product that makes use of neural networks to represent the interaction between $\mathbf{z}_j$ and $\mathbf{v}^i$.

Dual-Embeddings Deep learning Architecture We propose a neural network architecture that is more general than matrix factorization approaches since it learns a non-linear model on top of the representation vectors for both workloads and configurations. Since we don't represent the interaction here by a simple dot product, then there is no need that the configuration and workload embedding vectors share the same number of dimensions. The interaction can thus be represented by the output of several fully connected layers with potentially *non-linear* activations which take as input a concatenation of both vectors ($\mathbf{z}_j$ and $\mathbf{v}^i$). This architecture is shown in Figure 5.

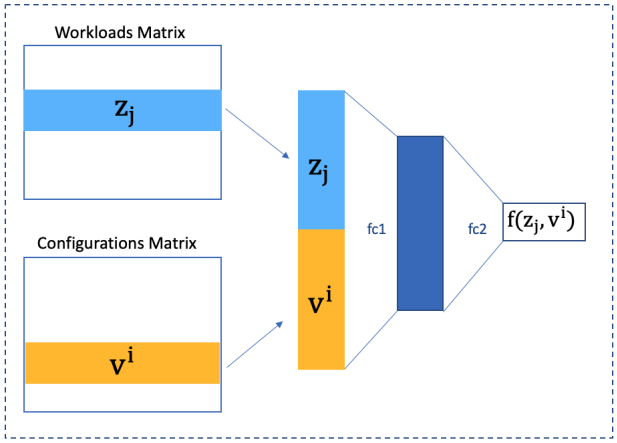


Figure 5: Dual Embeddings Deep Learning Architecture. Both $\mathbf{z}_j$ and $\mathbf{v}^i$ are updated by backpropagation.

Within this architecture, both the workload encoding vector $\mathbf{z}_j$ and the configuration encoding vector $\mathbf{v}^i$ are learned while training the architectures. These vectors are randomly initialized at first, and then their values are updated through backpropagation. The architecture is trained on minimizing the mean squared error between the true value of the runtime latency y^i_j, and its neural based approximation $f(\mathbf{z}_j, \mathbf{v}^i_j) \equiv f(j, i)$

$$\mathcal{L} = \frac{1}{N} \sum_{i,j} (f(j,i) - y^i_j)^2$$

Before we train this architecture we need to organize our training data into tuples of (j, i, y_j^i) where j denotes the job id, i denotes the configuration id and y_j^i is the true value of the runtime latency obtained with job j under configuration i. The architecture requires the ids at the input so that the corresponding vectors can be fed to the fully connected layers.

Similarly to matrix factorization techniques, this architecture requires additional incremental training upon the admission of a new workload $n+1$ with few profiled configurations among existing configurations profiled for training workloads. A new randomly initialized row vector $\mathbf{z}_{n+1}$ is appended to the embeddings matrix, and then this row, which represents the embeddings vector of the new workload, is updated by backpropagation while doing the incremental training. The other fully connected layers of the embedding architecture and the embedding layer of the configurations are frozen during the incremental training step. In the next section we'll introduce methods that make use of additional information while learning the interaction model between workloads and configurations.

4.2 Content based methods

The content based methods enhance models by making use of additional information collected from the interactions between configurations and workloads (other than the objective y_j^i that we are trying to approximate). They suffer far less from the cold start problem, compared to the collaborative filtering method, since new workloads and configurations can be described by their characteristics.

A content based method can be either (1) *workload centered* (2) *configuration centered* or (3) *neither configuration nor workload centered*. We provide a brief overview of these different content based methods.

1. Workload centered approaches. A workload centered approach builds for each workload a model that takes as input configuration knob features outputs an approximation of the runtime latency. Thus, with n workloads in our training data, this approach builds n separate models $f_1, f_2, ..., f_n$. The underlying model can be any type of regressor (a neural network, a random forest regressor, a GP regressor, etc..).

$f_j(\mathbf{v}^i)$ has to approximate y_j^i, and thus $f_j(\mathbf{v}^i) \approx y_j^i$. In contrast to previously introduced methods within the collaborative filtering family, the configuration vector $\mathbf{v}^i$ components consist here of the values of different knobs (as specified earlier in Table 2 of §6.3.1) for a particular configuration i. In collaborative filtering, a vector $\mathbf{v}^i$ was a latent representation of the configuration that doesn't hold direct semantics about the configuration knobs. To give an example, in content based methods, a vector $\mathbf{v}^i = (1, 50, 36, 48, 43, 1, 3, 4, 6553, 660000)$ denotes a particular knob configuration i, where the different knobs take these values: $v_1 = 1$s; $v_2 = 50$ms; $v_3 = 36$; $v_4 = 48$MB; $v_5 = 43$; $v_6 =$True; $v_7 = 3$; $v_8 = 4$; $v_9 = 6553$MB; $v_{10} = 660000$. In other words, here $\mathbf{v}^i = (v_1, v_2, v_3, ...v_{10})$ where each v_k represents a value for a knob introduced in Table 2.

This method does not usually work well if the admitted workload is new and is profiled with only few configurations. Ottertune [61, 68] on the other hand, which we previously introduced in the related work under §2 can be seen as a recommender system that falls in the category of workload centered approaches since it builds one separate model for each single job. However, it works better than a simple workload centered approach when the new admitted workload has few configurations. This is because Ottertune's inference pipeline

consists of mapping the new admitted workload to one of the past workloads and then using a past model for making predictions on top of the new workload. With this mapping extension, Ottertune can be seen as a mix between a *workload-centered* content based method and a *memory-based* collaborative approach. We leave the comparison to Ottertune to Chapter 7.

2. Configuration centered approaches. A configuration centered approach builds for each configuration a model based on workloads features. Thus, with m configurations in our training data, this approach builds m separate models, one for each configuration: f^1, f^2, ... f^m. In this case, $f^i(\mathbf{z}_j)$ has to approximate y_j^i and thus $f^i(\mathbf{z}_j) \approx y_j^i$.

This approach is limited because: *(1)* on one side, most configurations are run over one or very few workloads, and *(2)* on the other side, the approach doesn't allow to make predictions over a new combination of knobs never seen before with any of the training workloads, and thus doesn't allow exploration while recommending new configurations.
The workload descriptors $\mathbf{z}_j$ should be unique for the workload j even if the workload is run with several configurations $\mathbf{v}^i$ so that it complies with the invariance and independence properties outlined in §3.4. A configuration centered approach requires the workload characteristics $\mathbf{z}_j$ to be known in advance before training the models. On one hand, we can't simply use the runtime metrics collected for the workload j under a configuration i over which we want to make predictions because we don't have these metrics before running the workload with this particular configuration. On the other hand, since we opt for a non-invasive modeling, we also can't represent the workload by its graph embedding extracted from the dataflow graph. Therefore we leave the graph embedding representation as a future direction to explore, while we discuss in the next chapter how we learn an invariant embedding $\mathbf{z}_j$ from runtime metrics $\{\mathbf{x}_j^i\}$

3. Neither workload nor configuration centered approaches. Under this category, a model that stacks features from both workloads and configurations is trained, and thus the model is represented by a function f that approximates the runtime latency: $f(\mathbf{z}_j, \mathbf{v}^i) \approx y_j^i$. While this model considers at its input the vector $\mathbf{v}^i$ defined by the different knobs $(v_1, v_2, ...v_{10})$, this model also poses a major challenge regarding what features to use as input for the function f when it comes to representing the workload j. The next chapter is dedicated to further elaborating this direction while exploring different representation learning architectures that extract a meaningful representation $\mathbf{z}_j$ for each workload.

Approaches (1) and (3) have the benefits of allowing the model to make predictions over new configurations that haven't been necessarily seen while training. This allows the optimizer to explore interesting regions that can't be explored by other methods.

4.3 Hybrid methods

Hybrid approaches between collaborative and content based filtering can be obtained by either building separately one collaborative based model and another content based model, and then mixing their outputs, or they can be conceived within the same architecture. In this section, we propose a hybrid embedding model whose architecture is identical to the one we already proposed under Figure 5 with the exception this time that configurations $\mathbf{v}^i$ are given as input to the architecture instead of being learned alongside other weights of the neural network. Hence by providing the configurations vectors $\mathbf{v}^i$ as input, we are enriching the previous collaborative filtering approach with explicit content about the configurations

instead of implicitly learning a representation for the configurations. This also allows making predictions over new knob configurations (combinations of knob values which were not among the training points) possible.

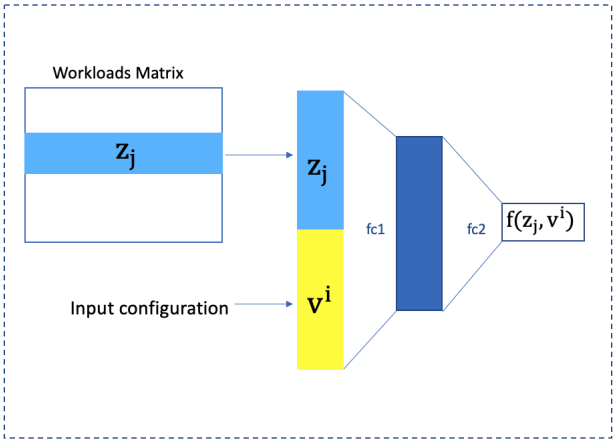


Figure 6: Embeddings Deep Learning Architecture. Only embeddings vectors $\mathbf{z}_j$ are updated by backpropagation, while configuration vectors $\mathbf{v}^i$ are provided as input.

This architecture is depicted in Figure 6 and is an example of a *hybrid* approach: on one side it is *collaborative* because it learns the interactions between configurations and workloads, and learns an inherent latent model for workload embeddings, and on another side it is *content based* because it accepts as input a vector describing the configuration.

This architecture consists of three parts: (1) An embedding layer with a weight matrix Z denoting the latent space. Each row $\mathbf{z}_j$ of this matrix represents a particular workload j as its embedding vector, randomly initialized first. (2) A concatenation layer that for a particular job j, concatenates the embedding vector $\mathbf{z}_j$ with an input (ith) configuration $\mathbf{v}^i$ into, $(\mathbf{z}_j||\mathbf{v}^i)$. (3) Several fully connected (FC) layers that take $(\mathbf{z}_j||\mathbf{v}^i)$ as the input and produce $f(\mathbf{v}^i, j) \equiv f(\mathbf{v}^i, \mathbf{z}_j)$ as the final output. The architecture is trained by minimizing the MSE between the predicted latency $f(\mathbf{v}^i_j)$ and the actual latency y^i_j, that is:

$$\mathcal{L} = \frac{1}{N} \sum_{i,j} (f(\mathbf{v}^i, j) - y^i_j)^2 \tag{9}$$

The embedding approach requires incremental training every time a new job is submitted: when the trace of a new job becomes available, we add a random row to the embedding matrix, freeze the weights of the neural network (except those at the embedding layer) and run incremental training using the trace of the new workload and backpropagate to update the embeddings.

4.4 Summary

In this chapter we have surveyed diverse methods used within recommender systems and explained how matrix factorization approaches in particular have been previously casted

for solving the configuration recommendation problem for cloud workloads. We have then provided an extension of the matrix factorization introduced within the work [20, 21] after adding biases for both configurations and workloads while solving our problem (in eq 7). We have also proposed two neural network based recommender approaches and explained how the *hybrid embedding* approach allows to make recommendations beyond already observed configurations. The recommender architectures introduced within this chapter have coupled both the encoding extraction as well as the regression task. They have ensured both the independence and invariance properties while extracting the encodings $\mathbf{z}$ for the workloads. They have also partially satisfied the reconstruction property since the interaction matrix (used within matrix factorizations) as well as the neural networks introduced in embeddings approaches aim to reconstruct only the runtime latency while minimizing the loss function. In the next chapter, we will focus on incorporating more content by leveraging the full runtime metrics $\mathbf{x}$ while extracting the encodings. We will attempt to separately extract the workload encoding $\mathbf{z}$ using advanced representation learning techniques and then train regression models using these extracted encodings. Finally, in Chapter 7 we will provide comparative results between the different recommender approaches we have surveyed and extended as well as other encoding/decoding architectures which we will introduce in the next chapter.

5 Representation learning based architectures

In this chapter, we introduce different representation learning techniques that serve for the task of modeling the runtime latency of distributed workloads. We show throughout this chapter how each of these techniques satisfy the different desired properties that we have introduced for the workload encoding.

We start this chapter by motivating the need for representation learning in §5.1 and then outline the architectures overview in §5.2. Then, we introduce a basic representation learning architecture consisting of a vanilla auto-encoder in §5.3. Then, we gradually increase the degree of complexity by proposing a domain knowledge driven design for a custom autoencoder, and then we augment it with an additional contractive term. We then introduce variational auto-encoders in §5.3, and explain how such generative auto-encoders compare to the previously introduced deterministic autoencoders in terms of quality of extracted encodings. Finally, we propose techniques from metric learning in §5.4 and then introduce some hybrid architectures combining both auto-encoders and siamese neural networks in §5.5.

We provide a qualitative comparison between the different methods by taking into consideration how well each of the different properties is satisfied.

5.1 The need for representation learning

State of the art tuning tools in the database community (such as Ottertune [61]) do not consider representing different workloads by a numerical vector. Hence such tools are forced to train multiple separate models (one model per workload) and miss the opportunity of training a single global model on the runtime latency.

Maintaining separate models is a burden because as soon as additional traces are periodically collected from multiple workloads, hyper-parameter tuning and model re-training need to be triggerred for corresponding workloads for which additional traces were profiled.

Furthermore, training a separate model for each workload doesn't allow to learn patterns across behaviors of different workloads. For instance, workloads submitted by the same cloud user tend to exhibit similarities when the user is working on the same dataset. The user would often repeat some analytics tasks by changing the selectivity for example. Having to map the new workload to one of the previous workloads (as is done in Ottertune [61]) prevents keeping a *gradient* information about that workload. A typical example would consist of a workload that is half-way similar to two other workloads, but far away from another third workload. Forcing the mapping to one of these two close workloads leads to a waste in the use of the information known about this workload.

In order to train a global model, we need to find a numerical way to describe each workoad. If we consider each workload as a separate category, then we can use a trivial encoding for representing the workloads: the one-hot encoding (or so called dummy variables). Such a representation is not optimal for several reasons. On one hand, one-hot encoding doesn't allow an online prediction scheme and requires retraining the regressor from scratch each time a new workload is observed. This is problematic since retraining models from scratch incurs an important delay overhead, while tuning Spark knobs for an admitted workload should be done quickly on the fly (as in the zero-shot learning scheme).

On the other hand, a onehot encoding scheme considers all workloads as equi-distant (there

are no difference in distance between any two workload vectors). Knowing in advance that workloads we are trying to model are parametrizations from some templates, it is natural for these workloads to have some similarities, and thus we don't expect a one-hot encoding scheme to work properly.

Moreover, we can't consider raw metrics $\mathbf{x}_j^i$ as encodings $\mathbf{z}_j^i$ ($\mathbf{z}_j^i \leftarrow \mathbf{x}_j^i$) assigned for configuration $\mathbf{v}_j^i$, without directly violating both the independence and invariance properties earlier introduced in the problem statement in §3.3. The independence property violation stems from the fact that the different collected metrics are correlated, while the invariance property stems from the fact that raw metrics from the same workload are not the same under different configurations. We later explain in §7.4 why the violation of these properties leads to an inferior performance on the task of modeling the runtime latency. Hence the need for a representation learning step.

5.2 Architectures overview

In this section, we first introduce our modeling pipeline that allows accurate estimation of the runtime latency. It is depicted in Figure 7 and consists of 3 different stages: (1) a *representation learning* module (2) an encodings *aggregator* (3) a *regression* module

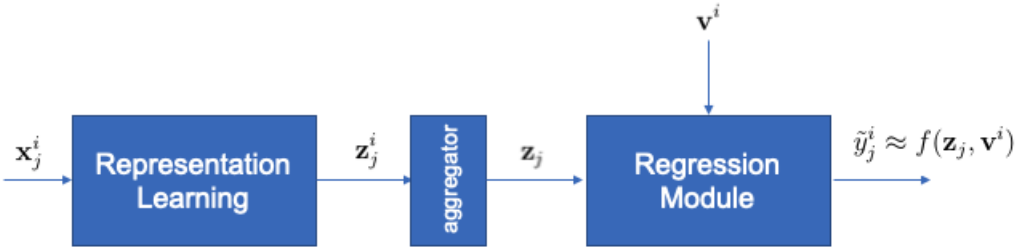


Figure 7: Main components within modeling.

The representation learning component first receives the runtime metrics $\mathbf{x}_j^i$ at its input and then outputs an encoding vector $\mathbf{z}_j^i$. This component is usually an encoder but a decoder can be coupled to the encoder during the training phase (for example if the chosen representation learning module is an auto-encoder). The aggregator extracts a unique encoding vector $\mathbf{z}_j$ for each workload j, from several traces $\{\mathbf{x}_j^i\}$ (yielded by several configurations $\{\mathbf{v}_j^i\}$). It is an attempt to force the invariance property regardless of the representation learning technique that has been deployed.

Thus, from multiple $\{\mathbf{z}_j^i\}$ at the input of the aggregator, it generates a single output $\mathbf{z}_j$.

Finally, a trained regressor can take an encoding vector $\mathbf{z}_j$ and any configuration vector $\mathbf{v}^i$ at its input and predict the runtime latency $\tilde{y}_j^i$ induced by this configuration. The regressor can be of any type: a neural network, a random forest regressor, a gradient boosting regressor, etc. In this work, we choose to use a neural network in order to get a regression function with continuous output so that gradient based optimization can be done using the same predictive function. The same pipeline applies if we're interested in modeling the throughput of workloads instead of the runtime latency, with the only requirement to swap the last

regression module with a *classification* module that can predict which class of throughputs the (job, configuration) combination belongs to.

The loss function used when training the regressor is the mean squared error:

$$\mathcal{L} = \frac{1}{N} \sum_{i,j} (f(\mathbf{v}^i, \mathbf{z}_j) - y_j^i)^2$$

When the architectures used for representation learning and regression are homogeneous (for example when both are implemented as neural networks if the representation learning is an auto-encoder and the regressor is a simple feed-forward neural network), we can fine-tune the layers of the representation learning architecture while minimizing the regressor's loss function.

We focus on studying different architectures for the representation learning module while keeping the regressor as a simple feed forward neural network.

5.3 Encoder-decoder architectures

Auto-encoders have seen a large body of work in recent years, especially with the ubiquity of powerful hardware infrastructure that enabled faster training of neural networks using back-propagation and new emerging automatic differentiation tools (such as Tensorflow [7, 8], Theano [10] and Pytorch [46]).

We start this section with a simple auto-encoder, and highlight its limitations in satisfying important properties known about our workloads from the domain knowledge. We then introduce a customization of such a deterministic auto-encoder in an attempt to disentangle generating factors within the bottleneck layer. Then, we introduce more advanced auto-encoders from the literature such as the contractive auto-encoder [50] which is also a deterministic auto-encoder and the variational auto-encoder [33, 34] from the family of generative models. We discuss the assumptions and limitations of each approach and propose an adjustment on the contractive auto-encoder to incorporate the domain knowledge again.

Original auto-encoder The auto-encoder was first introduced as a dimensionality reduction technique that makes use of neural networks. It has the advantage of supporting both linear and non-linear activations.

We can use the auto-encoder in order to learn a more compact representation instead of using the full runtime metrics as a representative encoding for each workload. Figure 8 is an example of the abstract diagram we earlier introduced in Figure 7. It shows the auto-encoder as an example of a concrete choice of representation learning technique followed by an aggregator that represents each workload by its centroid vector and a neural network regressor.

The bottleneck layer of the auto-encoder in Figure 8 consists of a latent encoding $\mathbf{z}_j^i$ that contains digested information from the runtime metrics vector $\mathbf{x}_j^i$ provided at the input. This auto-encoder architecture minimizes the reconstruction loss given by:

$$\mathcal{J} = \frac{1}{N} \sum_{i,j} ||\tilde{\mathbf{x}}_j^i - \mathbf{x}_j^i||^2 \tag{10}$$

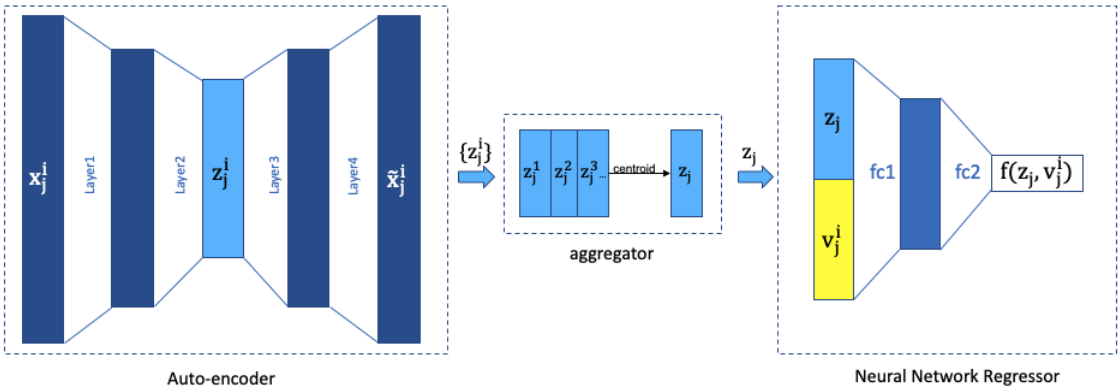


Figure 8: Runtime latency estimation pipeline with a simple auto-encoder as representation learning technique.

where $\tilde{\mathbf{x}}_j^i$ obtained at the output of the fourth layer of the auto-encoder denotes the approximation of the runtime metrics $\mathbf{x}_j^i$ (as depicted in Figure 8). If we use e to denote the encoding function, d the decoding function, then in this case $\mathbf{z}_j^i = e(\mathbf{x}_j^i)$ and $\tilde{\mathbf{x}}_j^i = d(e(\mathbf{x}_j^i))$.

The auto-encoder in this form focuses solely on the *reconstruction* property without providing any guarantees on the remaining properties. For instance, by uniquely having the workload encoding within the bottleneck layer, we can't guarantee that the encoding itself doesn't contain knob information since it is trained on approximating the whole vector of runtime metrics. So we can't guarantee the independence between the learned encodings and the knob vector yielding these encodings. We also didn't force an independence between different components that form the basis of the latent space. Furthermore, once the auto-encoder is trained, its weights are fixed, and thus for a different runtime metrics at the input $\mathbf{x}_j^k$ of the same job j we get a different latent representation $\mathbf{z}_j^k$. This means that the invariance property is violated due to architectural reasons, and this can explain why we have attempt to fix this violation upfront by introducing the aggregator in Figure 7.

Custom auto-encoder Traditional autoencoders are not meant for explicitly disentangling trace generation factors within a simple bottleneck layer. There are indeed five different types of factors that jointly yield a runtime trace vector, 3 of which are invariant with respect to the workload while the remaining two are variant. These factors are: (1) the *workload descriptors* (descriptors of the dataflow graph) (2) the *data characteristics* that depend on the data ingested within the system during the runtime of the workload. These are considered as constant across different executions of the same workload in our experimental setup as the same data is ingested across different runs. (3) the *hardware parameters* that depend on the type of the machines on which the workload is run (how many CPUs with how many cores and from which brand, how much memory, etc...). The hardware parameters are constant across all workloads since we're collecting the traces on identical hardware. (4) the distributed system's *knob configuration* that control the resources allocated for the workload as well as the partitioning and the shuffle behavior. (5) random noise that is due to the distributed environment. For simplicity reasons, we consider (1), (2) and (3) to jointly

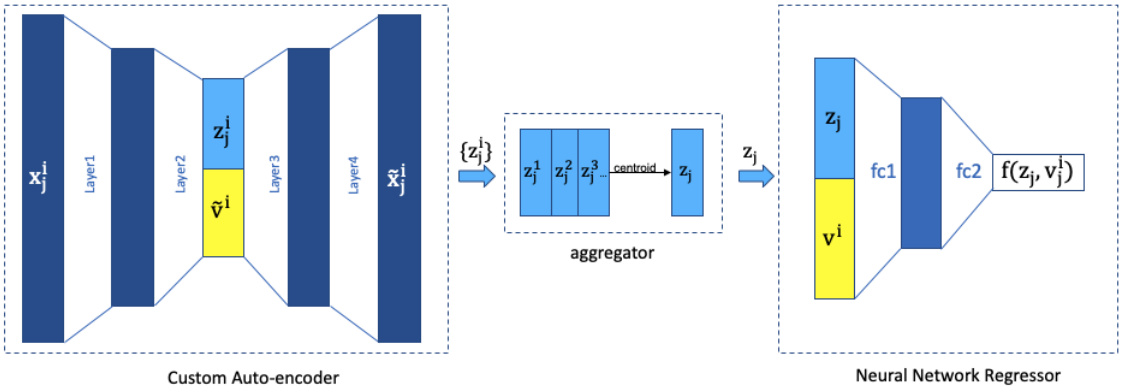


Figure 9: Runtime latency estimation pipeline with custom auto-encoder as representation learning technique.

describe a particular workload since all of these 3 factors remain invariant for different runs of the same workload within our experiments.

With this in mind, we attempt to design a customized auto-encoder that breaks the bottleneck layer into two parts and we present the architecture of this custom auto-encoder within the Figure 9.

The intuition here is to force the encoding function to extract a variant part $e_v(\mathbf{x}_j^i) = \tilde{\mathbf{v}}_j^i$ and isolate it in a separated block of the bottleneck layer (lower part colored in yellow within left diagram in Figure 9). This variant block has to approximate the main variant factors: the knob configuration $\mathbf{v}_j^i$ that is fed to the input of the neural network. Since we can't perfectly approximate the knob configuration while training, we suppose that the error that remains in this approximation corresponds to the random noise that is due to distributed system behavior. Then, presumably, the other part of the bottleneck layer, $e_{iv}(\mathbf{x}_j^i) = \mathbf{z}_j^i$, tend to become less variant for traces belonging to the same workload, and could thus be considered as the workload characteristics that we aim to extract. Such separation within the bottleneck layer favors the independence between the knob configurations and the workload descriptors.

After adding this little supervision, the loss function of the custom auto-encoder now becomes:

$$\mathcal{J} = \frac{1}{N} \sum_{i,j} (||\tilde{\mathbf{x}}_j^i - \mathbf{x}_j^i||^2 + \gamma ||\tilde{\mathbf{v}}^i - \mathbf{v}^i||^2) \tag{11}$$

where γ is a regularization coefficient and $\tilde{\mathbf{v}}^i$ represents the approximation of the configuration provided within the bottleneck layer of the custom auto-encoder architecture when the input is $\mathbf{x}_j^i$. In this setting, the encoder function e is broken into two parts, $e(\mathbf{x}_j^i) = (e_v(\mathbf{x}_j^i)||e_{iv}(\mathbf{x}_j^i))$, where $e_v(\mathbf{x}_j^i) = \tilde{\mathbf{v}}^i$ is an approximation of the generating configuration,

$e_{iv}(\mathbf{x}_j^i) = \mathbf{z}_j^i$ is the workload encoding, and $d(e(\mathbf{x}_j^i)) = \tilde{\mathbf{x}}_j^i$. Note that the part of the bottleneck layer designated to be invariant extracts dataflow descriptors, hardware parameters and data characteristics altogether defining our *workload characteristics* encoding vector.

While this architecture favors the independence between the knob configuration and the workload descriptor by forcing the separation between the two within the bottlneck layer, it doesn't provide any guarantees as to whether the encodings are indeed invariant for the same

workload. Hence in practice, we keep the aggregator as a preliminary step for regression.

Custom contractive auto-encoder In an attempt to reduce the variance within learned workoad encodings, we try to further augment our custom auto-encoder by borrowing ideas from the contractive auto-encoder in the literature [50]. The contractive auto-encoder adds a *contraction* term to the reconstruction loss function of the original auto-encoder. The contraction term is simply the Frobenius norm of the Jacobian matrix of encoder activations.

Our intuition is to force the designated invariant part of the encoding $e_{iv}(\mathbf{x}_j^i)$ to become less variant to input perturbations by adding the contraction term. Hence, we use the same architecture depicted in Figure 9 but minimize this loss function instead:

$$\mathcal{J} = \frac{1}{N}\sum_{i,j}\left(||\mathbf{x}_j^i - \tilde{\mathbf{x}}_j^i||^2 + \gamma||\mathbf{v}^i - \tilde{\mathbf{v}}^i||^2 + \lambda||J_{e_{iv}}(\mathbf{x}_j^i)||_F^2\right) \tag{12}$$

where $J_{e_{iv}}$ is the Jacobian of the encoding output $\mathbf{z}_j^i = e_{iv}(\mathbf{x}_j^i)$ with respect to the input $\mathbf{x}_j^i$. The original paper introducing contractive auto-encoders [50] provide the expression of the Jacobian term when the encoder contains one layer with a *sigmoid* activation. We provide in the Appendix E extensions of the Jacobian term computed up to two hidden layers with both *sigmoid* and *reLu* activations. However, when no activation is added on top of the encoder, the Jacobian term becomes equivalent a simple weight decay regularization.

Variational auto-encoder The so-far introduced auto-encoders are all deterministic. This means that for a particular vector of runtime traces $\mathbf{x}_j^i$ at the input of the auto-encoder, one encoding vector $\mathbf{z}_j^i$ is associated to it. The customizations we brought on top of these deterministic auto-encoders were essentially driven by the need to disentangle different generating factors from the runtime metrics so that the independence property is ensured alongisde the invariance property.

There is however another family of auto-encoders that learn distributions of encodings instead. These are called generative auto-encoders and are known to be good at the automatic disentanglement of generating factors according to a recent survey [60]. Among generative auto-encoders, we are particularly interested in variational auto-encoders [33, 34] (β-VAE [27] more precisely).

The central idea that governs variational auto-encoders is starting with a prior distribution for the encodings, and then updating the distribution using bayesian inference methods (in particular variational inference, hence the name *variational* autoencoders).

The diagram of a variational auto-encoder is depicted in Figure 10. The loss function that is usually minimized while training a β-variational auto-encoder is given by:

$$L(\beta) = -\mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z}) + \beta D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$$

More in depth insights about this loss function are deferred to the appendix within section C. This loss balances between minimizing a reconstruction term and minimizing the KL divergence between the posterior distribution and the prior distribution. The reconstruction term indicates how much the distribution of encodings should trust the observed data, while the KL divergence term indicates how much this distribution of encodings should mimic the prior imposed on these encodings.

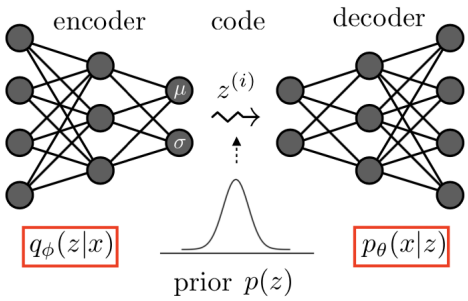


Figure 10: Variational auto-encoder diagram [60].

5.4 Siamese Neural Networks

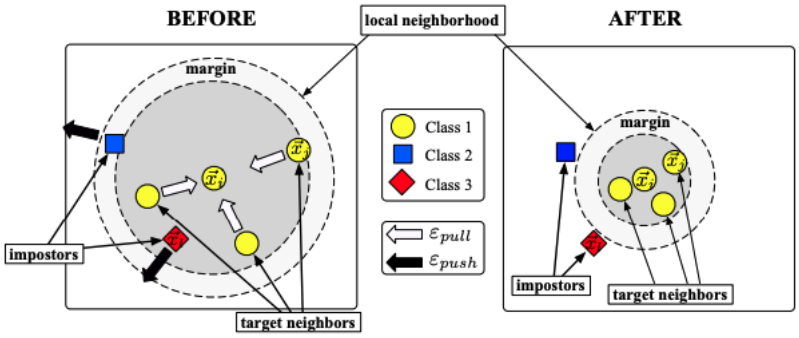


Figure 11: Diagram from [65] showing LMNN effect on local neighborhood. Points belonging to the same class are tightened and those belonging to different classes are further separated in the embedding space.

Earlier introduced auto-encoders focused mainly on both the reconstruction and the independence properties. We also attempted to fix the invariance property by adding an unsupervised jacobian term within the loss function of the custom contractive auto-encoder.

In an attempt to better control similarities between the learned encodings so that the encodings coming from the same workload become less variant to the input configuration, we resort to supervised methods from the metric learning family. The seminal work of Weinberger et al. [65] in the field of metric learning introduced the large margin nearest neighbor (LMNN) loss function. The central idea governing this approach consists of learning a distance function that preserves class similarity such that points belonging to the same class become closer in the new embedding space, and those that belong to different classes are separated by a certain margin as shown in Figure 11. If we consider each workload as a separate class, then we can apply this idea in order to learn more tightened encodings $\{\mathbf{z}_j^i\}$ from traces $\{\mathbf{x}_j^i\}$ belonging to the same workload j.

Recent advances in neural networks led to the conception of better architectures for metric learning. In particular, siamese neural networks have been widely used and had interesting applications in fingerprint detection and face verification for example [54]. All of these

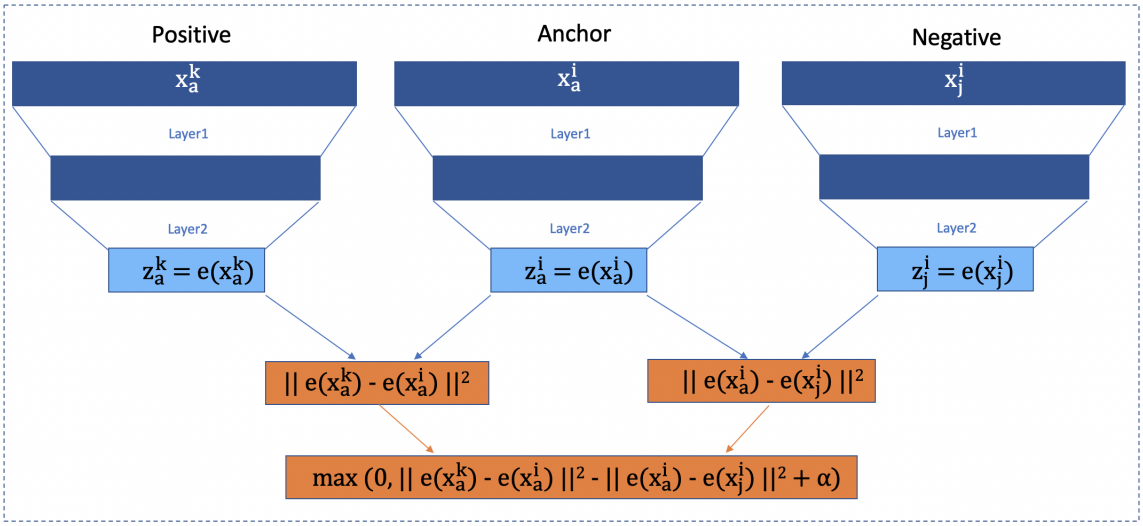


Figure 12: Siamese neural network architecture trained with a triplet loss function.

problems bear similarity to our use case where predictions over new data points is required within a zero-shot learning scheme.

There are multiple ways for training such networks and we focus in particular on two promising loss functions that help in learning encodings less variant to input configurations: (1) The *triplet loss* [54] (2) The *soft nearest neighbor loss* (SNN) [24, 51]

We first introduce the triplet loss by applying it on an encoder architecture, and then we present the soft nearest neighbor loss as part of a hybrid architecture in §5.5

Triplet Loss Training a triplet loss based architecture consists of organizing the data into triplets of:

- Anchor point: $\mathbf{x}_a^i$ which denotes the runtime metrics observed for an anchor job a when the knob configuration is set to a particular value $\mathbf{v}^i$.
- Positive observation point: $\mathbf{x}_a^k$ which denotes the runtime metrics observed for the same anchor job a but with a different knob configuration $\mathbf{v}^k$, instead of $\mathbf{v}^i$.
- Negative observation point: $\mathbf{x}_j^i$ which denotes the runtime metrics observed for a different job $j \neq a$ when the knob configuration was set to $\mathbf{v}^i$, the same as the one used in the anchor point.

Figure 12 shows an example of how the points are organized into triplets.

At the input of the architecture, we provide 3 runtime metrics vectors: $\mathbf{x}_a^i$, $\mathbf{x}_a^k$, $\mathbf{x}_j^i$. The same fully connected layers are applied to get the embeddings from the different observations, and we obtain their respective embeddings: $\mathbf{z}_a^i$, $\mathbf{z}_a^k$, $\mathbf{z}_j^i$. The loss function on this instance of triplets is $L_T(\mathbf{x}_a^i, \mathbf{x}_a^k, \mathbf{x}_j^i)$ and is given by:

$$L_T(\mathbf{x}_a^i, \mathbf{x}_a^k, \mathbf{x}_j^i) = max(0, ||e(\mathbf{x}_a^i) - e(\mathbf{x}_a^k)||^2 - ||e(\mathbf{x}_a^i) - e(\mathbf{x}_j^i)||^2 + \alpha)$$

The final loss to be optimized is the sum over all the instances of triplets:

$$\mathcal{J} = \sum_{a=1}^{n} \sum_{i=1}^{I_s} \sum_{j \neq a} L_T(\mathbf{x}_a^i, \mathbf{x}_a^k, \mathbf{x}_j^i) \tag{13}$$

Where α is a margin hyper-parameter that is tuned alongside other hyper-parameters. The training of this loss function requires I_s shared configurations across all training workloads. However, an arbitrary configuration can be observed for the new workload at inference time.

5.5 Hybrid architectures

In this section, we propose hybrid architectures that add decoders on top of Siamese neural networks.

Hybrid1. We start by augmenting the previous architecture with a decoder in order to add to the triplet loss, additional terms related to our customized disentanglement and reconstruction. We thus minimize this loss function:

$$\mathcal{J} = \sum_{a=1}^{n} \sum_{i=1}^{I_s} \sum_{j \neq a} L_T(\mathbf{x}_a^i, \mathbf{x}_a^k, \mathbf{x}_j^i) + \gamma L_R(\mathbf{x}_a^i, \mathbf{x}_a^k, \mathbf{x}_j^i) + \lambda L_C(\mathbf{v}_a^i, \mathbf{v}_a^k, \mathbf{v}_j^i) \tag{14}$$

with L_T as provided in Section 5.4, L_R is the reconstruction of the *anchor, positive*, and *negative* terms, and L_C corresponds to the configuration approximation for the 3 terms as well:

$$L_R(\mathbf{x}_a^i, \mathbf{x}_a^k, \mathbf{x}_j^i) = ||\tilde{\mathbf{x}}_a^i - \mathbf{x}_a^i||^2 + ||\tilde{\mathbf{x}}_a^k - \mathbf{x}_a^k||^2 + ||\tilde{\mathbf{x}}_j^i - \mathbf{x}_j^i||^2$$

$$L_C(\mathbf{v}_a^i, \mathbf{v}_a^k, \mathbf{v}_j^i) = ||\tilde{\mathbf{v}}_a^i - \mathbf{v}_a^i||^2 + ||\tilde{\mathbf{v}}_a^k - \mathbf{v}_a^k||^2 + ||\tilde{\mathbf{v}}_j^i - \mathbf{v}_j^i||^2$$

with $\mathbf{v}_a^i = \mathbf{v}_j^i \equiv \mathbf{v}^i$ and $\mathbf{v}_a^k \equiv \mathbf{v}^k \neq \mathbf{v}^i$

Hybrid2. In contrast to the triplet loss that samples one positive and one negative point for each anchor point in a batch of data, the **Soft Nearest Neighbor** (SNN) loss [24, 51] uses all the points in the batch to measure the separation between classes. We apply this loss to an encoder layer of the autoencoder so that the joint loss that we minimize has both the reconstruction term and the SNN term.

$$\mathcal{J} = \frac{1}{N} \sum_{i,j} \left(||\tilde{\mathbf{x}}_j^i - \mathbf{x}_j^i||^2 - \lambda \log \left(\frac{\sum\limits_{k \neq i} e^{-\frac{||\mathbf{z}_j^i - \mathbf{z}_j^k||^2}{T}}}{\sum\limits_{\substack{k,l \\ (k,l) \neq (i,j)}} e^{-\frac{||\mathbf{z}_j^i - \mathbf{z}_l^k||^2}{T}}} \right) \right)$$

where λ is a regularization coefficient and T is a temperature hyperparameter.

The soft nearest neighbor term for one training point (whose corresponding configuration is indexed by i and whose corresponding job is indexed by j) is given by (assuming $T = 1$ for now):

$$- \log \frac{\sum\limits_{k \neq i} exp(-||\mathbf{z}_j^i - \mathbf{z}_j^k||^2)}{\sum\limits_{\substack{k,l \\ (k,l) \neq (i,j)}} exp(-||\mathbf{z}_j^i - \mathbf{z}_l^k||^2)} = - \log \frac{numerator}{denominator}$$

The numerator is a sum of negative exponentials of distances between the encoding $\mathbf{z}_j^i$ of the current job j with the current configuration i and all other encodings $\mathbf{z}_j^k$ for the same job j within the same batch but obtained under a configuration k different than the initial configuration i (hence $k \neq i$). So it's a sum of distances between all "positive pairs". The denominator is a sum of negative of exponentials of distances between the encoding $\mathbf{z}_j^i$ and all other encodings $\mathbf{z}_l^k$ coming from different jobs l (hence $l \neq j$) and under different configuration from the current i (hence $k \neq i$).

Minimizing the soft nearest neighbor term requires minimizing the denominator within the logarithm of the above equation and maximizing the numerator within this same equation. The numerator is a sum of positive terms so it can be maximized by maximizing each of its term. Maximizing each $exp(-||\mathbf{z}_j^i - \mathbf{z}_j^k||^2$ is equivalent to minimizing each distance within the exponential term. So we are trying to minimize the distance between encodings coming from the same workload ($\mathbf{z}_j^i$ and $\mathbf{z}_j^k$). On the other hand, the denominator is as well a sum of positive terms, and minimizing the denominator requires minimizing each of its terms. Each term is minimized if the distance inside the negative exponential is maximized. So this corresponds to maximizing the distance between the current encoding $\mathbf{z}_j^i$ and other encodings $\mathbf{z}_l^k$ coming from different workloads under a different configuration within the embedding space.

As for the temperature parameter T, it controls the sensitivity to distances between different embeddings obtained for points within the same batch. If T is very high (close to infinity), then the values of the distances will not be taken into account. Instead, the numerator will be the number of points within the same batch that belong to the same workload, and the denominator will be the number of points within the same batch that belong to a different workload. In other words, under high values of T, this loss function does not ensure tightening encodings obtained with input traces $\mathbf{x}_j^i$ that belong to the same workload j, and thus we shouldn't consider high values of T for our use case. On the other side, if T is very low, then the loss becomes extremely sensitive to distances between the points and a small change in the distance can make a big difference in the value of the SNN function. We tune the temperature T parameter just like any other hyper-parameter while minimizing the loss function.

5.6 Summary

This chapter introduced representation learning architectures from two families of techniques both of which *explicitly* extract encodings from runtime traces before feeding them to a neural network dedicated for the end regression task on the runtime latency. We have covered different auto-encoders starting with deterministic autoencoders and explained why the basic auto-encoder violates both the independence and invariance properties. We have then added edits on top of the architecture and augmented the loss function with supervised and unsupervised terms as an attempt to favor these properties while training the auto-encoders which natively satisfy the reconstruction property. We also briefly covered the variational auto-encoder from the generative family of auto-encoders because it's known for its capacity of automatic disentanglement of generative factors within the bottleneck layer. Furthermore, we have also introduced another paradigm of extracting encodings that relies on siamese neural networks instead of auto-encoders, and explained how to train such networks that

satisfy a relaxation of the invariance property. Finally, we also gave examples of designs of hybrid architectures that combine auto-encoders and siamese neural networks. In chaper 7 we will show how the different introduced representation learning techniques can help in revealing interesting insights from the data that we have collected. We also compare the different approaches quantitatively within this Chapter 7 after we introduce in the next chapter the workloads and traces used for benchmarking.

6 Workloads, Runtime Environment, Sampling and Traces

In this chapter, we present the benchmark of Spark workloads that we use for comparing different modeling techniques. We then introduce important Spark knobs that we tune while sampling traces from these workloads. We also explain Spark heuristics and best practices we resort to while sampling the trace datasets and we finally give a brief overview of the collected runtime metrics within these traces.

6.1 Workloads description

We have developed two benchmarks of traces while running heterogeneous workloads on top of Spark [66]. The two benchmarks of traces were collected while running (1) *streaming workloads* that cover parametrized variations of workloads from a prior work [37] and (2) *batch workloads* which consist of parametrized variations of TPCx-BB workloads [59]. We collected the traces after developing a trace generation system that orchestrates dedicated Spark clusters for profiling. We present the details regarding the streaming trace dataset, and provide only a brief overview of the TPCx-BB traces collected by a colleague who extended our sampling code.

6.1.1 Streaming benchmark

The streaming workloads are parametrized variations from 7 templates: 5 templates cover SQL-like workloads with user defined functions and 2 other templates are variations of ML workloads. The SQL-like workloads operate on top of a clickstream dataset downloaded from the 1998 world cup website [36] while the ML workloads operate on top of synthetic streaming datasets.

SQL-like templates Most of the SQL-like workloads consist of windowed aggregates, so a natural way to parametrize such workloads consists of changing the streaming window size w. We also vary the selectivity δ of the filter at the end of the computation pipeline and add an artificial cpu pressure with a controllable degree π. We provide below the Scala code of the function that simulates the artificial cpu pressure:

```
1 protected def cpuPressure(pressure: Int): Unit = {
2 var x: Double = -4;
3 (1 to pressure).foreach(i => x = x + Math.pow(-1, (i + 1)) * 4 / (2 * i
4 - 1))
5 }
```

This function iteratively computes an approximation of the mathematical constant Π (not to be confused with π the parameter equivalent to pressure within the function). It is called while processing the data within each window of the streaming session.

We provide within this thesis the CQL representation [13] of the streaming SQL-like workloads since this representation is more compact than the full Scala code. Though, this representation doesn't allow us to include the call to the cpuPressure function.

- **Template A** consists of windowed aggregate workloads with selectivity δ, window size w and artificial cpu pressure π control.

```
1 SELECT userId, COUNT(*) as counts
2 FROM UserClicks [range w slide 10s]
3 GROUP BY userId
4 HAVING counts > δ
```

  From this template, we generate 12 workloads with different parameter values.
- **Template B** consists of windowed aggregate workloads with window size w and artificial cpu pressure π control.

```
1 SELECT userId
2 FROM UserClicks [range w slide 10s]
3 GROUP BY userId
```

  From this template, we generate 4 workloads with different parameter values.
- **Template C** consists of global aggregate workloads with selectivity δ and artificial cpu pressure π control.

```
1 SELECT URL, COUNT(*) as counts
2 FROM UserClicks
3 GROUP BY URL
4 HAVING counts > δ
```

  From this template, we generate 6 workloads with different parameter values.
- **Template D** consists of windowed aggregate workloads with selectivity δ, window size w and artificial cpu pressure π control.

```
1 SELECT URL, COUNT(*) as counts
2 FROM UserClicks [Range w slide 10s]
3 GROUP BY URL
4 HAVING counts > δ
```

  From this template, we generate 12 workloads with different parameter values.
- **Template E** consists of windowed aggregate workloads with a join operator to a static dataset. The workloads allow selectivity δ, window size w and artificial cpu pressure π control.

```
1 SELECT userId, SUM(pageRank)
2 FROM Rankings and UserClicks [Range w slide 10]
3 WHERE Rankings.pageURL = Userclicks.URL
4 GROUP BY userId
5 HAVING SUM(pageRank) > δ
```

  From this template, we generate 12 workloads by varying the window size w, the pressure π and the selectivity δ.

Details regarding specific values of the parameters for these workloads are provided within the appendix.

ML templates We extended the streaming benchmark with 2 additional ML templates:

- Template F: a template for classification workloads based on the library streamDM [15]. All of the workloads within this template operate on top of the same synthetic dataset but differ in the choice of the hyper-parameters. The workloads have been parametrized by changing four main hyper-parameters: the learning rate of the classifier, the loss type (logistic or hinge), the regularizer, and the regularization coefficient. More details about the 12 parametrized workloads from this template are provided in Table 10 in the appendix.
- Template G: a template of workloads running the clustering k-means algorithm [12]. The workloads are written in Scala using the library MLlib [45]. The workloads have been parametrized by changing the value of the hyper-parameter k of the K-means algorithm. We also parametrize the workloads by streaming at the receivers either a 3D or 4D synthetic datasets. Details about workloads from this template are provided in Table 11 within the appendix.

We recall that we tune both the SQL-like and ML workloads with the same goal in mind: to reduce the runtime latency of the workloads so that we get computation results as quickly as possible while the data is ingested into the system. That being said, we must shed the light on the fact that the current tuning context differs from the familiar context in which ML workloads are tuned for the only purpose of getting more accurate models regardless of the runtime latency. In this work, however, tuning the ML workloads doesn't refer to changing their hyper-parameters but rather changing the parameters of the distributed system on top of which these workloads are running (the parameters we tune in this work are Spark knobs later introduced in Table 2). The moment we change the hyper-parameters of a ML workload, it is considered a different workload within the same template. This is clarified within the appendix in Tables 10 and 11 which illustrate the parametrizations of the ML workloads. Our current parametrization of ML workloads within the same template is realistic as soon as we consider that the cloud user is submitting the *same* ML subroutine but with different hyper-parameters for example if he found better hyper-parameters and decided to submit the job with the new hyper-parameters. The cloud user's goal remains though to maintain a low runtime latency even if the new hyper-parameters increase the complexity of his ML model.

6.1.2 TPCx-BB workloads

The TPCx-BB benchmark specification includes 30 templates of workloads for batch analytics which can be divided into 14 SQL tasks, 11 SQL with UDFs and 5 ML workloads. From these 30 templates, 1160 parametrized workloads are generated and used for evaluating the modeling techniques introduced in earlier chapters. The orchestration tool which we developed to collect traces from streaming workloads has been extended to sample traces as well from the batch workloads. Since the parametrization of these batch workloads is not our main contribution but rather a contribution of another collaborator, we don't detail it within this thesis.

In the next section, we describe the distributed infrastructure on top of which we have deployed our workloads and collected runtime traces.

6.2 Distributed environment

We have set up 6 Spark clusters on 18 identical compute nodes with exactly the same hardware properties. A cluster spans 3 nodes and has Spark 2.3.1 installed alongside Hadoop 2.9.1. Each node has 2x Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz processors (with 16 cores per processor) and 754 GB of RAM. Note however that we change the resource allocation knobs of Spark while tuning the workloads without changing the underlying hardware. This means that we don't use the full node's capacity when we don't assign the full number of cores and the full memory to the current Spark job.

While running our workloads, we deploy the driver in client mode using Hadoop YARN [1,63]. This means that the driver is deployed locally as an external client not on the worker nodes. Thus, one node gets reserved for the driver, while Spark executors run on the two other remaining nodes of each spark cluster.

The workloads are implemented in Scala, and are submitted to Spark using the command *spark-submit*. This command takes as arguments both: (1) the name of the jar file containing compiled binaries of the workload code as well as (2) knob parameter values.

6.3 Trace datasets

6.3.1 Sampling heuristics and knob selection

After discussing with a Spark expert, we have selected the 10 most relevant knobs and we have tuned these knobs while sampling our traces. We provide the list of knobs within Table 2. These knobs cover 3 main categories:

- *Parallelism related knobs*: These are the knobs that dictate the level of parallelism at both mappers and reducers sides. In particular, we tune *batch interval* denoted as v_1 and block interval (*spark.streaming.blockInterval*, denoted as v_2). v_1 and v_2 jointly dictate the level of parallelism at the mappers side. We also tune the parallelism (*spark.default.parallelism*, denoted as v_3) which refers to the parallelism at the reducers side.
- *Granularity of Scheduling related knobs*: This category comprises nodes controlling: the maximum size of map outputs to be fetched from reduce tasks (*spark.reducer.maxSizeInFlight*, denoted as v_4), the maximum number of reduce partitions that represent the threshold for bypassing mergesort algorithm within the sort-based shuffle manager (*spark.shuffle.sort.bypassMergeThreshold*, denoted as v_5), whether or not to compress map output files prior to shuffling (*spark.shuffle.compress*, denoted as v_6).
- *Resource allocation knobs*: These are knobs that imply the choice of the resources booked for the spark application. For instance, when running a spark in cluster mode and not in standalone mode (which is used for debugging), we need to specify how many cores on the cluster and how much memory should be dedicated to the spark application. In spark terminology, the smallest unit of worker is defined as *executor* and resources should be specified per *executor*. We tune in particular these knobs: (1) number of executors (*spark.executor.instances*, denoted as v_7), (2) number of cores per executor (*spark.executor.cores*, denoted as v_8) (3) memory per executor (*spark.executor.memory*, denoted as v_9).

  These knobs are of particular importance because they also imply the dollar cost for reserving equivalent amount of computation resources in the cloud. We have restricted

Abbreviation	Knob	Technical Name
v_1	Batch interval	-
v_2	Block interval	*spark.streaming.blockInterval*
v_3	Parallelism	*spark.default.parallelism*
v_4	Maximum size in flight	*spark.reducer.maxSizeInFlight*
v_5	Bypass merge threshold	*spark.shuffle.sort.bypassMergeThreshold*
v_6	RDD Shuffle compress	*spark.shuffle.compress*
v_7	Number of executors	*spark.executor.instances*
v_8	Number of cores per executor	*spark.executor.cores*
v_9	Memory per executor	*spark.executor.memory*
v_{10}	Input rate*	-

Table 2: Spark knob abbreviations and names.

A	B	C	D	E	F	G
660000	880000	480000	600000	1200000	1100000	900000

Table 3: Input rate values (records/second) across different templates.

sampling values for these knobs from the choices presented within Table 4. This table provides a mapping from resource choices to candidate EC2 instances (from the compute optimized C5 family) that can fulfill the required resources. The mapping is given as an example so that we can have an idea of a cloud setup that can match the resource configuration. Nevertheless, let's recall that our real deployment of Spark cluster makes use of 2 worker nodes even though we sometimes map the resource configurations to 3 EC2 instances so that we get a sufficient # of cores (see choices 5,6,7,10,11, and 12 in Table 4).

We also control the input rate (denoted as v_{10}) at the level of the source before streaming the datasets into Spark engine, and we choose to fix the value of input rate across workloads within the same template so that we fix the data characteristics. However, across different templates we choose different values of input rate as detailed in Table 3. This knob is not tuned during the optimization process, but is fed as additional information to our models. Thus the dimension of the vector $\mathbf{v}^i$ is 10 within the streaming benchmark.

The TPCx-BB trace tunes the same knobs that we've introduced in Table 2 (except knobs v_1, v_2 and v_{10} which are reserved for streaming workloads). It also tunes 5 additional knobs, 4 of which are specific to batch SQL-like workloads: *spark.memory.fraction*, *spark.sql.inMemoryColumnarStorage.batchSize*, *spark.sql.files.maxPartitionBytes*, *spark.sql.autoBroadcastJoinThreshold*, and *spark.sql.shuffle.partitions*. Hence, $\mathbf{v}^i$ is of dimension 12 within the TPCx-BB trace.

Heuristics for sampling values for the different knobs. We adopt the following heuristics when it comes to sampling traces for streaming workloads:

- We always set the parallelism (v_3) after choosing first the resources on which we'd like to run the workload (knobs v_7, v_8 and v_9). We set the parallelism as 2 or 3 * total number of cores according to Spark best practices [4]. We denote below by β the parallelism

choice	n_exec	nc/exec	exec_mem(GB)	n_ec2_instances	ec2_type	$cost/hour
1	2	2	4	2	c5.large	0.17
2	3	2	4	3	c5.large	0.255
3	2	4	8	2	c5.xlarge	0.34
4	4	2	4	2	c5.xlarge	0.34
5	3	4	8	3	c5.xlarge	0.51
6	4	3	6	3	c5.xlarge	0.51
7	6	2	4	3	c5.xlarge	0.51
8	4	4	8	2	c5.2xlarge	0.68
9	8	2	4	2	c5.2xlarge	0.68
10	6	4	8	3	c5.2xlarge	1.02
11	8	3	6	3	c5.2xlarge	1.02
12	12	2	4	3	c5.2xlarge	1.02
13	8	4	8	2	c5.4xlarge	1.36
14	16	2	4	2	c5.4xlarge	1.36
15	18	4	8	2	c5.9xlarge	3.06
16	24	3	6	2	c5.9xlarge	3.06
17	36	2	4	2	c5.9xlarge	3.06

Table 4: Mappings between executor resources and EC2 resource pricing.

coefficient. The total number of cores is equal to the number of excecutors multiplied by the number of cores per executor ($total\_cores = v_7 * v_8$).

- Batch interval (v_1) values are randomly sampled from {1s, 2s, 5s, 10s} for workloads from templates {A, B, D, E} (because of constraints regarding window interval and slide interval within these templates). For the remaining templates, batch interval values are sampled from {1s, 2s, 3s, 4s, 5s, 6s, 7s, 8s, 9s, 10s}.
- Block interval (v_2) is set after choosing the value for batch interval and the resources allocated for the workload. The ratio of batch interval/block interval determines approximately the number of tasks per receiver per batch [57]. Thus, we choose to set the block interval as:

$$v_2 \leftarrow \alpha * \frac{v_1}{v_7 * v_8}$$

where α is uniformly sampled at random from {0.5, 1, 2}.
Similar batch-specific heuristics have been adopted when sampling traces from the TPCx-BB workloads.

Shared vs arbitrary pool of configurations We distinguish between two pools of knob configurations that we wish to sample for our different workloads. The first pool is a pool that contains traces coming from a common (shared) set of configurations and thus is denoted by the *shared pool*. The second pool on the other side, samples arbitrary configurations for each workload and is thus denoted by *arbitrary pool*. We need both pools of configurations in order to better explain the differences between diverse modeling techniques earlier introduced in previous chapters. The two pools allow us to closely examine the performances of different models given data specific assumptions.

Shared pool: We sample traces from a set of 32 common configurations across all workloads in the streaming benchmark. Some of the configurations fail while running on a subset of the workloads, so we end up with only 16 common configurations across all workloads.

Arbitrary pool: For each of the streaming workloads, we sample traces from 128 arbitrary configurations. Some of these configurations fail as well for some workloads, and we end up with fewer configurations.

Pseudo-codes 1 and 2 explain the sampling procedure for both pools. The pseudo-code assume the existence of these predefined functions:

- get_inputrate_for_job: given a job id, this function returns the corresponding input rate value as provided in Table 3.
- get_template: is a function that takes as input the job id and returns the template id of the corresponding job.
- uniformly_sample_from: is a function that takes as input a set of values, a number of points to sample and a boolean parameter indicating whether or not to replace a sampled value (if $n > 1$)
- sample_resource_choices: is a function that takes a parameter n as input and samples n resource choices out of the 17 possible resource choices given within Table 4
- sample_v4_values: is a special sampling function that first samples one of 4 different buckets, each with a different sampling probability, and then uniformly samples one value from within the sampled bucket. The buckets and their corresponding probabilities are:
  - Bucket #1: $v_4 \in \{k | k \in \mathbb{N} \wedge k \geq 12 \wedge k \leq 36\}$ with probability 0.2
  - Bucket #2: $v_4 \in \{48\}$ with probability 0.5 (because this is the default value)
  - Bucket #3: $v_4 \in \{k | k \in \mathbb{N} \wedge k \geq 64 \wedge k \leq 192\}$ with probability 0.2
  - Bucket #4: $v_4 \in \{k | k \in \mathbb{N} \wedge k \geq 192 \wedge k \leq 480\}$ with probability 0.1
- sample_v5_values: is a function that given a particular value of parallelism, uniformly samples a value between -11 and 11 and add it to parallelism (so that v_5 sampled values at the end are between 4 and 105).

We have open-sourced the dataset of traces collected and put them on: `https://github.com/udao-modeling/code`.

6.3.2 Collected metrics and preprocessing

We profile the workloads by collecting two types of metrics: (1) Spark related metrics (2) OS metrics collected using *Nmon* and parsed using PyNmonAnalyzer [48].

Since we are interested in modeling average performances of distributed workloads over the whole running period, we need to aggregate the traces by taking the mean of the metrics over the execution period. But for some metrics (such as counters), this mean isn't meaningful without additional feature engineering. We provide below a quick overview of the metrics collected from both Spark and Nmon and then we give details regarding the aggregation and additional feature engineering done on top of these metrics.

Spark metrics The online Spark documentation at the moment of writing this manuscript distinguishes between 5 different types of metrics yielded by csv sinks:

- (1) *Gauges* that represent instantaneous measurements of some values.

Pseudocode 1: Sampling configurations for the shared pool

```
Data: job_ids
Result: all_shared_configurations
shared_configurations = [] #shared across all workloads
#samples 4 joint values of resource allocation knobs (v7, v8, v9) from Table 4
#(out of 17 total);
resource_choices ← sample_resource_choices(n=4);
for resource_choice in resource_choices do
    v7, v8, v9 ← resource_choice;
    β ← uniformly_sample_from({2,3}, n=1);
    v3 ← β * v7 * v8; # (Parallelism = β * total cores);
    v1_choices ← uniformly_sample_from({1, 2, 5, 10}, n=8, replace=True);
    v2_choices ← []; # empty list
    for v1 in v1_choices do
        α ← uniformly_sample_from({0.5, 1, 2}, n=1, replace=True);
        v2_choices.append((α * v1)/(v7 * v8)) ;
    end
    v4_choices ← sample_v4_choices(n=8, replace=True);
    v5_choices ← sample_v5_choices(n=8, parallelism=v3, replace=True);
    v6_choices ← uniformly_sample_from([True, False], n=8, replace=True);
    for i in [0, 1, 2, ..., 7] do
        v1 ← v1_choices[i] ;
        v2 ← v2_choices[i] ;
        v4 ← v4_choices[i] ;
        v5 ← v5_choices[i] ;
        v6 ← v6_choices[i] ;
        config ← [v1, v2, v3, v4, v5, v6, v7, v8, v9];
        shared_configurations.append(config)
    end
end
all_shared_configurations = [];
for job in job_ids do
    v10 ← get_inputrate_for_job(job) ;
    for config in shared_configurations do
        all_shared_configurations.append([job] + config + [v10]);
    end
end
```

Pseudocode 2: Sampling configurations for the arbitrary pool

```
Data: job_ids
Result: configurations
configurations ← []; # empty list
for job in job_ids do
    v10 ← get_inputrate_for_job(job) ;
    v4_choices ← sample_v4_values(n=4, replace=True);
    v5_choices ← sample_v5_values(n=4, replace=True);
    v6_choices ← uniformly_sample_from([True, False], n=4, replace=True);
    if get_template(job) in ["A", "B", "D", "E"] then
        v1_choices ← [1,2,5,10]; #restricted choices of batch interval
    else
        v1_choices ← uniformly_sample_from({1,2, ..., 10}, n=4, replace=False);
    end
    #samples 8 joint values of resource allocation knobs (v7, v8, v9) from Table 4
    #(out of 17 total);
    resource_choices ← sample_resource_choices(n=8);
    for resource_choice in resource_choices do
        v7, v8, v9 ← resource_choice;
        # Calculates the total number of cores for this resource choice
        β ← uniformly_sample_from({2,3}, n=1);
        v3 ← β * v7 * v8;
        for v1 in v1_choices do
            α ← uniformly_sample_from({0.5, 1, 2}, n=1, replace=True);
            v2 ← (α * v1)/ (v7 * v8) ;
            for i in [0, 1, 2, 3] do
                v4 ← v4_choices[i] ;
                v5 ← v5_choices[i] ;
                v6 ← v6_choices[i] ;
                config ← [job, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10];
                configurations.append(config)
            end
        end
    end
end
```

- (2) *Counters* whose values are usually monotonically increasing or decreasing throughout the execution of the workload
- (3) *Histograms* that provide statistics for the values obtained within the stream of measurements.
- (4) *Meters* that describe the rate of events over time.
- (5) *Timers* containing the rate of calling a particular subroutine as well as statistics concerning its duration.

Metrics we collected with our older version of Spark (version 2.3.1) are already covered in the recent documentation as of Spark 3.0.0 (see [5] as well as the page provided by *Dropwizard* [6] for more details)

We group Spark metrics into 3 main categories according to how we addressed its preprocessing. We distinguish between:

- *Normal metrics* that include gauge and meters metrics. At each timestamp, the value of the metric can be arbitrary during the whole execution period. For these metrics, we simply take the average across the execution period. An example of such metric is: *driver.StreamingMetrics.streaming.lastCompletedBatch_processingDelay*
- *Counters*: In this category, we are particularly interested in counters that are monotonically increasing or decreasing throughout the execution period. For such counters, we engineer 3 features instead of taking the average value like we did for the so called *Normal metrics*. The features calculated from the data are:
  (1) *average increase*, which is computed as such: we first calculate the differences between values of consecutive timestamps and then we take out the mean of these differences across the whole execution period.
  (2) *rate of increase*, computed as such: we calculate on average how many timestamps it takes for the value to change throughout the whole execution period.
  (3) *n_unique*: This represents the number of distinct values for the original metric across the whole execution of the workload. This feature is useful because accumulators do not change every single timestamp but rather every couple of timestamps.
  An example of such type of metrics is:
  driver.StreamingMetrics.streaming.totalCompletedBatches
- Statistics: these include both the timers and histograms. For most of these metrics, we get 15 statistics from csv sinks for each timestamp. The statistics cover the mean, the min, max, std dev as well as some quantiles and moving averages. Right now, we only consider the mean out of these statistics and take its average value across the whole execution period. An example of such type of metrics is :
  driver.LiveListenerBus.listenerProcessingTime.org.apache.spark.HeartbeatReceiver (timer)
  An interesting direction to explore in the future would be to consider each of these statistics as a separate features for the modeling problem. This will inflate the dimension of the runtime trace vector $\mathbf{x}$ which may force us not to bypass the representation learning step prior to regression, so that we avoid doing the regression in very high dimensions.

After aggregating the metrics provided for each executor across the runtime period, we combine metrics from different executors and calculate the average across all executors. By doing so, we make sure we have a uniform set of metrics no matter how many executors were allocated (when the user puts an arbitrary value for the knob *spark.executor.instances*) prior

to running the workload. We end up with 111 executors related metrics as well as 148 driver metrics, so in total we have 259 Spark related metrics.

Nmon Metrics We first run the PyNmonAnalyzer [48] in order to parse Nmon logs and get grouped metrics within csv files. We end up with 336 Nmon metrics per worker node covering 5 CPU metrics per core (*idle*, *steal*, *sys*, *user*, *wait*) as well as other metrics covering memory and disk. Since we have 2 worker nodes, we have in total 672 Nmon metrics.

So for each configuration $\mathbf{v}^i$ run on a streaming workload j, the collected runtime trace vector $\mathbf{x}_j^i$ comprises 931 metrics in total (672 Nmon metrics for the 2 worker nodes and 259 spark metrics). We further preprocess the traces by metrics which take constant values or metrics for which some traces yield Nan (Not a number). We end up with 561 metrics for each of the streaming workloads. Hence the dimension of each runtime trace vector $\mathbf{x}_j^i$ is 561.

A slightly different preprocessing (in terms of metrics aggregation and feature engineering) for TPCx-bb trace yields 572 metrics before any trimming. We end up with 286 metrics after trimming metrics whose values remain constant.

Our profiling approach can be easily extended if we use newer computing infrastructure that have a GPU for example. For instance, if the GPU vendor is NVIDIA, then additonal metrics can be collected using the command *nvidia-smi*. An example of the GPU metrics that can reflect the behavior of the workloads are: processor and memory clocks (in Mhz), memory usage in percentage, power usage (in Watts), etc... In this case, new models can then be trained using all 3 types of metrics: (1) Spark metrics (2) Os metrics (3) GPU metrics.

6.4 Summary

This chapter has covered 3 main contributions: (1) extending a previous benchmark of streaming workloads with more parametrizations and more workloads (2) Developing a trace generation orchestration tool that runs and profiles workloads from the streaming benchmark as well as TPCx-BB benchmark (3) Using best practices and heuristics from Spark and sampling traces by running spark workloads on our distributed computing infrastructure. In the next chapter, we will use these traces in order to evaluate the different modeling approaches we earlier presented in Chapters 4 and 5.

7 Experiments

In this chapter, we provide experimental results with recommender modeling approaches from chapter 4 and representation learning modeling approaches we outlined in chapter 5 after running them on traces collected from workloads introduced in the previous chapter.

We start by providing details regarding the experimental setup for the different experiments in §7.1. Then in §7.2 we explain our evaluation methodology, the metric we use to compare different models and how we validate different models before evaluating them on the test set. In §7.3 we compare collaborative filtering methods introduced in Chapter 4 with the aim of understanding whether representing the interaction between workloads and configurations as a simple dot product (with or without biases) is a better choice than representing it using a neural network. We then recall the limitations of the different collaborative recommender architectures and their inability to recommend configurations beyond those already observed within training data and motivate the need for a hybrid embedding approach which we compare in §7.4 to other representation learning based approaches from chapter 5. Then in order to assess the robustness of each method, we conduct an ablation study in §7.5 and we try to gradually reduce the number of configurations available for training and we track its effect on the prediction errors while keeping the same evaluation set for test workloads. We also record the fitting time of each representation learning technique under different number of training points so that we report in the same subsection results on the scalability of each approach and the range of errors that it yields. We conclude this subsection with an analysis regarding how scalability can affect critical business decisions such as how many times our system can afford to retrain a global model. Within the same paragraph (§7.5), we briefly explain that a workload mapping step (similar to the one that is used within Ottertune) is not needed with our tuning system. Finally, in §7.6 we conduct an end-to-end experiment in which we compare between our best performing modeling technique and Ottertune [61]. This experiment reveals that our modeling approach can provide better latency improvement than configurations recommended by Ottertune under different scenarios.

7.1 Experimental setup

Traces. We use the two benchmarks of Spark workloads (streaming and TPCx-BB) earlier introduced in the previous chapter in order to conduct our modeling exepriments. We recall that we collected a *trace* for each workload under a particular configuration, covering two types of metrics: (i) Spark related metrics and (ii) OS related metrics.

The streaming benchmark controls 10 knobs ($d_v = 10$ for this benchmark) and the traces cover 70 workloads including 53 training workloads and 17 test workloads (whose parametrizations are available in the appendix within tables 9, 10, and 11) with 128 traces each sampled from the arbitrary pool and 16 traces sampled from the shared pool. The TPCx-BB benchmark tunes 12 knobs ($d_v = 12$ for this benchmark) and includes 30 templates, from which we generated 1160 parametrized workloads. Among them, 928 are used as training workloads including (i) 58 intensively sampled workloads with around 316 traces each arbitrarily sampled (ii) 870 sparsely sampled workloads with around 30 arbitrary traces each. Furthermore, 232 workloads are left for evaluation and each has 30 arbitrary traces. Finally, we have also sampled 15 shared configurations for all TPCx-BB workloads (covering

training and evaluation workloads).

Further preprocessing. We further preprocess the traces by adding a Min-Max scaling layer so that both the knob configuration vectors $\mathbf{v}^i$ and the runtime metrics $\mathbf{x}_j^i$ vector components get scaled between 0 and 1 before being fed to neural networks. This can help in faster convergence and better training of our neural network based architectures [14].

7.2 Evaluation methodology

Evaluation metric. We use the Mean Absolute Percentage Error (MAPE) metric for reporting results for the different modeling methods.

Hyper-parameter tuning. For the encoder/decoder based architectures as well as the neural networks we tune topology, optimization and other hyperparameters (such as coefficients within loss functions) by using a 5 fold cross validation scheme that simulates the same training settings as in practical cases (observing few configurations for workloads in the left out fold). More details about hyper-parameter tuning are provided within section D of the appendix.

Fine tuning representations. With our best performing representation learning technique, fine-tuning the weights of the encoder while trainnig the regressor on the end task didn't bring improvements over training the regressor separately without further fine tuning.

Implementation and hardware details. We have implemented the recommender based architectures (matrix factorization and advanced matrix factorization), all encoder/decoder based architectures as well as the neural network regressor using Tensorflow [7, 8]. We adopt the tied-weights case implementation of auto-encoders as recommended by [14]. We have implemented the embeddings approach (*dual embeddings* and hybrid embedding approach) using Keras [18]. We use open source implementations from scikit-learn [47] for the baseline representation learning techniques (PCA and KPCA [53]) to which we compare.

We have a dedicated cluster of 20 nodes for training our models. Each node has 2 x Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz processors (with 16 cores per processor) and 754 GB of RAM.

7.3 Comparative results of recommender architectures

	Streaming Trace	**TPCx-BB Trace**
Matrix Factorization	40.17 ± 18.08	8.16 ± 7.73
Advanced Matrix Factorization	21.12 ± 8.98	7.53 ± 7.63
Dual Embeddings	24.74 ± 20.52	13.26 ± 7.44
Hybrid Embeddings	49.71 ± 21.90	10.26 ± 6.78

Table 5: Runtime latency MAPE with different recommender approaches averaged over 10 runs. The comparison so far is limited to previously seen configurations.

	Streaming Trace	TPCx-BB Trace
Matrix Factorization	1.001s	3.568s
Advanced Matrix Factorization	1.163s	3.667s
Dual Embeddings	0.9503s	6.602s
Hybrid Embeddings	1.159s	9.154s

Table 6: Incremental training average ovearhead (averaging over 10 runs for each test job).

In this section, we focus on comparing performances of two promising recommender systems approaches from the collaborative filtering family of methods we introduced earlier in Chapter 4, namely the matrix factorization which uses the dot product as a way to account for the interaction between the workload and the configuration, as well as neural based recommender architecture (called *dual embeddings*) which models the interaction using a neural network after concatenating both latent representations. For this specific study, we train the approaches while using all configurations from shared and arbitrary pools corresponding to our training workloads. However, for each test workload, we only observe 5 shared configurations and trigger incremental training using these 5 configurations. We evaluate the results on top of the 10 other remaining not observed shared configurations for each test workload.

While training our models, we found that the matrix factorization version that adds biases to the dot product ($\gamma = 1$ within equation 8) gave better results on top of the streaming trace, while for TPCx-bb trace, we got the best results with the matrix factorization approach without adding biases. When admitting a new workload we experimented with the folding-in [52] technique but we found that randomly initializing a new latent representation and then incrementally training the matrix factorization and updating the representations worked better in our use case.

The results are shown within Table 5 and reveal that matrix factorization techniques work better than more complex neural network architectures when it comes to recommending one of the previously seen configurations. We attribute this finding to the fact that our traces have right now limited number of configurations. For instance, we have 6026+16 unique configurations for the streaming workloads and 6272+15 unique configurations for the tpcx-bb trace, whereas large scale recommender systems where more complex models (such as neural networks) yield better results than traditional matrix factorization techniques, usually have much more items to recommend (configurations in our context). Moreover, we found that allowing more degrees of freedom while training the matrix factorization approach (introducing a middle matrix S) and randomly initializing all 3 matrices Z, S and V instead of initializing them using SVD decomposition gave better results than the original version of matrix factorization.

Table 6 presents the time it takes to trigger incremental training required as soon as a new workload is admitted into our system. We can see that the incremental training time incurs a small overhead since it's on the order of few seconds for a workload from the TPCx-bb benchmark and nearly 1 second for a workload from the streaming benchmark.

We have also tried to replace the *dual embeddings* architecture with a hybrid collaborative-content based embedding architecture (the one shown in Figure 6) and we've observed that this has yielded lower errors on the TPCx-BB trace (10.26 ± 6.78) than the dual

	Streaming Trace				TPCx-BB Trace			
	Shared Pool		Arbitrary Pool		Shared Pool		Arbitrary Pool	
	5 obs	1 ob	5 obs	1 ob	5 obs	1 ob	5 obs	1 ob
All metrics (scaled)	11.9	10.9	34.8	34.9	**7.2**	**7.6**	8.6	29.6
PCA	11.4	11.3	24.4	60.7	11.9	16.4	70.1	50.8
KPCA	**8.5**	**9.9**	17.9	21.3	35.2	42.8	59.0	58.8
Hybrid Embedding	32.8	-	22.5	-	14.7	-	12.4	-
Custom AE	16.0	13.0	20.2	21.4	16.9	22.2	19.2	49.9
Custom contractive AE	10.6	12.2	13.0	19.7	9.7	14.3	28.9	53.0
VAE	**8.5**	11.2	17.7	18.7	11.4	14.6	28.4	37.5
Siamese Network (triplet)	10.6	12.6	**9.6**	**11.6**	7.7	7.9	6.5	**9.5**
Hybrid1	11.4	12.0	27.0	11.9	**7.6**	8.2	**6.2**	9.7
Hybrid1($\lambda = 0$)	10.3	11.5	10.5	12.6	**7.6**	**7.6**	6.3	9.6
Hybrid2	9.9	12.4	11.2	12.8	7.9	8.3	6.8	10.7
Ottertune (default)	83.7	84.0	67.6	95.5	52.1	44.6	42.2	61.2
Ottertune (tuned)	50.8	63.8	36.8	67.8	41.0	33.5	35.2	38.2

Table 7: Runtime latency MAPE computed over test sets and averaged over 10 runs.

embeddings architecture but couldn't find hyper-parameters that allow better performances on the streaming trace. Nevertheless, this hybrid architecture allows us to make predictions over new unseen configurations and allows us to make predictions if the new job is admitted with an arbitrary configuration, while matrix factorization and the dual embeddings technique fail to provide any prediction if the configuration is new and never seen before within our training data. As such, we adopt the *hybrid embeddings* approach in the next section when we compare recommender based approaches to other modeling approaches that explicitly learn an encoding from runtime metrics before training a neural network regressor on the runtime latency estimation task.

7.4 Comparative results of representation learning based techniques

We provide the main comparative results between different modeling techniques in Table 7. We start with results from a baseline called "*all metrics*" that bypasses representation learning and uses the whole vector of trace $\mathbf{x}_j^i$ as the encoding for the workload ($\mathbf{z}_j^i = \mathbf{x}_j^i$ in this case). Then, we introduce two other baseline methods from the early literature of representation learning: Principal Component Analysis *(PCA)* and *Kernel PCA* [53] and use them as an encoding extraction tool instead of neural based auto-encoders. Then, we list the results obtained with the previously introduced neural network modeling techniques: (1) *Hybrid Embedding* architecture introduced in §4.3 (2) *Custom autoencoder*, (3) *Custom contractive autoencoder* and (4) *Variational autoencoder* from §5.3, and (5) the *siamese neural network* from §5.4. We also list results from the 2 *hybrid* representation learning methods introduced in §5.5. Finally, we compare to the state of the art tuning tool, Ottertune [61, 68] earlier introduced in the related work in §2.1 and which doesn't learn any workload encoding, but instead trains a separate model for each workload and then maps the test workload to one of the training workloads before making predictions.

Encoding Extraction Scheme. We consider two schemes for extracting encodings from configurations: (*a*) *shared scheme*: $\mathbf{z}_j$ is extracted from traces coming from a shared pool of configurations (averaging $\{\mathbf{z}_j^i\}_i$ with i selected from the shared pool). (*b*) *arbitrary scheme*: $\mathbf{z}_j$ is extracted from traces coming from an arbitrary pool of configurations (averaging $\{\mathbf{z}_j^i\}_i$ with i selected from the arbitrary pool).

We also distinguish between extracting the encoding for test workloads with either 1 or 5 observations, under each of the above (*a*) and (*b*) schemes, as shown in the header of Table 7.

It is worth noting that the *arbitrary scheme* is more practical than the *shared scheme* since a cloud optimizer must expect receiving an *arbitrary* configuration for a newly submitted job. The modeling problem becomes even harder when only 1 trace is observed for the test workload. Nevertheless, we explicitly make the comparison between the two schemes in Table 7 to better understand which modeling technique works best under different job admission settings, and we color the most practical case *(arbitrary, 1 ob)* in Table 7

We make the following observations from Table 7 and profiling results in Fig. 13 (different colors within this figure represent different templates of workloads):

1. **Baseline Methods.** If we bypass representation learning techniques and directly train a global regressor model on the runtime metrics $\mathbf{x}_j^i$ (but taking their job centroid $\mathbf{x}_j$) alongside the input configuration(s) $\mathbf{v}_j^i$, then we can get low errors on the latency estimation if we guarantee having seen a job configuration from the *shared pool*. Similarly, PCA and KPCA, two basic representation learning techniques, also work well under the same *shared scheme.* These baseline methods, however, fail to work when a job is admitted by the system under an *arbitrary* configuration. A closer look at the encodings obtained with KPCA applied on raw metrics $\mathbf{x}_j^i$ in Figure 13 shows how encodings from different job templates are scattered in the 2D space and thus clearly violate the invariance property. This behavior is actually expected from PCA based methods since a PCA consists of projecting the data on axis such that the variance is maximized across all training points. We have nevertheless considered comparing both PCA and KPCA to our proposed neural based architectures, since both can be considered as particular cases of auto-encoders and are widely used as baseline representation learning techniques.

2. **Autoencoders.** The *custom autoencoder* fails to provide better performances than baseline methods under the different schemes. Its design, which mainly focuses on reconstructing the variant part by adding a supervision term to the reconstruction loss function, fails to offer the invariance property in the other designated part of the bottleneck layer. This insight is verified in Figure 13: while encodings learned from the custom autoencoder have better clustering properties, according to different jobs, than those learned from a basic autoencoder or KPCA, they are still scattered and not tight enough along each job's centroid.

Further adding a *contractive* term on top of our custom autoencoder provides consistently better results across all encoding schemes for the streaming trace, but only under shared scheme for the TPCx-BB trace. The *contraction* is induced by adding the Frobenius norm of the Jacobian matrix in Eq. 12. This additional unsupervised term hence doesn't condition the invariance of encodings according to each specific workload, but rather affects all workloads encodings by contracting them at once as seen in Figure 13. We attempted to overcome this problem by sharding the training data into batches of traces coming from the same workload and then training the architecture on these custom batches. The intuition here is to constraint the contraction of the encodings and only apply contraction if encodings come

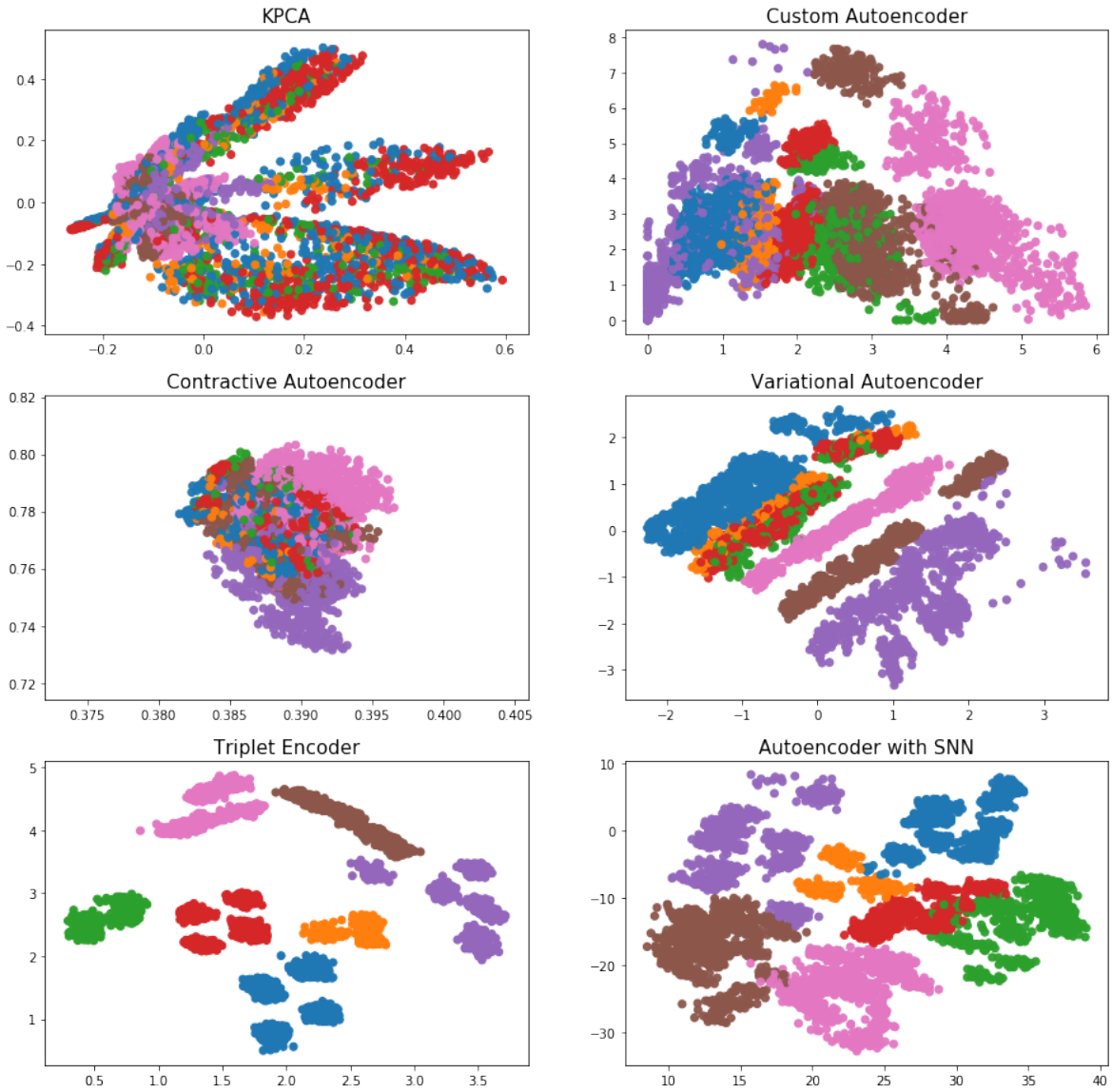


Figure 13: 2D encodings obtained with different encoding/decoding techniques using the streaming trace dataset.

from the same job. This however didn't improve the results, and we got similar clustering properties for the learned encodings as the ones learned while training on randomly sampled batches out of the training data. The reason is that our loss function still didn't include a term that forces separating encodings extracted from traces of different jobs.

On the other hand, the *variational autoencoder* further improves the errors on the streaming trace, but doesn't bring improvements on the TPCx-bb trace, especially when it comes to the *arbitrary scheme*. By examining the encodings obtained from this approach we see similar clustering properties as the one induced by our custom disentanglement in Figure 13.
Within the subplot corresponding to variational auto-encoder within Figure 13, each dot represents an encoding obtained for a particular workload with a particular configuration. Although we're learning a distribution of encodings, we have considered the mean of the distribution that is learned for a particular workload under a particular configuration as

$\mathbf{z}_j^i = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x}_j^i)}(\mathbf{z}|\mathbf{x}_j^i, \phi)$ where ϕ are the parameters of the neural network that approximates the posterior distribution $q_\phi(\mathbf{z}|\mathbf{x})$. An interesting insight from the encodings learned for the variational auto-encoder is that the prior structure is still reflected on the obtained encodings which appear within a circle centered at the origin and having a radius equals to 3 (equivalent to 3 times the standard deviation of the normal distribution of the prior).

3. **Siamese neural networks** focus on a relaxation of the invariance property and achieve drastic improvements on the errors obtained in the most constrained (challenging) setting of observing 1 *arbitrary* configuration for an admitted job, under both streaming and TPCx-BB datasets. The success of this architecture is attributed to its capacity not only to tighten encodings from traces of the same workload but also to separate encodings of different workloads, and thus focusing on learning a more invariant encoding for each workload.

4. **Hybrid methods.** Augmenting the triplet loss function with a reconstruction term and a custom disentanglement didn't bring improvements beyond those achieved with the siamese neural network alone. Indeed, while tuning the hyperparameters of the loss function in Hybrid1, we found that γ was assigned a small value for the best hyperparameters chosen, which indicates that the loss function puts less emphasize on the reconstruction term. Further, by closely examining the results obtained with Hybrid1 and Hybrid1 ($\lambda = 0$), we can conclude that the invariance property subsumed independence in our problem settings across the two datasets. The supervised triplet loss function gave indeed consistent results on the test sets no matter how many (1 or 5) and from which pool (*arbitrary* or *shared*) configurations were sampled. The second hybrid loss function, which combines a soft nearest neighbor term (a more recent metric learning method) and a reconstruction term, provides error on the same scale as the first hybrid loss function when λ is set to 0.

5.**Ottertune** [61, 68], the state-of-the-art tuning tool for RDBMS previously introduced in *Related Work* under §2 does not leverage traces from different workloads or use representation learning techniques to train a single model. In contrast to our approaches, it trains one model per workload and then forces mapping each test workload to one of the past training workloads in order to model its performances. This leads to higher errors across the different training settings under both datasets using the default hyper-parameters from Ottertune. Even when we tried to tune the length-scale, magnitude and ridge parameters of GP models within Ottertune, the error order with Ottertune remains high compared to other approaches.

6. **Hybrid Embedding.** The *hybrid embedding* approach (hybrid between collaborative based and content based methods) we introduced in Figure 6 under §4.3 doesn't fully use the raw metrics $\mathbf{x}_j^i$ to extract an encoding upon the admission of a new job. Instead, it learns an embedding by backpropagating the least squares loss that focuses solely on the actual runtime latency y_j^i to learn a unique encoding $\mathbf{z}_j$. While experimenting with this approach, we have tried to pretrain the weights of the architecture alongisde the embeddings by first trying to approximate all the runtime metrics at the output, and then trimming the last layer and refitting on the runtime latency uniquely (by fine-tuning the layers that are kept and updating the new layers using backpropagation) but this didn't bring improvements on the current results.

Although this approach fully satisfies the invariance property, it remains inferior to other neural based approaches grounded in representation learning. This is because it leverages less information while learning the workload encoding. Despite that fact, it still outperforms

Ottertune. Since the embedding approach requires incremental training before being able to predict, we train it with a number of observed data points at least equal to the degrees of freedom (the number of components) of the embedding vector. Therefore, we don't apply this approach when having only 1 observation since we consider it will have a poor performance.

7.5 Ablation, Scalability and Mapping studies

Ablation. We further add another experiment in order to understand how robust each of the different methods is with respect to the number of training configurations per workload.

We recall that in a real tuning system, we don't expect the user to tune many configurations, and this is the reason for having only 1 or 5 observations for each test workload prior to making prediction, as shown within Table 7. That being said, we have previously assumed that our optimizer has collected more configurations for training workloads in the offline runtime session (as explained in §3.1) and the errors we reported previously are calculated by taking into consideration that we have 128 traces per training workload within the streaming trace, and 30 traces per workload for the TPCx-BB (except a few intensively sampled training workloads for which we have 316 traces). We would like to understand the impact of decreasing the number of training configurations per training workload on the prediction errors if we don't change the evaluation set for test workloads. Such insights could help us decide whether sampling additional configurations for our training workloads can help in making better predictions on the test workloads.

The main results are within Figures 14 and 15 for both the streaming and TPCx-BB workloads respectively. In both Figures, the x-axis refers to the number of training points per workload [4] while the y-axis refers to the errors obtained on the test set. We use the same evaluation metric (MAPE) as before.

For the streaming trace, reducing the number of configurations per workload below 120 configurations yields higher errors on the test set across different representation learning approaches and for different encoding schemes. Generally speaking, and across the different representation learning techniques, reducing the number of training points per workload yields gradually higher errors. Despite that fact, the triplet encoder and the SNN remain the best two approaches among representation learning based architectures up to 40 training points per workload. Below this value, we notice a difference in the behavior as the error with the triplet encoder slightly increase while the errors under SNN are much higher under 20 and 10 training points. This observation is directly linked to the way we feed the data to these 2 architectures. While for the siamese neural network we prepare the data into triplets before training, within the auto-encoder trained with SNN, we compute the *positive pairs* and the *negative pairs* within each batch. This has led to instabilities while training the SNN auto-encoder under few training points per workload. In particular, by inspecting the different auto-encoders trained on both the reconstruction and the SNN loss with only 10 or 20 training points per workload, we noticed that the representation learning model architecture has failed to start because of exploding gradients. The encodings that have been later on fed to the neural network regressor are hence the encodings that were randomly

[4]The 316 x-tick for TPCx-bb workloads in Figure 15,refers to the number of training points for the few offline workloads that are intensively sampled, while for the remaining workloads we still have only 30 points.

initialized before attempting to train the SNN autoencoder. This explains the higher errors that we obtain with SNN under few training points per workload. Hence, when we need to consider whether to train a triplet loss or a SNN loss within our system, we need to pay attention to how many training points do we expect to sample per workload. We'll see in the next paragraph (same section §7.5 but paragraph **scalability.**) that the more stability guaranteed by the triplet loss (and how we fed the data to the network) comes at the cost of a higher fitting time: the triplet loss is 1-2 orders of magnitude slower than its SNN counterpart.

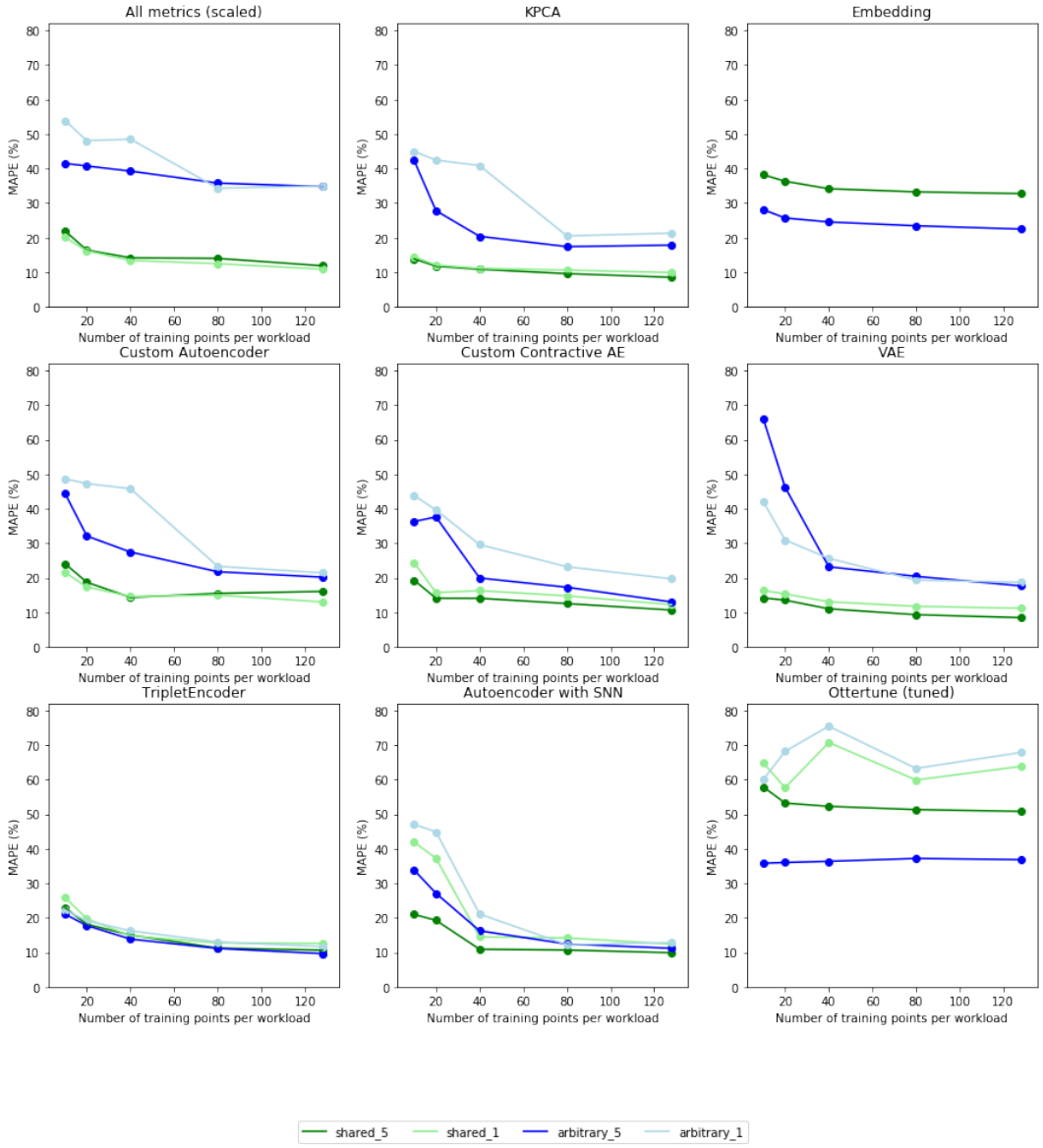


Figure 14: Ablation study on different modeling techniques over the streaming trace dataset. Number of arbitrary configurations per workload has been decreased from 120 to 10.

For TPCx-BB ablation plots within Figure 15, the trends are different. On one side, and in

contrast to what we expected, reducing the number of training points of intensively sampled workloads with 316 configurations to 30 points per workload led to lower errors with KPCA and the custom auto-encoder. By closely inspecting the trained models, we noticed that the more training points, the more models get underfitted (meaning the models could not fit the training data and have as such high training error) and the higher test errors. So decreasing the number of training points led to a decrease in the test errors because the models were able to better fit fewer data points where the encodings do not characterize well the complex behaviors of the workloads.

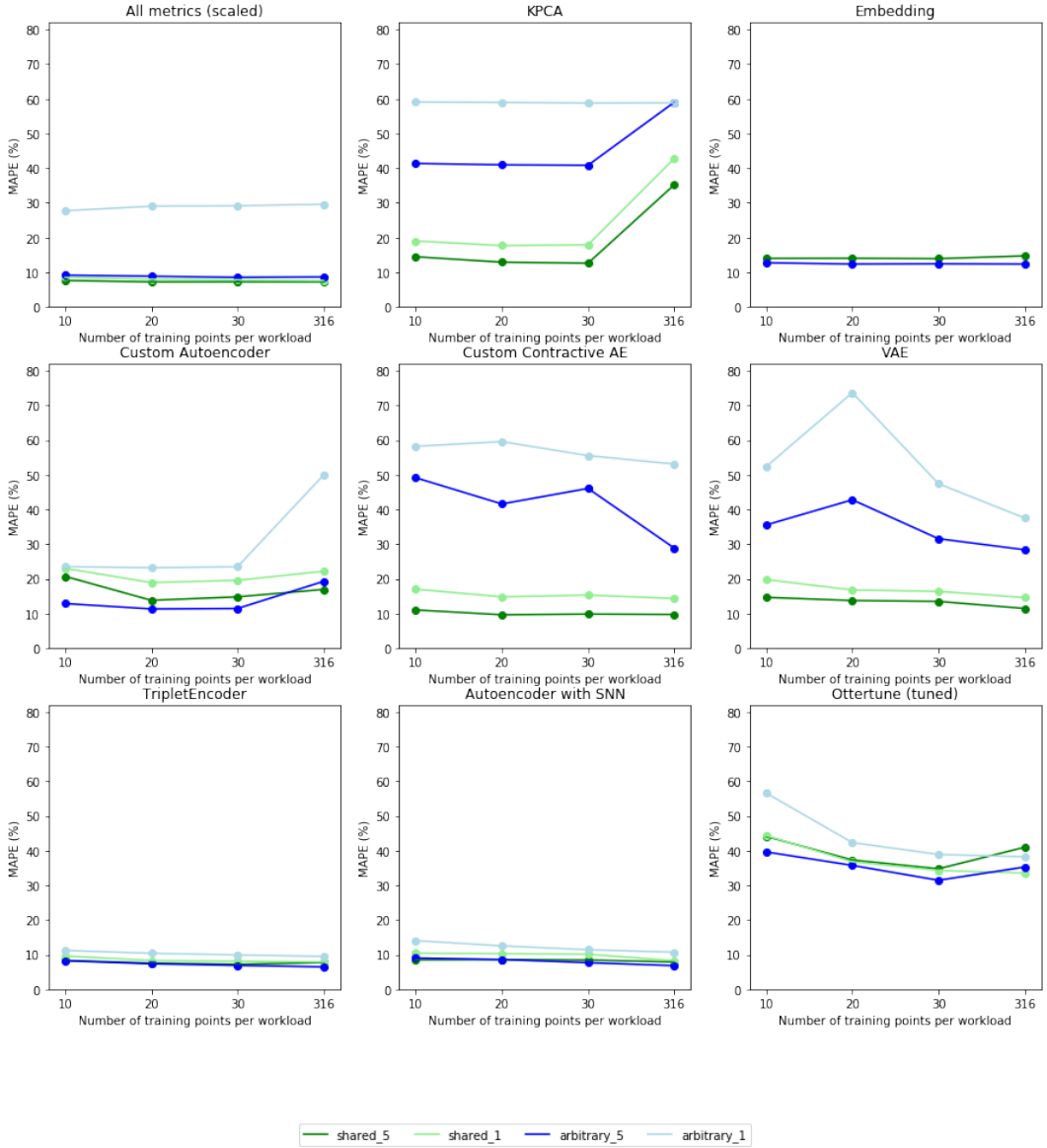


Figure 15: Ablation study on different modeling techniques over the TPCx-BB trace dataset. Number of arbitrary configurations per workload has been decreased from 316* to 10.

On the other side, further reducing the number of training points across each training

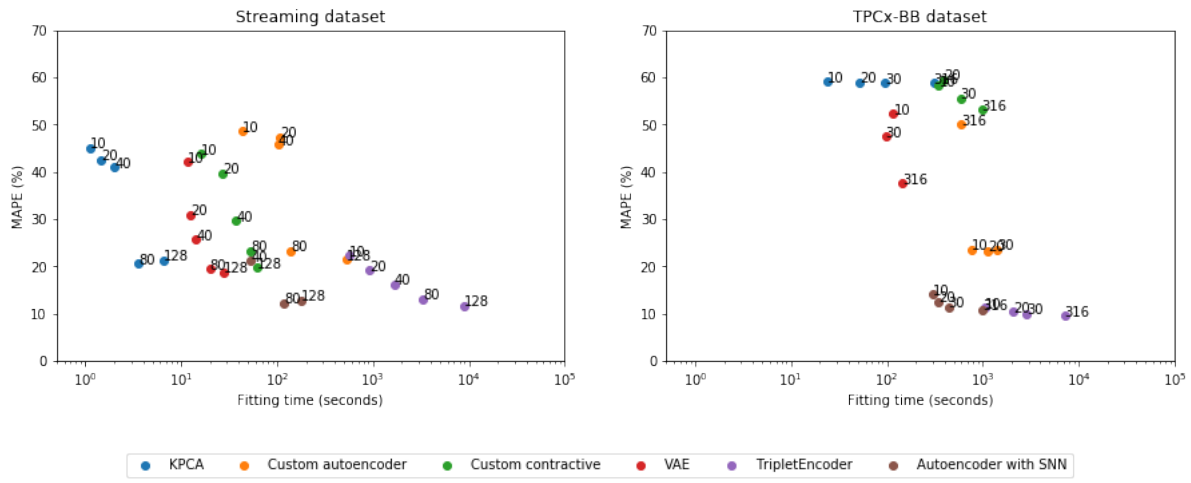


Figure 16: Scalability plots for models trained for 1 arbitrary configuration admission scheme.

workload (so that we have below 30 configurations per training workload) does not yield to remarkably higher test errors across different modeling techniques under different job admission schemes on workloads from the TPCx-BB dataset. Moreover, with our best performing techniques (triplet encoder and autoencoder trained with SNN), no difference is noticed if we further reduce the number of training points to 10 configurations per training workload. We didn't perceive these trends within the streaming trace. We attribute this observation to the fact that we have much more parametrized workloads within TPCx-BB dataset rather than the streaming dataset. Moreover, we also notice that our best performing techniques reduce the gap between the most challenging job admission scheme (observing 1 arbitrary configuration) and the remaining other schemes even under small number of training points. While the SNN had some issues within the streaming trace under few training points, we didn't have the same problem because even under few training points per workload, we already have a large number of workloads within the TPCx-BB trace.

As for Ottertune, we notice that across both datasets and under fewer training points, it still maintains high errors compared to our best techniques.

Finally, this ablation study has revealed that having around 5000 or above data points (# of training workloads * # of configurations per workload) results start to get consistent if we fix the test datasets. This is because under the streaming dataset which has 53 training workloads, as of 80 configurations per workload start to give as good results as using the full data points (128 per workload). Similarly for TPCx-BB dataset which has 928 training workloads, we have noticed that increasing the number of configurations per workload beyond 10 doesn't improve the prediction power of our best modelling method.

Scalability. We recall that adding a representation learning step is crucial whenever the system needs to admit a job with an arbitrary configuration. However, we have clarified previously that our optimizer should be able to make predictions in a zero-shot scheme. This means that at inference time, no retraining is required and we should directly use already fitted architectures to make predictions. That being said, it's important to analyze the overhead that the representation learning step incurs so that we can take critical design

decisions such as: (1) with which frequency the models need to be retrained and updated throughout the day (2) which models are best suited when the optimizer admits a lot of transactional workloads throughout the day vs only few analytical long running workloads, etc. Such an analysis also should allows us to understand the computation budget that needs to be spent for keeping the models up to date.

Plots provided within Figure 16 show the *MAPE* of the different models vs the fitting time of the representation learning step. Each representation learning method is represented with a different color, and each dot within the scatter plots appears alongside a number that refers to the # of training points per workload that are kept while fitting the representation learning architecture.

Furthermore and as expected, the more number of training points we have, the more time it takes to fit the same architecture across both datasets. However, different modelling choices span different ranges of fitting time. We notice that across both datasets, the fitting time of different models is not related to the errors obtained (if we don't consider the particular case of the triplet encoder). In other words, if a particular model is complex so that it takes a lot of time to fit, this does not guarantee that this model ensures lower errors. For example, the custom auto-encoder takes more time to fit than the SNN auto-encoder, and yet the errors obtained with representations extracted from the SNN auto-encoder are lower than their custom autoencoder counterparts. Also, despite the fact that our best performing tools, the triplet encoder and the SNN autoencoder, are on par with respect to the modeling error across both datasets, they have 1-2 orders of magnitude of difference in terms of fitting time.

These insights are interesting because if the tuning system admits a lot of new jobs throughout the day and thus requires more frequent retraining, we may favor a SNN autoencoder training rather than training a triplet encoder. On the other side, if we know upfront that the system will only admit few workloads (with probably few data points), then it becomes better to train a triplet encoder to avoid having gradient problems as explained earlier within the first part of §7.5 (under paragraph entitled Ablation.)

Workload mapping. Across the different proposed methods that learn an encoding $\mathbf{z}_{n+1}$ for a new admitted workload, we have attempted to add a mapping step right after the aggregator (which we introduced previously in Figures 7, 8 and 9). The inspiration of this idea came from Ottertune which applies as well a mapping step but uses a different procedure for doing so. Our mapping consists of calculating the distances between the current workload's encoding $\mathbf{z}_{n+1}$ and all other encodings $\mathbf{z}_j$ of training workloads, and then borrow encoding from the nearest workload j' (the workload that minimizes $j' = \underset{j \in \{1..n\}}{\operatorname{argmin}} ||\mathbf{z}_j - \mathbf{z}_{n+1}||^2$) and feed $\mathbf{z}_{j'}$ to the regression function instead of feeding $\mathbf{z}_{n+1}$ prior to prediction.

Figure 17 shows how *workload mapping* and *encoding borrowing* works. We have found that the workload mapping step does not improve the prediction accuracy across the different methods, and thus we have dismissed it and do not consider any workload mapping step in our tuning system.

7.6 End-to-End experiments

In this section, we conduct an end-to-end experiment in order to compare our tuning system with the state of the art tuning tool Ottertune [61]. We use our best modeling technique

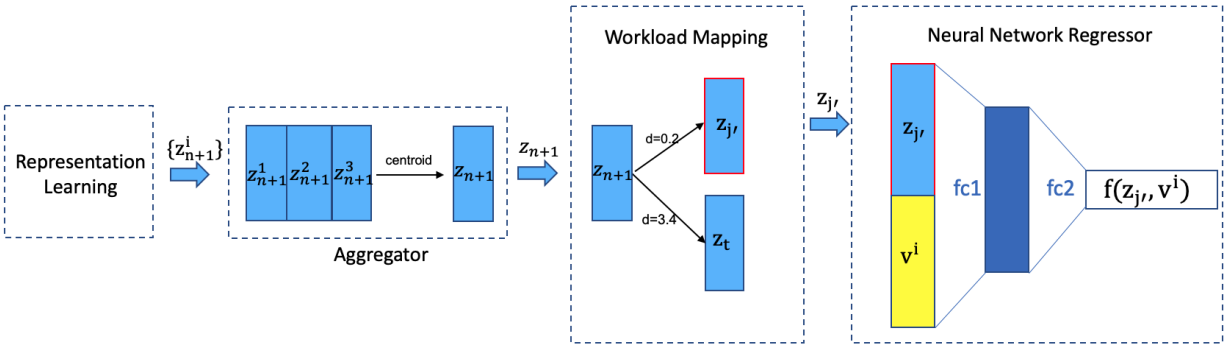


Figure 17: Mapping the new workload *n+1* to its closest neighbor j'.

(siamese neural network) in order to drive this end-to-end experiment while observing a single arbitrary configuration for each test job. This initial arbitrarily set configuration is not necessarily the same across different test workloads, because we assume in a real world scenario it should correspond to an arbitrary configuration initially set by the engineer running the analytics on top of Spark.

Since in this thesis we are mainly interested in studying the impact of our modeling techniques on the end-to-end performances we simply consider an exhaustive search optimizer (which is a naive way for doing optimization) that make use of our models in order to predict the value of the latency. The exhaustive search optimizer enumerates combinations of different knob choices, then runs them through the neural network regressor after feeding a workload encoding extracted for each test job using the siamese neural network. The optimizer then recommends, for each test job, the configuration that minimizes its runtime latency.

In the next paragraphs, we explain two ways for conducting our end-to-end experiments: in the first experiment, we allow both our optimizer and Ottertune to freely control all knobs (including the knobs v_7, v_8 and v_9 that control the resources allocated as shown in Table 2). In the second experiment, we consider a scenario in which the user has a fixed cloud computation budget, and can't afford to upgrade the resource, hence we fix the resource knobs and allow the optimizer to tune only the remaining knobs.

In both experiments, we record the runtime latency for the recommended configuration[5], and then compute the average of latency improvement over the initial configuration, $(1 - \frac{\text{new latency}}{\text{initial latency}})$, across the different workloads.

Expt1: Freely upgrading resources. Figure 18 gives us direct insights on the distribution of speedup recorded for the runtime latency of workloads from both benchmarks. The two leftmost plots show histograms for average speedups with our optimizer on test workloads from both streaming and TPCx-BB datasets respectively. The two rightmost plots show histograms for average speedups recorded with Ottertune as an end-to-end comparative system.

On average, we achieve a latency improvement of 52.4% on streaming workloads and

[5]The optimizer's recommendation is sometimes too optimistic due to extrapolation in a sparse search space. If the job fails to be launched with the recommended configuration, the optimizer recommends another one.

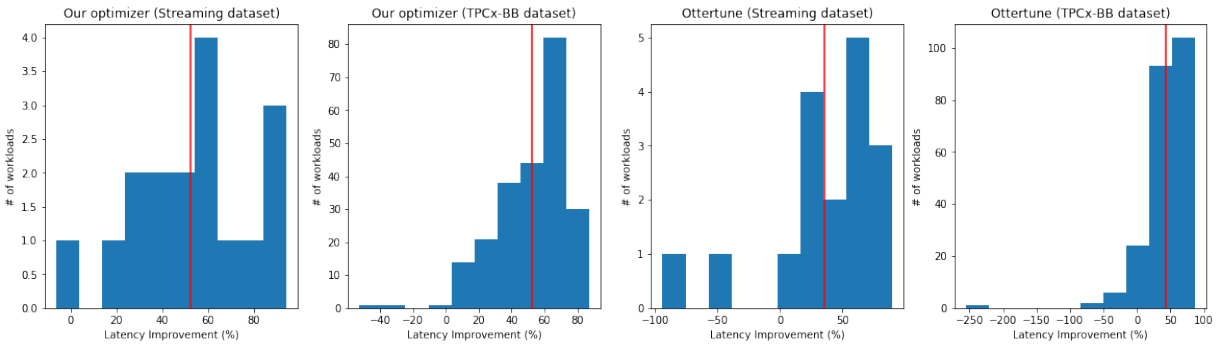


Figure 18: End to end performances and comparison to Ottertune with resource upgrade option.

52.44% on TPCx-BB workloads, compared to 35.96% and 43.19% for Ottertune, respectively.

After closely examining the configurations recommended by our method and Ottertune, we noticed that both methods aggresively increase the amount of resources allocated in most of the test workloads. Increasing the amount of resources allocated for a workload (such as the total number of cores and the memory per executor) yields in general better runtime latencies regardless of the choice of the remaining knobs. This explains why the gap is not very big between both methods when it comes to end-to-end performances. However, in some of our test workloads, where initial configurations are already assigned the biggest resource capacity, and where both optimization methods keep the resource allocation knobs intact but change other knobs, our method tends to recommend better configurations than Ottertune. Hence this gave us the motivation to conduct a second end-to-end experiments in which we fix the choice of the resources allocated in the initial configuration.

Expt2: Fixing resource allocation knobs. We repeat the end-to-end experiments this time but without allowing neither our optimizer nor Ottertune to upgrade the resources (so we have frozen the values of the resource knobs v_7, v_8 and v_9). The main results are shown within Figure 19. Our optimizer achieves, on average a latency improvement of 30.68% on streaming jobs and an improvement of 7.48% on TPCx-bb jobs without changing the resource knobs. This is an interesting finding because it means that with a fixed cloud computation budget, our optimizer can still recommend better configurations in terms of runtime latency. On the other side, Ottertune recommends configurations that are 41.01% worse than the initial configuration on the streaming trace (negative average latency improvement) and fails to provide noticeable improvements on TPCx-BB workloads after scoring only 0.13% of improvement on average.

7.7 Summary

This chapter has covered experiments for modeling architectures introduced in previous chapters as well as an and end-to-end comparison to a state of the art tuning tool, Ottertune [61], using our own trace datasets.

Experiments with recommender modeling architectures with implicit encoding extraction revealed that a dot product based modeling (such as the one in matrix factorization approaches) works better than modeling the runtime latency using a neural network based recommender approach if we guarantee to sample 5 shared configurations for a newly admitted workload.

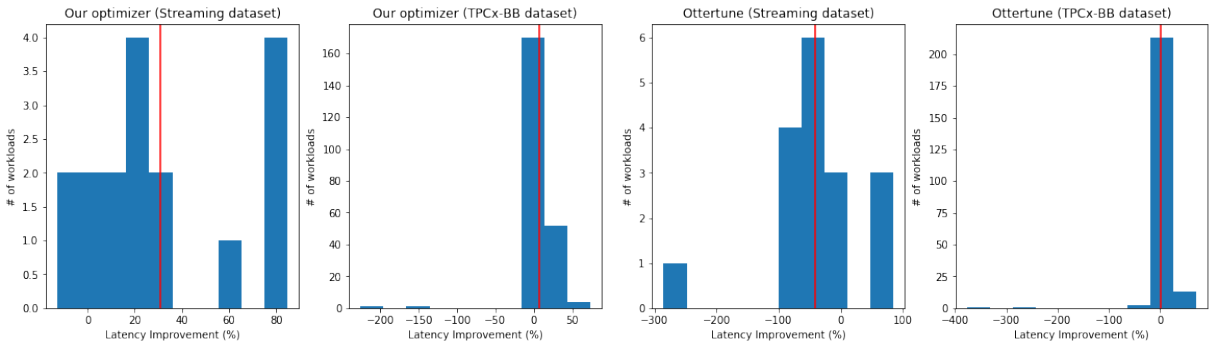


Figure 19: End to end performances and comparison to Ottertune under fixed resources.

We explained that these methods have the limitation of not supporting a zero-shot learning scheme and thus require an incremental training step prior to prediction. They also suffer from not being able to support prediction over arbitrary combinations of configurations.

We then outlined results from the other modeling methods that explicitly extract encodings from runtime metrics before training a neural network regressor. We found that the siamese neural networks which satisfy a relaxation of the invariance property are the best at modeling the runtime latency when jobs are admitted in the most practical case with a single arbitrary configuration. This architecture provided modeling errors 4-5x lower than state of the art tuning tool Ottertune and 3x lower than bypassing the representation learning step under a single arbitrary configuration admission scheme. The ablation and scalability studies revealed that the triplet loss from the siamese networks family is 1-2x slower in training than its SNN counterpart, but is more robust to fewer training points per workload.

The better modeling results with the siamese network translated into better end-to-end improvements than a state of the art tuning tool, Ottertune. For instance, our optimizer achieves latency improvements around 52.4% on both benchmarks while Ottertune achieves latency improvements of 35.96% on the streaming benchmark and 43.19% on the TPCx-bb benchmark when jobs are admitted with an arbitrary configuration and our optimizer is allowed to scale out resources. In addition, in the setting of limited resources, our optimizer achieves a latency improvement of 30.68% and 7.48% on the streaming and TPCx-bb benchmarks respectively while Ottertune fails to provide better configurations. For instance, the configurations recommended by Ottertune yield latencies that are 41.01% worse on average than the initial configurations (negative average latency improvement) on the streaming benchmark, and yield a latency improvement of only 0.13% on the TPCx-bb benchmark.

8 Extended Discussions

In this section, we present an extended discussion of some of the technical issues.

Lack of open source trace datasets that support replication. In order to assess the performance of the unified data analytics optimizer whose modeling component is detailed in this thesis, we have ensured that it's possible to conduct end to end comparative results with state of the art tuning tools and explain in which scenarios our optimizer would work better. However, the database community's lack of open source datasets of workload traces has made it harder to do such a comparison. Even if there were some open source datasets, it could have been the case that the optimizer recommended a configuration that is not within the existing collected traces, and thus no comparative results could be reported because it is hard to replicate the same computing environment from which open source traces were sampled in order to sample additional configurations. Hence, this required us to deploy the benchmarks on our distributed computing environment and this is why we collected ourselves traces from different workloads within these benchmarks. These traces have been used to train different machine learning based predictive models which were an essential component probed by the optimizer prior to making a recommendation. Then, the recommended configurations were run on the same computing environment from which the first initial traces were collected. A substantial amount of engineering work has been employed in order to run the traces, collect their outputs and then rerun them in case of failures. We made the traces available at `https://github.com/udao-modeling/code`.

That being said, we must clarify here that the comparison that we've done with respect to Ottertune [61] uses our own dataset of traces and not Ottertune's trace datasets which have not been made available yet by the authors of the paper at the moment of writing this manuscript.

Difficulty of training under 1 arbitrary configuration and the need to support this admission scheme. In the previous chapters, we've explained that admitting a job profiled with a single arbitrary configuration is the most challenging use case to handle because less information is available to modeling. Most representation learning methods we've surveyed have failed to reduce the gap on test errors between this challenging admission scheme and other remaining schemes (admitting a job with 5 arbitrary configurations or with 1 or 5 shared configurations). These high test errors found with other representation learning techniques are in fact due to underfitted regressors when encodings are not extracted from a siamese neural network based architecture.

Yet, we need to support this admission scheme within our tuning system for Spark workloads. This is because we can't force the cloud user of not choosing his first initial configuration with which he'd like to run his workload. We should also take into consideration that some jobs can't be run with a particular configuration. To give an example, consider a streaming workload parametrized from Template A as in §6.1.1. Spark requires the value of the batch interval knob (knob v_1) to be a divider for both the window size and the slide interval. In other words, the window size and slide interval have to be a always a multiple of batch interval as per the requirements of DStreams [67] implemented within Spark. The slide interval in this

template is set to 10, and if we consider the value of window size equal to 30 seconds ($w = 30$), then v_1 can't take values outside $\{1, 2, 5, 10\}$. Suppose that we've added this workload to our system which had previously some shared configurations where v_1 had values in $\{3, 6, 9, ...\}$, there is no way for us to run this workload with the same shared configuration. Hence the need for admitting a workload under an arbitrary configuration.

Hyperparam tuning of ML models. So far we have been interested in tuning knobs of Spark, but we haven't discussed that our tuning system itself requires tuning. Indeed, the machine learning methods used to model runtime latency of Spark workloads have been tuned before they could give the performances we reported. In this thesis, we have resorted, to a random search for tuning hyper-parameters of the ML architectures we introduced. While we have presented AutoML tools in Chapter 2 (such as Spearmint [55], Hyperband [38] and BOHB [23]) as systems that can be used to tune Spark workloads (and we explained that in our use case these systems require actually setting the configuration several times before they yield good predictions), we could have used these tools in order to tune the hyper-parameters of the ML architectures we have proposed. The tuning problem in this case appears to be a recursive problem (any tuning tool itself needs to be tuned). We recall that it's fine to tune our machine learning architectures overnight, but the problem of tuning spark workloads that we are addressing in this thesis needs to be on the fly in a zero-shot learning scheme.

Qualitative Comparison to Ottertune Our proposed modeling pipeline introduced in Figure 7 of §5.2 focused particularly on the representation learning step that learns encodings from the traces and where each component can be considered as a combination of several metrics, in sharp contrast to Ottertune which resorts to reducing the dimension of runtime metrics by selecting a subset of metrics as representative metrics. This representation learning step is a core component of our tuning system and is responsible of characterizing different workloads according to the properties we outlined in the beginning of the thesis in §3.4: reconstruction, independence, invariance, and similarity. Learning encodings of different workloads allowed us to train a single global regression model that allows predicting the runtime latency accurately for test workloads in a zero-shot scheme. Such global models bypass a workload mapping step which remains required within Ottertune's modeling pipeline because it doesn't represent a workload by a numerical vector. Our experiments revealed a particular interest in focusing on the similarity property while characterizing the workloads and the superior performance of our best techniques with respect to Ottertune in the most practical and yet the most challenging job admission scenario that consists of observing a unique arbitrary configuration. This job admission scenario poses major concerns for Ottertune as soon as a new workload is admitted with a configuration that is incompatible with existing training workloads as explained in the previous paragraph. While Ottertune fails to map the workload to a past workload in such a scenario, our method will still be able to handle such a particular case.

9 Conclusions

In this chapter, we summarize the contributions we made in this thesis, discuss some of the challenges encountered, and state some research directions for future work.

9.1 Contributions of this thesis

In this thesis, we presented our solution to performance modeling for cloud data analytics, including (*i*) a system design that suits the constraints in real world applications, (*ii*) a notion of learning workload embeddings with desired properties for different jobs, thereby enabling performance prediction when used together with job configurations; (*iii*) an in-depth study of different modeling choices that meet our requirements. Results of extensive experiments show the strengths and limitations of different modeling methods, reveal the best performing technique to be the one that can best approximate the invariance property of workload embeddings, and demonstrate our superior performance over a state-of-the-art modeling technique for cloud analytics.

9.1.1 Recommender systems architectures

We have surveyed diverse methods used within recommender systems and explained how matrix factorization approaches in particular have been previously used in Paragon [20] and Quasar [21] for solving the configuration recommendation problem for cloud workloads. We have provided an extension of the matrix factorization approach and proposed two neural network based recommender approaches. We explained how the hybrid embedding approach, in particular, allows to make recommendations beyond already observed configurations and hence generalize by recommending new configurations. We outlined that recommender architectures have an implicit encoding learning step since they couple both the workload encoding learning step as well as the regression step within the same architecture. We have explained that these methods have the limitation of not supporting a zero-shot learning scheme and thus require an incremental training step prior to prediction. They also suffer from not being able to support prediction over arbitrary combinations of configurations.

9.1.2 Representation learning based architectures

We have introduced representation learning methods as a solution to the limitations of existing recommender architectures. We explained that representation learning based methods incorporate more content by leveraging the full runtime metrics while extracting the encodings. They explicitly extract encodings from runtime traces before feeding them to a neural network dedicated for the end regression task on the runtime latency. We have covered different auto-encoders starting with deterministic autoencoders and explained why the basic auto-encoder violates both the independence and invariance properties. We have then added edits on top of the architecture and augmented the loss function with supervised and unsupervised terms as an attempt to favor these properties while training the auto-encoders which natively satisfy the reconstruction property. We also briefly covered the variational auto-encoder from the generative family of auto-encoders because it's known for its capacity

of automatic disentanglement of generative factors within the bottleneck layer. Furthermore, we have also introduced another paradigm of extracting encodings that relies on siamese neural networks instead of auto-encoders, and explained how to train such networks that satisfy a relaxation of the invariance property. Finally, we also gave examples of designs of hybrid architectures that combine auto-encoders and siamese neural networks.

9.1.3 Comparative results for modeling

	Embedding	**Autoencoder**	**Siamese Neural Network**
Independence	✓	Depends	x
Invariance	✓	x	x
Similarity	-	x	✓
Reconstruction	Partially	✓	x
Zero-shot learning	x	✓	✓

Table 8: Properties of the different families of approaches.

Tradeoffs between different properties across modeling techniques. While presenting the different modeling approaches we have listed the workload encodings properties satisfied by each of these methods. We summarize in the Table 8 the different families of techniques as well as the different properties satisfied by each of these techniques. Both embedding architectures (Dual embeddings and Hybrid embeddings) satisfy both the independence and invariance properties by design. These architectures learn embeddings while minimizing the mean squared error loss between the approximation of the runtime latency and its ground truth value. Hence they also partially satisfy the reconstruction property. The auto-encoder based architectures however do not satisfy by default the independence property if we don't break the bottleneck layer for example as we did with the custom auto-encoder, or if we don't start with prior assumptions as this is the case with variational auto-encoders. By default, all of the auto-encoder architectures satisfy the reconstruction property but are likely to violate the invariance property when it comes to learning encodings. It is because these two properties can't be achieved at the same time due to the architectural design of the auto-encoder. On the other hand, when we introduced siamese networks, we explained that these networks focus on a relaxation of the invariance property, which we denoted by similarity. For both the auto-encoder based architectures and siamese neural networks, we have forced an invariant encoding by adding an aggregator before training the neural network regressor (see Figure 7).

Experiments comparing recommender architectures. Experiments with recommender modeling architectures revealed that a dot product based modeling (such as the one in matrix factorization approaches) works better than modeling the runtime latency using a neural network based recommender approach if we guarantee to sample 5 shared configurations for a newly admitted workload.

Experiments comparing modeling architectures Throughout experiments covering representation learning techniques, we've observed that (1) the invariance property subsumes the independence property, and (2) focusing on a relaxation of the invariance property while leveraging all the runtime metrics within explicit based encoding extraction techniques worked much better than implicit encoding extraction, which, although learns an invariant encoding, does not leverage the full runtime trace vector during training and inference. We found that the siamese neural networks which satisfy a relaxation of the invariance property are the best at modeling the runtime latency when jobs are admitted in the most practical case with a single arbitrary configuration. This architecture provided modeling errors around 10% on both benchmarks of runtime traces. These modeling errors are 4-5x lower than a state of the art tuning tool, Ottertune [61], and 3x lower than bypassing the representation learning step under a single arbitrary configuration admission scheme. Hybrid architectures that combined both the auto-encoder and the siamese neural networks didn't provide better modeling errors than pure siamese networks on the regression task on the runtime latency.

The ablation and scalability studies revealed that the triplet loss from the siamese networks family is 1-2x slower in training than its SNN counterpart, but is more robust to fewer training points per workload.

9.1.4 End-to-end comparison to Ottertune

The better modeling results with the siamese network translated into better end-to-end improvements than a state of the art tuning tool, Ottertune [61]. For instance, our optimizer achieves latency improvements around 52.4% on both benchmarks while Ottertune achieves latency improvements of 35.96% on the streaming benchmark and 43.19% on the TPCx-bb benchmark when jobs are admitted with an arbitrary configuration and our optimizer is allowed to scale out resources. In addition, in the setting of limited resources, our optimizer achieves a latency improvement of 30.68% and 7.48% on the streaming and TPCx-bb benchmarks respectively while Ottertune fails to provide better configurations. For instance, the configurations recommended by Ottertune yield latencies that are 41.01% worse on average than the initial configurations (negative average latency improvement) on the streaming benchmark, and yield a latency improvement of only 0.13% on the TPCx-bb benchmark.

9.2 Future directions

As part of our future work, we propose some extensions to recommender based approaches that leverage the kernel trick to replace the dot product, then we provide an extension to the hybrid embedding approach that leverages additional meta-information available upon the submission of the workload, and finally we explain how to extend the current modeling approach to predict time series of the target objective instead of mean performance modeling. We detail below the different extensions we propose.

Kernel Matrix Factorization The collaborative filtering approach proposed so far can be generalized to go beyond simple matrix factorization. Indeed, instead of representing the interaction between workload j and configuration i as the dot product between $\mathbf{z}_j$ and $\mathbf{v}^i$ in the euclidean space ($\mathbf{z}_j^T \mathbf{v}^i$), we can think of the kernel trick in order to substitute the

simple dot product by a dot product of embeddings in a higher dimensional space. If ϕ is some implicit feature mapping defined as: $\phi : \mathcal{X} \rightarrow \mathcal{H}$ where $\mathcal{X}$ is the original space and $\mathcal{H}$ is the Hilbert feature space, then the embedding of $\mathbf{z}_j$ in $\mathcal{H}$ can be written as $\phi(\mathbf{z}_j)$ and the embedding of $\mathbf{v}^i$ in $\mathcal{H}$ can be written as $\phi(\mathbf{v}^i)$. The interaction between the configuration and the workload can be then modeled by $k(\mathbf{z}_j, \mathbf{v}^i) = \phi(\mathbf{z}_j)^T \phi(\mathbf{v}^i)$ instead of a simple dot product. The paper [40] gives a closed form solution for the kernel matrix factorization problem but requires solving $n + m$ separate optimization problems. An interesting direction to explore is understanding which types of kernels would model well the interaction between the workloads and the configurations, and hence we leave it as part of a future work.

Leveraging workload meta-data within hybrid recommender approach. Although our optimizer tunes spark workloads in a blackbox manner, it can leverage the meta-data that is submitted alongside the workload for extracting some additional workload embeddings so that the workload at the end can be described by two kind of embeddings: (1) embeddings extracted from runtime trace vectors (as we did in this thesis) (2) embeddings extracted from the *submit* command arguments. Take the following *spark-submit* command as an example.

```
spark-submit --class org.apache.spark.streamdm.streamDMJob \
--master yarn --deploy-mode client \
--conf spark.executor.cores=3 \
--conf spark.default.parallelism=6 \
--conf spark.streaming.blockInterval=50 \
--conf spark.executor.instances=8 \
--conf spark.reducer.maxSizeInFlight=170m \
--conf spark.executor.memory=4915m \
--conf spark.shuffle.sort.bypassMergeThreshold=1 \
--conf spark.shuffle.compress=false \
--conf spark.metrics.conf=/home/repo/hsc1/metrics.properties \
~/trace-generation-system/jars/spark-benchmark-streamdm.debug.jar \
inputRate=1100000 \
batchInterval=1 \
jobId=67 \
receiverPort2=10453 \
receiverPort3=10454 \
receiverPort1=10452 \
receiverPort6=10457 \
receiverPort4=10455 \
receiverPort5=10456 \
receiveFromHostname3=node1 \
sdmParams="EvaluatePrequential -s (SocketTextStreamReader) -l (SGDLearner -l 0.1 \
-o HingeLoss -r L2Regularizer -p 0.01 -f 3)" \
NumReceivers=6 \
receiveFromHostname4=node1 \
outputPath=hdfs://10.0.0.1:8020/user/hsc1/benchmark/data \
sparkLogDir=/mnt/disk5/khaled/hsc1/logs/spark \
receiveFromHostname2=node1 \
receiveFromHostname1=node1 \
receiveFromHostname6=node1 \
receiveFromHostname5=node1 \
checkpointDir=hdfs://10.0.0.1:8020/user/hsc1/benchmark/checkpoint \
signature=hsc1_1585646953
```

Additional information regarding the name of the class that is being executed by the workload, as well as other non configuration related arguments (such as the name of the jar file, *sparkLogDir*, *outputPath*, *NumReceivers*, *sdmParams*) could be used to extract a text-based embedding for the workload, and hence use it alongside other embeddings extracted from the runtime trace in order to predict the runtime latency. We leave this as a future direction to explore.

Relational Embeddings for incorporating more data related content The introduced approaches within this thesis have focused on learning workload specific embeddings which assumed that the inherent data characteristics are part of the workload itself. If we consider the subset of analytics that operate on top of structured datasets, such as relational queries which operate on well defined relational schemas, then we can isolate learning the data characteristics from the workload defined on top of these data. For example, the recent work in [17] learns embebddings from relational datasets for a data integration task. We can use the underlying ideas to learn data embeddings for relational workloads and use these embeddings alongside other workload embeddings we introduced for the end regression task.

Time series performance tuning instead of average performance tuning. We've been so far interested in predicting on average the performances of Spark workloads during a short period of time for streaming workloads. The same modeling techniques that we covered in this thesis can be coupled with sequential or recurrent neural networks (such as LSTMs [29] or Transformers [62]) in order to train a time series that can predict instantaneous performances instead of average performances. This can be interesting in applications where the workload behaviors are constantly changing throughout the execution of the workload.

References

[1] Apache hadoop yarn documentation. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[2] Sparklens. https://github.com/qubole/sparklens.

[3] Tune hadoop and spark performance with dr. elephant and sparklens on amazon emr. https://aws.amazon.com/blogs/big-data/tune-hadoop-and-spark-performance-with-dr-elephant-and-sparklens-on-amazon-emr/.

[4] Tuning spark. https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism.

[5] Spark 3.0.0. Monitoring documentation. https://spark.apache.org/docs/3.0.0/monitoring.html.

[6] Dropwizard 3.1.0. Metrics documentation. https://metrics.dropwizard.io/3.1.0/getting-started/.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[9] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015.

[10] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier

Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[11] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 469–482, USA, 2017. USENIX Association.

[12] MLLib Apache Spark. Streaming k-means. `https://spark.apache.org/docs/2.3.1/mllib-clustering.html#streaming-k-means`.

[13] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In Georg Lausen and Dan Suciu, editors, *Database Programming Languages*, pages 1–19, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[14] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, 2012.

[15] A. Bifet, S. Maniu, J. Qian, G. Tian, C. He, and W. Fan. Streamdm: Advanced data mining in spark streaming. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 1608–1611, 2015.

[16] S. Blackburn, Robin Garner, C. Hoffmann, A. Khan, K. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, A. Hosking, M. Jump, H. Lee, J. B. Moss, Aashish Phansalkar, D. Stefanovic, Thomas VanDrunen,

D. V. Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06*, 2006.

[17] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1335–1349, New York, NY, USA, 2020. Association for Computing Machinery.

[18] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[19] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Christina Delimitrou and Christos Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.*, 31(4), December 2013.

[21] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[22] Sergey Dudoladov, Chen Xu, Sebastian Schelter, Asterios Katsifodimos, Stephan Ewen, Kostas Tzoumas, and Volker Markl. Optimistic recovery for iterative dataflows in action. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1439–1443, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[24] Nicholas Frosst, Nicolas Papernot, and Geoffrey Hinton. Analyzing and improving representations with the soft nearest neighbor loss. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2012–2020, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[25] Michael A. Gelbart, Jasper Snoek, and Ryan P. Adams. Bayesian optimization with unknown constraints. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, UAI'14, page 250–259, Arlington, Virginia, USA, 2014. AUAI Press.

[26] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[27] Irina Higgins, Loïc Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[28] Alexander Hinneburg, Charu C. Aggarwal, and Daniel A. Keim. What is the nearest neighbor in high dimensional spaces? In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 506–515, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[30] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[31] Kevin G. Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. *CoRR*, abs/1502.07943, 2015.

[32] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. *CoRR*, abs/1409.0792, 2014.

[33] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[34] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *CoRR*, abs/1906.02691, 2019.

[35] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[36] Lawrence Berkeley National Laboratory (LBNL). Dataset of requests to 1998 worldcup website. `ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

[37] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting scalable analytics with latency constraints. *Proc. VLDB Endow.*, 8(11):1166–1177, July 2015.

[38] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[39] Linkedin. Dr. elephant. `https://github.com/linkedin/dr-elephant`.

[40] Xinyue Liu, Chara Aggarwal, Yu-Feng Li, Xiaugnan Kong, Xinyuan Sun, and Saket Sathe. Kernelized matrix factorization for collaborative filtering. pages 378–386, 06 2016.

[41] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[42] Ryan Marcus and Olga Papaemmanouil. Wisedb: A learning-based workload management advisor for cloud databases. *Proc. VLDB Endow.*, 9(10):780–791, June 2016.

[43] Ryan Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, July 2019.

[44] Ryan Marcus, Sofiya Semenova, and Olga Papaemmanouil. A learning-based service for cost and performance management of cloud databases. *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1361–1362, 2017.

[45] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[47] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[48] pyNmonAnalyzer. `https://github.com/madmaze/pyNmonAnalyzer`,.

[49] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 415–427, New York, NY, USA, 2016. Association for Computing Machinery.

[50] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 833–840, Madison, WI, USA, 2011. Omnipress.

[51] Ruslan Salakhutdinov and Geoff Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In Marina Meila and Xiaotong Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2

of *Proceedings of Machine Learning Research*, pages 412–419, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR.

[52] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Incremental singular value decomposition algorithms for highly scalable recommender systems. *Fifth International Conference on Computer and Information Science*, 01 2002.

[53] Bernhard Schölkopf, Alexander J. Smola, and Klaus-Robert Müller. *Kernel Principal Component Analysis*, page 327–352. MIT Press, Cambridge, MA, USA, 1999.

[54] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 815–823, 2015.

[55] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, page 2951–2959, Red Hook, NY, USA, 2012. Curran Associates Inc.

[56] Jasper Snoek, Kevin Swersky, Richard Zemel, and Ryan P. Adams. Input warping for bayesian optimization of non-stationary functions. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–1674–II–1682. JMLR.org, 2014.

[57] Spark. Level of parallelism in data receiving. `https://spark.apache.org/docs/latest/streaming-programming-guide.html#level-of-parallelism-in-data-receiving`.

[58] Kevin Swersky, Jasper Snoek, and Ryan P. Adams. Multi-task bayesian optimization. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2004–2012, Red Hook, NY, USA, 2013. Curran Associates Inc.

[59] TPCx-BB. Tpcx-bb (bigbench) benchmark for big data analytics. `http://www.tpc.org/tpcx-bb/`.

[60] Michael Tschannen, Olivier Bachem, and Mario Lucic. Recent advances in autoencoder-based representation learning. *CoRR*, abs/1812.05069, 2018.

[61] 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 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[63] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[64] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 363–378, USA, 2016. USENIX Association.

[65] Kilian Q. Weinberger and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.*, 10:207–244, June 2009.

[66] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[67] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[68] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. A demonstration of the ottertune automatic database management system tuning service. *Proc. VLDB Endow.*, 11(12):1910–1913, August 2018.

[69] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[70] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, October 2012.

A Equivalence between the matrix factorization formulation and iterative updates within Paragon

In this section, we show that minimizing the loss function we previously introduced in equation 5 in its matrix form is equivalent to the iterative update formulas provided in scalar form within Paragon [20]. We recall that the loss function we earlier introduced can be written as:

$$(\hat{Z}, \hat{V}) = \underset{Z,V}{\operatorname{argmin}} \frac{1}{2}||1_K \circ (Y - ZV)||_F^2 + \frac{\lambda}{2}||Z||_F^2 + \frac{\lambda}{2}||V||_F^2$$

Let's rewrite the same loss function in scalar form this time:

$$\mathcal{L} = \sum_{j,i} L_{ji}$$

where

$$L_{ji} = \frac{1}{2}\epsilon_{ji}^2 + \frac{\lambda}{2}||\mathbf{z}_j||^2 + \frac{\lambda}{2}||\mathbf{v}^i||^2$$

with ϵ_{ji} is given by:

$$\epsilon_{ji} = y_{ji} - \mathbf{z}_j^T \mathbf{v}^i$$

If we compute the gradient of L_{ji} with respect to both vectors $\mathbf{z}_j$ and $\mathbf{v}^i$, we get:

$$\nabla_{\mathbf{z}_j} L_{ji} = -\epsilon_{ji}\mathbf{v}^i + \lambda \mathbf{z}_j$$

$$\nabla_{\mathbf{v}^i} L_{ji} = -\epsilon_{ji}\mathbf{z}_j + \lambda \mathbf{v}^i$$

Thus, the update on $\mathbf{z}_j$ can be written as:

$$\mathbf{z}_j \leftarrow \mathbf{z}_j - \eta \nabla_{\mathbf{z}_j} L_{ji}$$

$$\mathbf{z}_j \leftarrow \mathbf{z}_j + \eta(\epsilon_{ji}\mathbf{v}^i - \lambda \mathbf{z}_j)$$

And the update on $\mathbf{v}^i$ can be written as:

$$\mathbf{v}^i \leftarrow \mathbf{v}^i - \eta \nabla_{\mathbf{v}^i} L_{ji}$$

$$\mathbf{v}^i \leftarrow \mathbf{v}^i + \eta(\epsilon_{ji}\mathbf{z}_j - \lambda \mathbf{v}^i)$$

These updates are equivalent to the updates provided within Paragon [20] on page 3 (but by taking into consideration of course the different notations).

B Workloads dataset details

Details regarding parametrization of the streaming workloads are provided withinn Tables 9, 10 and 11. Test jobs have their rows prefixed with a *.

	job id	w	π	δ		job id	w	π	δ
*	*A.1*	10	35	0	*	*D.2*	10	35	500
	A.2	10	35	300		*D.3*	10	35	10000
	A.3	10	35	1000		*D.4*	10	70	0
	A.4	10	70	0		*D.5*	10	70	500
	A.5	10	70	300		*D.6*	10	70	10000
*	*A.6*	10	70	1000	*	*D.7*	30	35	0
	A.7	30	35	0		*D.8*	30	35	500
	A.8	30	35	300		*D.9*	30	35	10000
	A.9	30	35	1000		*D.10*	30	70	0
	A.10	30	70	0		*D.11*	30	70	500
*	*A.11*	30	70	300	*	*D.12*	30	70	10000
	A.12	30	70	1000		*E.1*	10	35	0
*	*B.1*	10	35	N/A		*E.2*	10	35	10^7
	B.2	10	70	N/A	*	*E.3*	10	35	2.10^7
	B.3	30	35	N/A		*E.4*	10	70	0
	B.4	30	70	N/A		*E.5*	10	70	10^7
	C.1	N/A	35	0		*E.6*	10	70	2.10^7
	C.2	N/A	35	1800	*	*E.7*	30	35	0
	C.3	N/A	35	80000		*E.8*	30	35	10^7
	C.4	N/A	70	0		*E.9*	30	35	2.10^7
*	*C.5*	N/A	70	1800		*E.10*	30	70	0
	C.6	N/A	70	80000		*E.11*	30	70	10^7
	D.1	10	35	0	*	*E.12*	30	70	2.10^7

Table 9: Parametrizations of SQL-like jobs.

	job id	learning rate	Loss	Regularizer	Regularization Coefficient
*	F.1	0.01	Logistic	-	-
	F.2	0.01	Hinge	L1	1
	F.3	0.01	Logistic	L2	1
	F.4	0.01	Hinge	-	-
*	F.5	0.01	Logistic	L1	1
	F.6	0.01	Hinge	L2	1
	F.7	0.1	Logistic	-	-
	F.8	0.1	Hinge	L1	0.01
	F.9	0.1	Logistic	L2	0.01
	F.10	0.1	Hinge	-	-
	F.11	0.1	Logistic	L1	0.01
*	F.12	0.1	Hinge	L2	0.01

Table 10: Parametrizations of ML classification jobs (template F).

	job id	K	Input Dataset
	G.1	2	3D
	G.2	4	3D
*	G.3	8	3D
	G.4	10	3D
	G.5	20	3D
	G.6	100	4D
	G.7	2	4D
	G.8	4	4D
	G.9	8	4D
*	G.10	10	4D
	G.11	20	4D
*	G.12	100	4D

Table 11: Parametrizations of ML clustering jobs (template G).

C Details on the variational auto-encoder

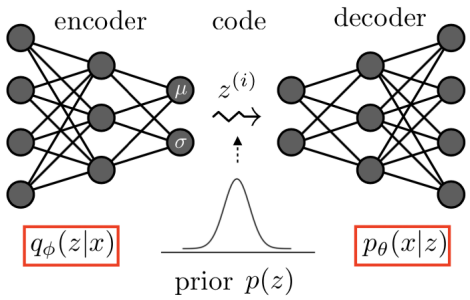


Figure 20: Variational auto-encoder diagram [60].

The loss of the variational autoencoder is given by:

$$L(\beta) = -\mathbb{E}_{z\sim q_\phi(z|x)} log p_\theta(x|z) + \beta D_{KL}(q_\phi(z|x)||p(z))$$

where $p(z)$ represents the prior distribution on the encodings (usually the prior $p(z) = \mathcal{N}(0, I)$), $q_\phi(z|x)$ represents the family (parametrized by ϕ) of the approximate posterior distribution and $p_\theta(x|z)$ represents the likelihood of x given z when z is sampled from $q_\phi(z|x)$.

A trick that is commonly used in order to get an easy formulation of the problem consists of choosing a normal distribution for the posterior distribution of the encoding z as well. The encoder part of the auto-encoder then predicts the parameters (mean and co-variance matrix) of the distribution that corresponds to a particular input point. The co-variance matrix of this distribution is assumed to be always a diagonal matrix no matter what input point is fed to the architecture. This leads to simplifications and ensures ensures getting an easy closed form solution of the KL-divergence term.

For an encoding of size s_z, the output of the encoder layer should be $2s_z$. The first s_z dimensions are reserved for the mean of the distribution, and the second s_z dimensions correspond to the diagonal of the covariance matrix.

D Hyper-parameter tuning of different models

For the encoder/decoder based architectures as well as the neural networks we tuned: (1) *topology hyper-parameters*: (number of layers, number of hidden units per layer, activations) (2) *optimization hyper-parameters* (learning rate, number of epochs, patience (for early stopping) (3) *loss related hyper-parameters* (regularization coefficients balacing different terms within a loss function, temperature, etc...)

We tuned the hyper-parameters by random sampling from a pool of hyper-parameters. We use a 5-fold cross validation scheme for tuning our workloads. We try to simulate the same conditions on the test set when we do cross validation, thus we consider observing only few (1 or 5 traces) for training workloads within the left out fold during the cross validation procedure.

It may appear that we are casting tuning Spark workloads to tuning hyper-parameters of machine learning models. It is important to emphasize though that the machine learning solution we are proposing to modeling performances of spark workloads can be tuned overnight (and not at the time of the execution of the Spark workload). Having a robust global model that allows us to predict performances of a new submitted Spark job from a unique (or few) trace(s) in a zero-shot scheme makes the tuning of Spark workload non-invasive to the user, and much faster.

E Extensions of the contraction term with more layers and activations

The original paper describing the contractive auto-encoder provides the calculations for the Jacobian term under a single layer with a *sigmoid* activation. We provide extensions while implementing the contractive auto-encoder and calculate the Jacobian in addition under:

- 2 layers of sigmoid non linearity
- 1 layer of ReLu non linearity
- 2 layers of ReLu non linearity

In order not to abuse the notation while providing the extensions, let's apply the contractive loss on a simple auto-encoder instead of the custom auto-encoder as in §5.3. The extension to our custom auto-encoder is straightforward.

The original loss function introduced within the contractive autoencoders paper [50] can be written in our notation as:

$$\mathcal{J} = \sum_{\mathbf{x} \in D_n} \left(L(\mathbf{x}, d(e(\mathbf{x})) + \lambda ||J_e(\mathbf{x})||_F^2 \right)$$

where e is the encoding function, g is the decoding function and L is the MSE in our case and D_n is the set of all training points.

$$||J_e(\mathbf{x})||_F^2 = \sum_{i,j} \left(\frac{\partial z_i}{\partial x_j} \right)^2$$

z_i is the i-th component of the encoding vector $\mathbf{z}$ and x_j is the j-th component of the input vector $\mathbf{x}$.

E.1 1 layer of sigmoid activation

$$\mathbf{z} = f(\mathbf{x}) = sigmoid(W\mathbf{x} + \mathbf{b})$$

with:
- $\mathbf{x} = [x_1, x_2, ..., x_{dx}]^T$
- $\mathbf{z} = [z_1, z_2, ..., z_{dz}]^T$
- $\mathbf{b} = [b_1, b_2, ..., b_{dz}]^T$ is the intercept vector of the encoder layer of the neural network.
- W, of shape $[d_z, d_x]$, is the weights matrix of the first layer of the neural network.

$$\boxed{||J_e(\mathbf{x})||_F^2 = \sum_{i=1}^{d_z} z_i^2(1 - z_i)^2 \sum_{j=1}^{d_x} w_{ij}^2}$$

E.2 2 layers of sigmoid activations

$$\mathbf{h} = h(\mathbf{x}) = sigmoid(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{z} = e(\mathbf{x}) = e(h(\mathbf{x})) = sigmoid(W^{(2)}\mathbf{h} + \mathbf{b}^{(2)})$$

with:
- $\mathbf{x} = [x_1, x_2, ..., x_{dx}]^T$
- $\mathbf{h} = [x_1, x_2, ..., x_{dh}]^T$
- $\mathbf{z} = [z_1, z_2, ..., z_{dz}]^T$
- $\mathbf{b}^{(1)} = [b_1^{(1)}, b_2^{(1)}, ..., b_{dh}^{(1)}]^T$
- $\mathbf{b}^{(2)} = [b_1^{(2)}, b_2^{(2)}, ..., b_{dz}^{(2)}]^T$
- d_z: number of hidden dimensions of the encoding layer
- d_h: number of hidden dimensions of the first layer
- d_x: number of input dimensions
- $W^{(1)}$ is of shape $[d_h, d_x]$
- $W^{(2)}$ is of shape $[d_z, d_h]$

$\frac{\partial z_i}{\partial x_j} = ?$

Applying the chain rule, we get: $\frac{\partial z_i}{\partial x_j} = \sum_k \frac{\partial z_i}{\partial h_k} \frac{\partial h_k}{\partial x_j}$

$$\frac{\partial z_i}{\partial h_k} = z_i(1 - z_i)w_{ik}^{(2)}$$

$$\frac{\partial h_k}{\partial x_j} = h_k(1 - h_k)w_{kj}^{(1)}$$

$$\frac{\partial z_i}{\partial x_j} = z_i(1 - z_i)\sum_k h_k(1 - h_k)w_{kj}^{(1)}w_{ik}^{(2)}$$

$$||J_e(\mathbf{x})||_F^2 = \sum_{i,j}\left(\frac{\partial z_i}{\partial x_j}\right)^2$$

$$\boxed{||J_e(\mathbf{x})||_F^2 = \sum_{i=1}^{dz} z_i^2(1 - z_i)^2 \sum_{j=1}^{dx}\left(\sum_{k=1}^{dh} h_k(1 - h_k)w_{kj}^{(1)}w_{ik}^{(2)}\right)^2}$$

E.3 1 layer of ReLu activation

Using the same notation as in the case of 1 layer of *sigmoid* activation, we can write:

$$\mathbf{z} = e(\mathbf{x}) = relu(W\mathbf{x} + \mathbf{b})$$

If we denote by $\tilde{\mathbf{z}}$ the pre-activation of $\mathbf{z}$, then we can write:

$$\mathbf{z} = relu(\tilde{\mathbf{z}}) = relu(W\mathbf{x} + \mathbf{b})$$

$$\frac{\partial z_i}{\partial x_j} = w_{ij}\mathbb{1}_{\tilde{z}_i > 0}$$

$$\boxed{||J_e(\mathbf{x})||_F^2 = \sum_{i=1}^{dz}\mathbb{1}_{\tilde{z}_i > 0}\sum_{j=1}^{dx} w_{ij}^2}$$

E.4 2 layers of ReLu activations

$$\mathbf{h} = h(\mathbf{x}) = relu(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\mathbf{z} = e(\mathbf{x}) = e(h(\mathbf{x})) = relu(W^{(2)}\mathbf{h} + \mathbf{b}^{(2)})$$

Let's denote by $\tilde{\mathbf{z}}$ the pre-activation of $\mathbf{z}$ and $\tilde{\mathbf{h}}$ the pre-activation of $\mathbf{h}$
By applying again the chain rule, we get:

$$\frac{\partial z_i}{\partial x_j} = \sum_k \frac{\partial z_i}{\partial h_k}\frac{\partial h_k}{\partial x_j}$$

$$\frac{\partial z_i}{\partial h_k} = w_{ik}^{(2)}\mathbb{1}_{\tilde{z}_i > 0}$$

$$\frac{\partial h_k}{\partial x_j} = w_{kj}^{(1)} \mathbb{1}_{\tilde{h}_k > 0}$$

$$\frac{\partial z_i}{\partial x_j} = \mathbb{1}_{\tilde{z}_i > 0} \sum_{k=1}^{dh} \mathbb{1}_{\tilde{h}_k > 0} w_{kj}^{(1)} w_{ik}^{(2)}$$

$$\boxed{||J_e(\mathbf{x})||_F^2 = \sum_{i=1}^{dz} \mathbb{1}_{\tilde{z}_i > 0} \sum_{j=1}^{dx} \left(\sum_{k=1}^{dh} \mathbb{1}_{\tilde{h}_k > 0} w_{kj}^{(1)} w_{ik}^{(2)} \right)^2}$$

ECOLE DOCTORALE

Titre : Modélisation à Base de Réseaux de Neurones des Performances des Plateformes Cloud

Mots clés : Apache Spark, Apprentissage profond, Analyse de données, Bases de données, Cloud Computing, Representation Learning, Systèmes de recommandation.

Résumé : L'analyse des données en utilisant des ressources cloud est désormais omniprésente dans l'activité des entreprises qui s'engagent dans une transformation digitale pour mieux comprendre les données volumineuses dont elles disposent.
La modélisation des performances des plateformes cloud utilisées dans ce contexte est une nécessité pour pouvoir garantir une bonne performance des requettes réparties (appelées jobs) ainsi qu'une meilleure gestion des ressources cloud. Les techniques de modélisation traditionnelles ne s'adaptent ni à la diversité de ces jobs ni aux différents comportements des systèmes distribués. Dans cette thèse, nous proposons des techniques récentes de Deep Learning pour pouvoir automatiser cette tâche de modélisation avec un focus en particulier sur la plateforme Spark utilisée pour les calculs distribués. Au coeur de nos travaux de recherche, on présente la notion d'apprentissage d'embeddings, vecteurs capables de décrire de façon compacte les caractéristiques fondamentales des différents jobs. Nous montrerons dans cette thèse comment ces embeddings permettent une meilleure prédiction des performances des jobs sous différentes configurations du système de calculs répartis. Nous aborderons aussi une étude de différents choix de modélisation à base de réseaux de neurones répondant à nos besoins.
En premier temps, nous présenterons des méthodes d'apprentissage d'embeddings de la famille des systèmes de recommandation avec un focus en particulier sur les méthodes de filtrage collaboratif. Ensuite nous présenterons des techniques d'aprentissage d'embeddings basés sur des auto-encodeurs et des réseaux de neurones siamois qui prennent en compte plus d'informations profilés lors de l'exécution des jobs cloud.
Les résultats de nos expériences révèlent les forces et les limites des différents choix de modélisation. Nos expériences dévoilent aussi des performances supérieures d'une méthode qu'on propose par rapport à l'état de l'art dans la modélisation des systèmes de gestion de base de données.

Title : Neural-Based Modeling for Performance Tuning of Cloud Data Analytics

Keywords : Apache Spark, Cloud computing, Deep Learning, Databases, Data Analytics, Representation Learning, Recommender systems.

Abstract : Cloud data analytics has become an integral part of enterprise business operations for data-driven insight discovery.
Performance modeling of cloud data analytics is crucial for performance tuning and other critical operations in the cloud. Traditional modeling techniques fail to adapt to the high degree of diversity in workloads and system behaviors in this domain. In this thesis, we bring recent Deep Learning techniques to bear on the process of automated performance modeling of cloud data analytics, with a focus on Spark data analytics as representative workloads. At the core of our work is the notion of learning workload embeddings (with a set of desired properties) to represent fundamental computational characteristics of different jobs, which enable performance prediction when used together with job configurations that control resource allocation and other system knobs. Our work provides an in-depth study of different modeling choices that suit our requirements.
Throughout this manuscript, we first go over methods that learn job embeddings by mapping our problem to a recommender systems framework. We focus particularly on the family of collaborative filtering techniques. Afterwards, we present autoencoders and siamese neural networks from the representation learning family that leverages more content collected while cloud jobs are being profiled during their execution.
Results of extensive experiments reveal the strengths and limitations of different modeling methods, as well as superior performance of our best performing method over a state-of-the-art modeling tool for cloud analytics.

Institut Polytechnique de Paris
91120 Palaiseau, France