Performance Modeling and Multi-Objective Optimization for Data Analytics in the Cloud

Khaled Zaouk

Master Thesis

Performance Modeling and Multi-Objective Optimization for Data Analytics in the Cloud

Khaled Zaouk

Supervised by: Prof. Yanlei Diao

Submitted in fulfillment of

Engineering Degree (Diplôme d'ingénieur) from Télécom Paristech

&

Masters Degree in Applied Mathematics from Ecole Polytechnique

Major: Data Science

Defended on September 20th 2017 in front of the jury

Prof. Joseph Salmon (From Télécom Paristech)

ACKNOWLEDGMENTS

First and above all, I praise God, the Almighty, for endowing me with health, patience, and knowledge to complete this work.

Sincere and special appreciation goes to my supervisor Prof. Yanlei DIAO, for her supervision and constant support. Her invaluable help of constructive comments and suggestions throughout the project have contributed to the success of this research.

I would like to thank as well the other contributors to this research project: Fei Song (Inria (Saclay)) and Zhao Cao (Beijing Institute of Technology) who helped me throughout my internship, and I would like to thank Wei Sheng who helped us in the data generation process.

I thank the jury member Prof. Joseph Salmon for his time to read and judge my work.

Sincere thanks to all my professors from the Lebanese University, Télécom Paristech and Ecole Polytechnique for providing me the necessary knowledge which helped me during this project. I would like to specially thank Prof. Florence D'Alché-Buc for supporting me with advices during my studies at Télécom Paristech.

I want to thank INRIA for providing me with the opportunity of doing this research internship, and I want to thank all the members of the CEDAR team at INRIA.

I would definitely thank my friends and colleagues for every moment we spent together, for every support and happy memories.

My deepest gratitude goes to my beloved parents and my sisters for their endless supply of love, prayers and encouragement.

ABSTRACT

Today's cloud service providers guarantee machine availabilities in their Service Level Agreement (SLA), without any guarantees on performance measures according to a specific cost budget, and running analytics on big data systems require the user not to only reserve the suitable cloud instances over which the big data system will be running, but also setting many system parameters like the degree of parallelism and granularity of scheduling. Of course, these parameters, as well as the choice of the cloud instances need to meet user objectives regarding latency, throughput and cost measures, which is a complex task if it's done manually by the user. Hence, the need to transform cloud service models from availabily to user performance objective rises and leads to the problem of multi-objective optimization.

Given different cost models, the optimizer will search a multi-dimensional space, compute execution plans that are not dominated by others (known as Pareto plans) and explore meaningful tradeoffs between different objectives to find the optimal plan for each analytical task. It's difficult to build a general purpose cost model for each new user objective because big data systems can run arbitrary analytics on heterogeneous hardware with very complex system behaviors. We consider analytical tasks encoded as dataflow programs as in Hadoop and Spark systems. When such dataflow programs are submitted to the cloud, we aim to provide a multi-objective optimizer that can automatically find an optimal execution plan of the dataflow program, which meets specific user performance objectives. Developing an optimizer for dataflow programs in the cloud raises two major challenges: The optimizer needs cost models for running complex dataflow programs in the cloud, and, it further needs a new algorithmic foundation for multi-objective optimization across user-specific objectives.

In this project, we aim to develop a principled optimization framework that can take complex dataflow program and offer guarantees among multiple user objectives including throughput, latency and cost with some promising initial results reported in [11].

My contribution to this project consists of building a prediction model for one of the performance objectives (latency). So, I tried to explore what accuracy deep learning and other machine learning algorithms provide while learning such models from the system measurements collected from running and other workloads. This performance model will be used by the multi-objective optimizer to construct a pareto optimal skyline.

Contents

List of Figures

List of Tables

List of Abbreviations

AE	Auto-Encoder
CV	Cross Validation
CVEE	Cross Validation Error Estimate
DBMS	DataBase Management System
DL	Deep Learning
FC	Fully-Connected
GB	Gradient Boosting
IO	Input-Output
MAPE	Mean Absolute Percentage Error
MSE	Mean Squared Error
NN	Neural Network
RF	Random Forest
SGD	Stochastic Gradient Descent

Terminology

Notation	Meaning
θ	Parallelism
φ	Batchinterval (in seconds)
ρ	Blockinterval (in milli-seconds)
λ	Input rate
C_j^i	$(\varphi_j^i \cup \rho_j^i \cup \theta_j^i \cup \lambda_j^i)_j^i$: a vector representing the jth configuration of dataflow program i
P^i	Dataflow program (a.k.a. job, query, workload) i
E^i	Vector of size p representing the extracted workload characteristics of P^i
$O_j^i(t)$	vector of dimension 151 representing traces of P^i ran with configuration C_j^i at time t
O_j^i	vector of dimension 151 representing average of traces of P^i ran with configuration C_j^i
$l_j^i(t)$	latency (target) corresponding to P^i with configuration C_j^i running at time t
l_j^i	Average latency (target) corresponding to P^i with configuration C_j^i

Note: Throughout this report, we use different nouns for "analytical task": ***workload***, ***job***, ***dataflow program*** *and* ***query*** *are all synonyms in our context.*

1 Problem Statement

Performing analytical tasks over streams can be done today using the popular data stream systems such as Flink, Storm (twitter) and Apache Spark. If someone has to perform an analytical task over a stream of URL clicks for example, he has to write a dataflow program that describes the analytical actions he wants to achieve over the stream, in the language supported by the streaming system. For example, some systems support CQL (Continuous Query Language), and some others use some higher level functions.

These systems address not only the volume of the data, but also the velocity at which data arrives. In order to run a certain query over these systems with the best performances, we need to configure such systems properly. Such configuration contains parameters related to parallelism and granularity of scheduling of the streaming system. The difficulty in modeling the performance measures comes from the fact that in big data systems, analytical tasks are coded in Java for example, and there are no fixed operators like in relational algebra. In addition, these analytical tasks often run on heterogeneous hardware which makes the problem more complex.

Our objective is to design the multi-objective optimizer such that it automatically tunes the streaming system to the appropriate configuration in order to sustain a certain value of input rate, under the performance goals of the user. The biggest technical issue in our project is building cost models for the multiple objectives. Modeling this objective is done by constructing a regression module on that objective to be used by the optimizer to search through configurations that meet the user requirements.

In our research, we use a well known Big Data System called Spark Streaming [1] over which we aim to build our multi-objective optimizer. As shown in figure 1, Spark starts running a new query (dataflow program) from some initial configuration. Then the multi-objective optimizer tries to see if there's a better configuration that meets performance goals. To do so, the multi-objective optimizer will use several regression modules over the performance objectives. Each regression module will be trained on a target (latency, throughput, ...) by using other query traces. The multi-objective optimizer will search a multi-dimensional space, compute execution plans that are not dominated by others (known as pareto plans) and explore meaningful tradeoffs between the differrent performance objective to finally tune Spark Streaming on the configuration that achieves best performances for the current query.

In this thesis, we focus on building regression modules for user objectives to be used by the multi-objective optimizer. To simplify, we model only one objective: the latency.

The most important challenge is that the workload description is missing. When the optimizer needs to predict the value of the latency obtained by running a certain query with a particular configuration of the streaming system, all we have is configuration: we don't have any numerical vectors (known as workload characteristics) that can describe the query

we want to run. In this master thesis, we propose to use deep learning to extract workload characteristic vectors (also called job encodings or job embeddings) for each analytical task (a.k.a query or job) and to learn performance models automatically. We then compare the deep learning approaches intensively with some baseline techniques that do not extract any workload characteristics vector and try to learn the latency models by only using the configuration parameter values.

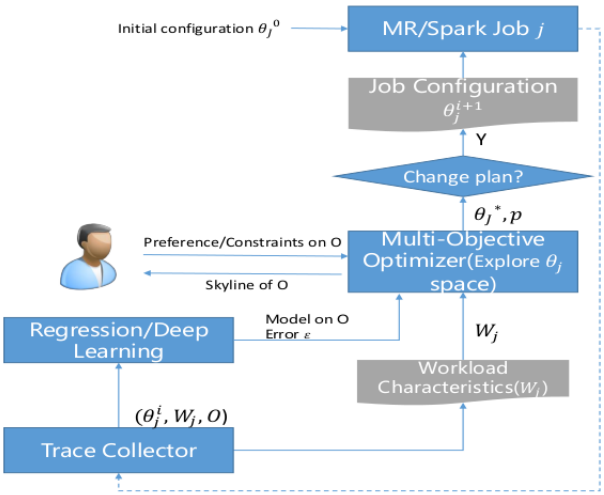

Figure 1: Multi-Objective optimization workflow

We start our research by building a single objective regression model for Spark Streaming. This model needs to predict the latency using mainly 4 features that constitute the system parameter (also called configuration): ***batch interval***, ***block interval***, ***parallelism*** and ***input rate*** (detailed later in section 9.5.4). We try to compare 3 different approaches:

- The baseline approach: this approach learns the latency for each query by only using information relative to the configuration parameters. It considers each different query as a category and uses categorical encoding for the query (onehot encoding). So, this approach doesn't use any additional information that describe numerically the query for which we want to predict the latency.
- Deep Learning embedding architecture: this approach tries to learn at the same time the latency obtained for a particular query with each configuration of the streaming system, and the workload encoding (vector that can describe this query).
- Deep Learning auto-encoder architecture: this approach tries to first extract some vector that can encode the query, and then tries to predict the latency by using two types of information: the configuration parameters as well as the workload (query) encoding.

We intend in our research to answer these questions: Is it possible to learn meaningful workload characteristics? What additional benefits does deep learning offer compared to the baseline approach?

2 System Environment and System Abstraction

In this chapter, we introduce the streaming system environment and abstraction to the reader. Then, we state the formal description of our modelling problem.

2.1 System Environment

In our modeling work we use Spark Streaming over a cluster of 6 nodes and we collect traces from the different workloads consisting of:

- Application level measures (Spark related metrics) collected through Spark listener
- System measures (CPU, memory, IO, network metrics) collected using Nmon

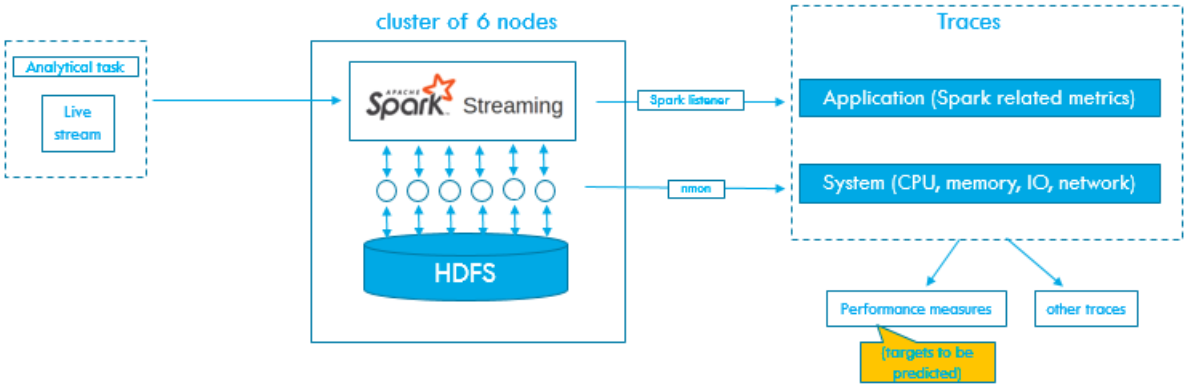

Figure 2: System Environment

2.2 System Abstraction

A complex analytical task i can be modeled as a logical dataflow program denoted as P^i and shown in figure 3. When we apply a particular value of the configuration vector $C_j^i = (\varphi, \rho, \theta, \lambda)$ to that dataflow program, then we can get a closer look on a MR pair in the physical execution plan in figure 4. This configuration parameters determine the number of tasks to be launched in a map/reduce phase, and thus determines the number of mapper and reducer nodes.

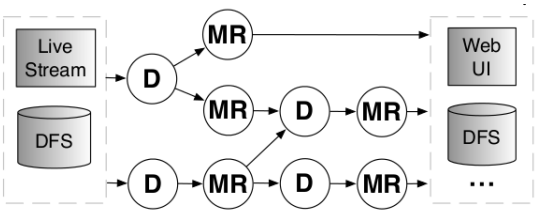

Figure 3: Logical plan of a dataflow program

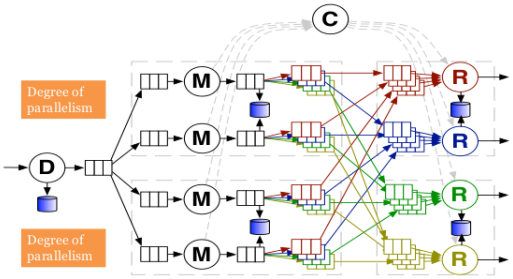

Figure 4: Physical plan of a dataflow program

2.3 Formal Description of the problem

2.3.1 Workload characteristics definition

We assume, for each dataflow program (also called job, or workload, or analytical task), there exists a workload characteristics vector which is a invariant to time and configurations. The formal definition is:

Definition 1 Workload Characteristics *E^i is a real valued vector satisfying three conditions:*

1. **Independence**. *E^i should be independent from the other causation factors (Configuration).*
2. **Invariance**. *For the same logical dataflow program, E^i should be (approximately) an invariant.*
3. **Reconstruction**. *E^i should carry all the information to reconstruct O^i_j, given C^i_j.*

2.3.2 Formal description of the modelling system

Under our assumptions, the modelling system can be abstracted as a deterministic function F s.t.

$$F(C_j^i, E^i) = l_j^i$$

Interpreted as an optimization problem, our goal is to find function F^* such that:

$$F^* = \arg\min_F \quad \text{avg}(\text{distance}(F(C_j^i, E^i), l_j^i)$$

where average is taken among all the training points such that F^* will be a reasonably good regressor which is able to generalize well over unseen points.

2.3.3 Optimization target for workload characteristics

We still need to define the optimization target for calculating the workload characteristics. Mathematically, the independence and invariance conditions can be captured by:

$$E^{i*} = \arg\min_{E^i} \quad \text{var}(E_j^i) \tag{2.3.3.1}$$

Where E_j^i is the encoding obtained with a particular configuration C_j^i of a dataflow program i, by using the vector of traces O_j^i. The variance is taken for the same logical dataflow program i, among different observations O_j^i (obtained from different configurations C_j^i) with j varying.

The third condition (reconstruction) can be captured by:

$$E^{i*} = \arg\min_{E^i} \quad \text{avg}(\text{distance}(F'(C_j^i, E^i), O_j^i) \tag{2.3.3.2}$$

where F' is a similar function to F, and is defined by:

$$F'(C_j^i, E^i) = O_j^i$$

In practice, we need an algorithm to construct E^i from existing data, such that it satisfies the formulas 2.3.3.1 and 2.3.3.2 at the same time. Sometimes the two conditions contradict with each other, so how to construct E^i satisfying both requirements is the key to our work.

3 Data generation, preprocessing and data splits

This chapter describes the different workloads (queries) P^i used in our benchmark, and details their generation process. It then goes through the preprocessing operations done on data before fitting any regressor, and finally it shows how data is split inside each training setting.

3.1 Workloads

In our work, we used a dataset that consists of all the requests made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998. During this period of time the site received 1,352,804,107 requests. From this dataset, we created an artificial data stream source that pumps data to the streaming system receivers. The analytical tasks that we used in our modeling can be expressed in CQL (Continous Query Language).

Workload 1 This analytical task is a windowed aggregate query with selectivity control that computes the number of clicks made by each user every φ seconds.

```
SELECT userId, COUNT(*) as counts
FROM UserClicks [range φ slide φ]
GROUP BY userId
```

Workload 2 This analytical task is a global aggregate query with selectivity control that computes the number of times a URL has been visited and keeps only URLs with more than 1000 visits. The computation is done on the batches as they are received, but the output of this query is obtained when the stream ends.

```
SELECT URL, COUNT(*) as counts
FROM UserClicks
GROUP BY URL
HAVING counts > 1000
```

Workload 3 This analytical task is a windowed aggregate query with selectifity and window control that computes the number of clicks made by each user over the last 30 seconds, and updates the count every 20 seconds.

```
SELECT URL, COUNT(*) as counts
FROM UserClicks [Range 30s slide 20s]
GROUP BY URL
HAVING counts > 1000
```

Workload 4 This analytical task is a windowed operation with control over the window size that keeps up reporting every φ second(s) the ids of the users who've been visiting the website during the last 30 seconds. Note that here not all resulting tuples are flushed to HDFS.

```
SELECT userId
FROM UserClicks [range 30s slide φ]
GROUP BY userId
```

Workload 5 This analytical task is a windowed join with selectivity control. The join operation is done between the data stream (corresponding to clicks) and a dataset stored in HDFS that contains the page ranks. It computes for every user, the sum of the ranks of the pages he has visited over the last 30 seconds, and updates the results every φ second(s). Note that here not all resulting tuples are flushed to HDFS.

```
SELECT userId, SUM(pageRank)
FROM Rankings and UserClicks [Range 30s slide φ]
WHERE Rankings.pageURL = Userclicks.URL
GROUP BY userId
```

Workload 6 This analytical task is the same as the previous one with the difference that all resulted tuples are flushed to HDFS in here. The CQL equivalent of this workload is the same as the previous one.

```
SELECT userId, SUM(pageRank)
FROM Rankings and UserClicks [Range 30s slide φ]
WHERE Rankings.pageURL = Userclicks.URL
GROUP BY userId
```

3.2 Data generation

After writing the Scala equivalent of the previous CQL queries, we ran the analytical tasks using the Spark Streaming environment earlier introduced in section 2.1 for different system configurations $C_j = (\varphi, \rho, \theta, \lambda)_j$. At each time we choose a value of λ, we make sure to set this value in both the artificial data pump and the Spark streaming system. For each particular configuration C_j^i corresponding to P^i, we obtain a vector of traces $O_j^i(t)$ as well as $l_j^i(t)$ corresponding to latencies obtained after processing each batch.

Data Generation process is time consuming, and changing the configuration is only possible every 40 minutes approximately. There are 10 minutes to warm up and to reach steady state when a Spark streaming job is being run. The next 30 minutes correspond to trace collection, and during which we construct $l_j^i(t)$ and $O_j^i(t)$. So it's possible to run a workload (query)

on at most 36 configurations per day. This is why our multi-objective optimizer needs a strong regressor capable of making good predictions in order to avoid wasting computation time. Note that both $l_j^i(t)$ and $O_j^i(t)$ are available only when we want to train the regressor, but when it comes to reality, it's not possible to obtain $O_j^i(t)$ before running the job on configuration C_j^i. This is what makes our problem not a classical machine learning one.

3.3 Preprocessing

The first step we did in our preprocessing is to take the average of the latency and observation vectors for each particular configuration C_j^i since we're interested in predicting the average latency obtained for C_j^i and not predicting the time series. This is because the time series doesn't make any sense in our context, and the notion of time comes from the fact that our system is processing one batch after another.

Before taking the average of observations and latencies, our data for a particular job i, run with n_i different configurations to obtain computations over k batches is on the form:

$$\begin{pmatrix} C_1^i & O_1^i(t_1) & O_1^i(t_2) & ... & O_1^i(t_k) \\ C_2^i & O_2^i(t_1) & O_2^i(t_2) & ... & O_2^i(t_k) \\ C_3^i & O_3^i(t_1) & O_3^i(t_2) & ... & O_3^i(t_k) \\ ... & ... & ... & ... & ... \\ C_{n_i}^i & O_{n_i}^i(t_1) & O_{n_i}^i(t_2) & ... & O_{n_i}^i(t_k) \end{pmatrix}$$

with each $O_j^i(t_k)$ represents a vector of dimension 151.

In order to keep only the average for each particular configuration C_j^i, we keep:

$$\bar{O_j^i} = \frac{1}{k} \sum_{t=t_1}^{t_k} O_j^i(t)$$

The same thing applies to latency:

$$\bar{l_j^i} = \frac{1}{k} \sum_{t=t_1}^{t_k} l_j^i(t)$$

For the ease of notation, whenever we drop the t, we consider the average value: $l_j^i \equiv \bar{l_j^i}$ and $O_j^i \equiv \bar{O_j^i}$. So now the data corresponding to Job i, which was run on n_i different configurations looks like:

$$\begin{pmatrix} C_1^i & \bar{O}_1^i \\ C_2^i & \bar{O}_2^i \\ C_3^i & \bar{O}_3^i \\ ... & ... \\ C_{n_i}^i & \bar{O}_{n_i}^i \end{pmatrix} \equiv \begin{pmatrix} C_1^i & O_1^i \\ C_2^i & O_2^i \\ C_3^i & O_3^i \\ ... & ... \\ C_{n_i}^i & O_{n_i}^i \end{pmatrix}$$

We also scaled our training matrices (consisting of the configurations) and the average observations before fitting any regressor, but we kept latencies without scaling, so that we can interpret the numbers we obtain from the errors on prediction. Let's denote by ϕ_j^i the scaled vector of observations, and by c_j^i the scaled vector of configurations, so that a particular scaled configuration looks like: $c_j = (\varphi', \rho', \theta', \lambda')$ with $(\varphi', \rho', \theta', \lambda')$ are scaled representations of $\varphi, \rho, \theta, \lambda$ respectively.

$$\phi_j^i = \text{MinMaxScale}(O_j^i)$$

$$c_j^i = \text{MinMaxScale}(C_j^i)$$

3.4 Data Description, Evaluation metric and some statistics

In our experiments, we consider having two datasets D_1 consisting of the full data that has been collected and D_2 with $D_2 \subset D_1$ corresponding to traces from common configurations tested across all workloads. The number of samples in D_1 is detailed in Table 1.

In dataset D_2 the configurations were obtained from all the possible combinations from:

- Batch Interval (s): $\varphi \in \{5, 10\}$
- Block Interval(ms): $\rho \in \{200, 400, 600, 800, 1000\}$
- Parallelism $\theta \in \{18, 36, 54, 72, 96\}$
- Input Rate (Million of samples/second) $\lambda \in \{0.48, 0.66\}$

Which makes in total 100 configurations. However, for some workloads, some of these configurations failed, which left us at the end with 92 common configurations across all workloads.

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6
Number of samples	1356	449	294	289	125	216

Table 1: Total number of configurations for each workload in dataset D_1

Job	Min	Max	Median	Mean	Variation	Naive MAPE
1	1099	5857	2880.50	2904.96	23.11%	26.54%
2	1451	9697	3765.50	4017.33	30.91%	36.05%
3	3023	13009	6252.50	6501.09	23.27%	25.38%
4	1308	8271	3512.00	3663.00	30.20%	35.87%
7	1419	6465	3522.00	3633.58	26.81%	31.68%
8	1554	6423	3514.50	3506.24	26.73%	31.40%

Table 2: Some statistics about latencies (ms) in the dataset D_1

Evaluation metric

we use the Mean Absolute Percentage Error (MAPE) metric in order to evaluate the accuracy of our regression model. This means that if $\tilde{l}_j^i$ represents the predicted value of the latency for C_j^i, and l_j^i is the real value, and if I denotes the set of the job ids over which the error is calculated, and J_i the total number of configurations for job i, then the error can be calculated by:

$$MAPE = \frac{100}{m} \sum_{i \in I} \sum_{j=1}^{J_i} \frac{|l_j^i - \tilde{l}_j^i|}{l_j^i}$$

with m being the number of samples over which the error is calculated.

The Table 2 contains some statistics corresponding to Dataset D_1. The "Variation" column represents the mean absolute deviation from the average latency. The variation gives us a rough idea about the fluctuations of the values of the latency around the mean value. The "Naive MAPE" column represents the MAPE obtained if we have a naive regressor that always predict the mean value for each job. Any regression approach we should adopt, has to perform better than the error obtained with the naive regressor.

More precisely,

$$variation(job_i) = \frac{100}{J_i} \sum_{j=1}^{J_i} \frac{|l_j^i - l^i|}{l^i}$$

and

$$NaiveMAPE(job_i) = \frac{100}{J_i} \sum_{j=1}^{J_i} \frac{|l_j^i - l^i|}{l_j^i}$$

with l^i representing the average latency per job $l^i = \frac{1}{J_i} \sum_{j=1}^{J_i} l_j^i$ and J_i corresponds to the number of configurations on which job i was run.

The second dataset (D_2) contains 92 configurations for each job. These configurations are exactly the same across all other jobs in this dataset. In Table 3, we show similar statistics to those in Table 2 but obtained only using traces from common configurations. In both Tables 2 and 3 we can see that job 1 has the lowest variation from the mean value, and job 2 has the highest variation. When a job has a lower variation around the mean value with respect to other jobs, we expect it to be easier to be modeled, or at least, we expect to have lower values of MAPE for that job.

Job	Min	Max	Median	Mean	Variation	Naive MAPE
1	1267	3865	2557.00	2539.75	22.52%	25.72%
2	1558	9697	3673.50	4157.22	34.99%	41.95%
3	3023	11470	6391.50	6477.83	26.84%	30.91%
4	1308	6546	3136.50	3273.66	29.04%	33.97%
7	1419	6465	3344.00	3419.68	27.24%	32.18%
8	1554	6348	3538.50	3517.05	27.15%	31.70%

Table 3: Some statistics about latencies (ms) in the dataset D_2

We should note that features obtained from different O_j^i are highly correlated between themselves and with respect to the latency. This can be confirmed in figure 5 that represents the Heatmap of the absolute value of the empirical correlation between the features in the observation matrix obtained from dataset D_1

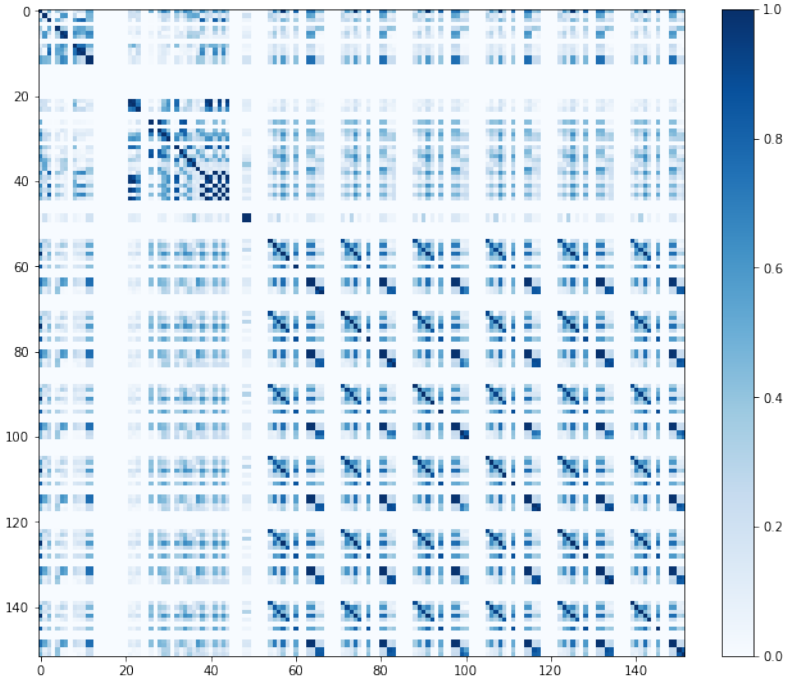

Figure 5: Heatmap of correlation between features of observations matrix (with invariant features)

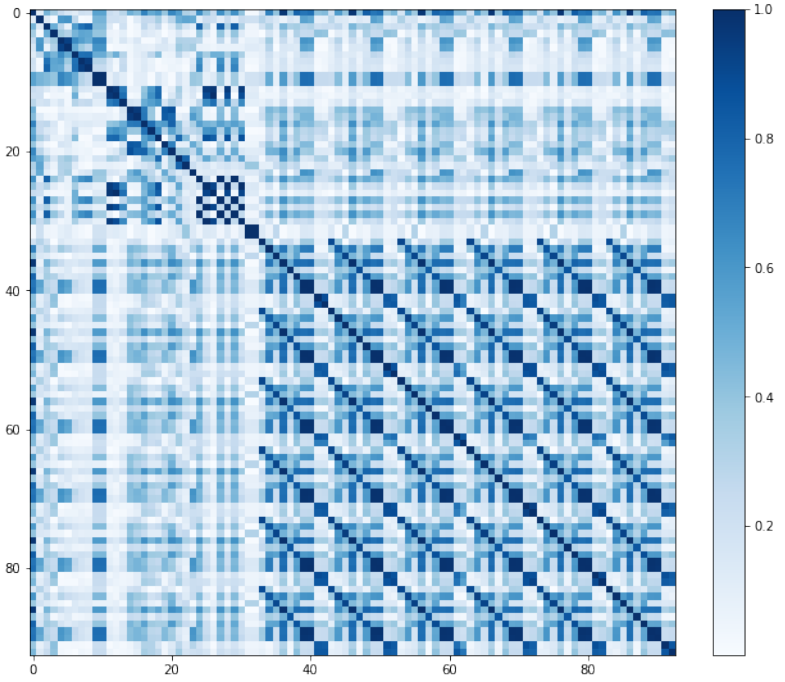

Figure 6: Heatmap of correlation between features of observations matrix (without invariant features)

A row in the "observations" matrix correspond to a vector O_j^i. On the diagonal of the correlation matrix shown in figure 5 we have some null values, this is due to the presence of some features which are invariant across all configurations and for all jobs, and that later on need to be removed by using a dimensionality reduction technique. If we just remove those invariant features from the observations matrix, we obtain the heatmap shown in figure 6.

This strong correlation between the observations and the latency (target to be predicted) can be seen by observing the first row (or first column) of the heatmap, and can be even illustrated in figure 7, where we can see the plot of the latency and the `mem_active` (a feature in the observations vector O_j^i) on one of the cluster nodes (node 2), after both of them are rescaled in $[0, 1]$ interval. Both curves are aligned, which once again means that the latency is correlated with this measure.

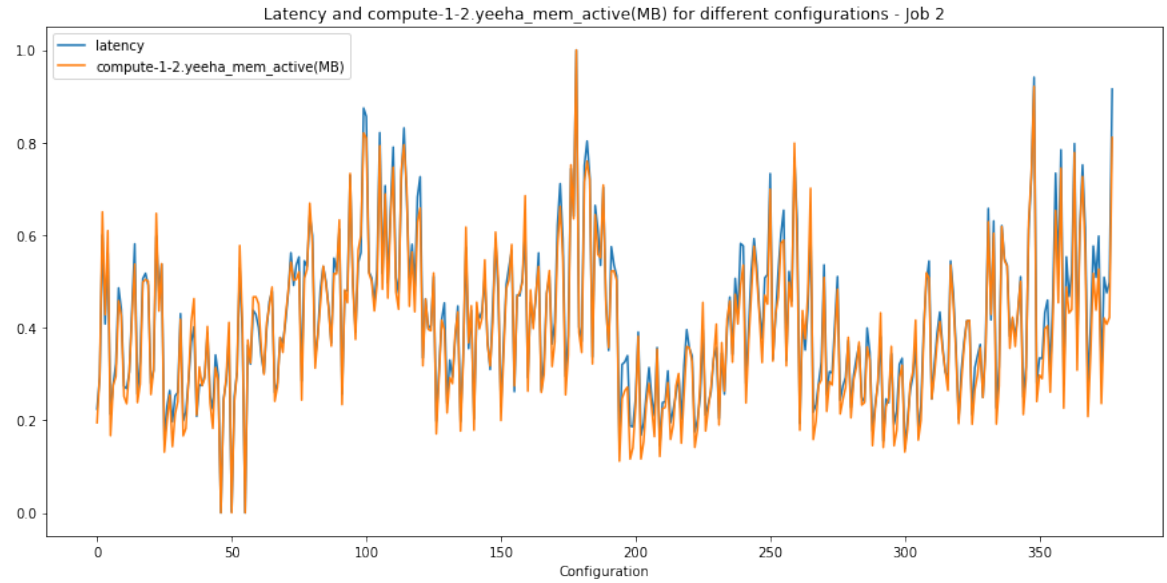

Figure 7: Average latency vs Average mem-active for some configurations in job 2

3.5 Training settings and Data splits

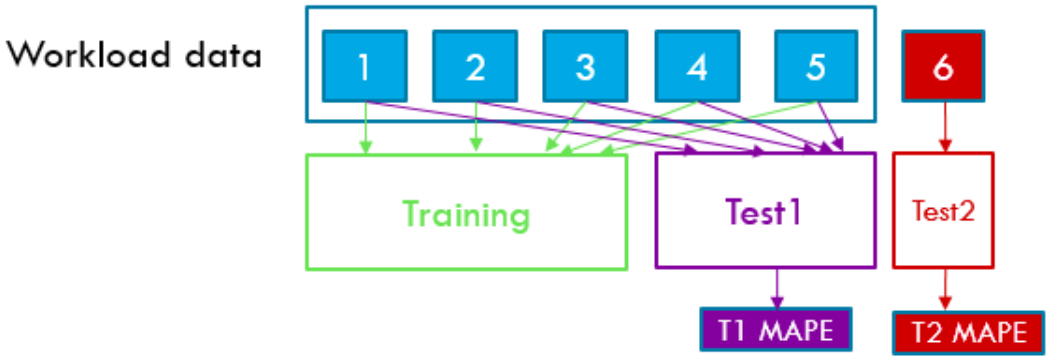

Figure 8: Data partitioning to training/validation and test sets in LO_6

The whole dataset consisting of 6 jobs $\{1, 2, 3, 4, 5, 6\}$ is used in our benchmark experiments. However, we define what we call a training setting LO_i (Leave out job i), the training setting in which we leave Job i out of the training samples. The purpose of defining such training setting is to emulate the real case scenario. In reality, the optimizer will need to use other workloads data to predict latencies over new workloads. So we define two types of errors: Type1 error which is evaluated on familiar jobs (jobs over which the regressor was trained), and which reflects the generalization power over unseen configurations from familiar jobs; and Type2 error which is evaluated on unfamiliar jobs (on jobs that the regressor has seen at most once: (job i in training setting LO_i)) and reflects the power of generalization over unseen jobs. Splitting the data into training, test1 and test2 for training setting LO_6 is illustrated in figure 8. We run our benchmark over 6 different training settings by leaving out one job at a time: LO_1, LO_2, LO_3, LO_4, LO_5 and LO_6. Note that in order to have some consistency across different training settings, when two different training settings share some same job id among familiar jobs (for example LO_1 and LO_2 share jobs 3, 4, 5 and 6), then the same splitting of samples between Training and Test1 is applied. This same splitting also applies when we do the cross validation using all the jobs by using all samples observed in the different Training splits (but not in Test1 splits). Note that we do the cross validation once for each regression algorithm, in order to have consistent parameters across different training settings. A review of cross validation has been added to Appendix (see section 9.1)

4 The baseline approaches for regression

The purpose of this series of experiments is to see whether training a regressor on multiple jobs can leverage additional information than training one regressor per job, and to see what errors we can achieve by trying to predict using only the workload id and the configuration parameters (without using the information from traces).

We intend to compare different machine learning algorithms performances in predicting the latency, and these algorithms are: Bagging, ExtraTrees, Gradient Boosting, K Nearest Neighbors, Random Forest, and SVM. A quick review over these machine learning algorithms can be found in Appendix (section 9.2). We start by choosing, using the cross validation technique, the hyper-parameters of these algorithms. Then, we exhibit the errors we obtain in both approaches: Standalone approach, and One-hot categorical encoding approach. In these approaches, we try first to fit different regressors, each one on a particular workload (and we call these experiments standalone regressors) and then later on, we try to fit a regressor on multiple workloads at the same time by using a categorical feature encoding of the jobs: the onehot encoding.

In both approaches, we don't use the traces collected while running previous Spark Streaming jobs, and the only information we use while trying to predict the latency is the job id and the system configuration $C_j^i = (\varphi, \rho, \theta, \lambda)$.

4.1 Hyper-parameter selection

In order to have consistent results across the different experiments and across different training settings, we did once a cross validation of the hyper-parameters of the different algorithms by training on the onehot encoding features (so our training matrix looks has a shape (m, p) where m is the number of training points contained in the dataset D_1 (after splitting data into training and test), and p is equal to 10 (p = 4 (dimension of C_j^i) + 6 (dimension of the onehot vector)). The best hyperparameters that we obtained for each regressor are illustrated below, and some details about the hyper-parameters as well as the space from which these hyper-parameters were selected can be found in Appendix section 9.2.

max_features	*n_estimators*
10	500

Table 4: Best hyper-parameters for Bagging regressor

max_depth	*max_features*	*n_estimators*
25	4	200

Table 5: Best hyper-parameters for ExtraTrees regressor

learning_rate	*loss*	*max_depth*	*max_features*	*n_estimators*
0.1	`lad`	4	5	500

Table 6: Best hyper-parameters for GradientBoosting regressor

n_neighbors	*weights*
11	`distance`

Table 7: Best hyper-parameters for K Nearest Neighbors regressor

max_depth	*max_features*	*n_estimators*
`None`	6	500

Table 8: Best hyper-parameters for Random Forest regressor

C	*epsilon*	*kernel*
10^6	10	`rbf`

Table 9: Best hyper-parameters for SVM

4.2 Experiment 1: Standalone regressors

4.2.1 Design

This experiment aims to understand how well a standalone regressor makes a prediction when trained on each job separately. In this experiment, we fit the regressor on $C_j^i \rightarrow l_j^i$. We compare the MAPE for the regression algorithms: Bagging, ExtraTrees, GradientBoosting, KNN, RandomForest and SVM on each job.

4.2.2 Experimental setup

This experiment is done under the bulk training framework, which means that we train the regressor on all the training data for the corresponding job, and we test on the split corresponding to test. We show the results obtained with the two datasets D_1 and D_2. The number of training and test points relative to each job in each dataset is shown in tables 10 and 11 .Since we're training on one single job and predicting on the same job, we represent here only type 1 MAPE. We set the values of the hyper-parameters for the different algorithms to those obtained in section 4.1 except `max_features` which was reset to 4 in this series of experiments.

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6
Number of trainings samples	1084	359	235	231	100	172
Number of test samples	272	90	59	58	25	44
total	1356	449	294	289	125	216

Table 10: Number of training and test points in D_1

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6
Number of trainings samples	73	73	73	73	73	73
Number of test samples	19	19	19	19	19	19
total	92	92	92	92	92	92

Table 11: Number of training and test points in D_2

We've used Scikit-Learn [12] implementation of the different regression algorithms. Our code was run over Python 3.5.2 with NumPy 1.12.1 and Scikit-Learn 0.18.1 libraries. To avoid randomness that comes from the stochastic behavior of some regression algorithms (especially algorithms with sampling like RandomForest), we fix the random seed of NumPy to the value 22.

4.2.3 Results and Analysis using dataset D_1 (all configurations)

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6	AVG
Bagging	0.98%	6.04%	4.74%	4.02%	4.26%	3.71%	3.96%
ExtraTrees	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
GradientBoosting	1.20%	7.81%	5.15%	6.34%	2.86%	3.56%	4.49%
KNN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RandomForest	0.95%	5.63%	4.69%	4.06%	4.08%	3.68%	3.85%
SVM	1.99%	10.83%	10.37%	7.90%	6.55%	6.13%	7.30%

Table 12: Training MAPE for standalone regressor using dataset D_1

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6	AVG
Bagging	2.87%	16.95%	13.96%	10.79%	11.71%	10.69%	11.16%
ExtraTrees	2.78%	17.68%	14.20%	11.57%	12.45%	10.99%	11.61%
GradientBoosting	2.03%	14.63%	14.57%	10.25%	13.32%	11.99%	11.13%
KNN	3.90%	16.02%	14.90%	12.01%	11.52%	13.59%	11.99%
RandomForest	2.86%	16.72%	13.51%	10.62%	11.37%	10.89%	11.00%
SVM	2.38%	15.66%	13.97%	10.66%	11.74%	11.67%	11.02%

Table 13: Type 1 MAPE for standalone regressor using dataset D_1

Observations

- We can notice that both KNN and ExtraTrees algorithms are overfitting due to the huge gap between training and test errors. Compared to all other algorithms, KNN is the worse for all the workloads. It's a naive algorithm that lacks generalization power.
- We have 3 patterns among all jobs:
  - High training & Test errors: This is the case for Job 2 with RF, Bagging, GB and SVM. This means that the regression algorithm wasn't able to learn well from the data in Job 2.
  - Low training & Test errors: This is the case for Job 1. Hypothesis H_1: These very low values of Type 1 error are due to the fact that we have the highest number of training samples for job 1.
  - Training error is low, but test is high and the gap between training and test is about 10%. This is the case of jobs {3,4,5,6} (where we're slightly overfitting)

4.2.4 Results and Analysis using dataset D_2 (common configurations)

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6	AVG
Bagging	1.71%	6.01%	4.81%	4.63%	4.40%	5.02%	4.43%
ExtraTrees	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
GradientBoosting	0.83%	4.67%	2.64%	2.35%	2.76%	4.09%	2.89%
KNN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RandomForest	1.60%	5.67%	4.79%	4.60%	4.41%	4.98%	4.34%
SVM	1.40%	9.76%	6.36%	6.07%	6.15%	7.41%	6.19%

Table 14: Training MAPE for standalone regressor using dataset D_2

Training Settings	Job_1	Job_2	Job_3	Job_4	Job_5	Job_6	AVG
Bagging	3.95%	13.91%	11.12%	17.92%	12.61%	9.88%	11.56%
ExtraTrees	3.50%	12.37%	11.51%	15.92%	11.71%	10.38%	10.90%
GradientBoosting	5.75%	15.42%	13.38%	18.35%	14.95%	14.50%	13.72%
KNN	15.90%	20.14%	18.83%	29.11%	21.78%	20.85%	21.10%
RandomForest	3.72%	13.16%	11.23%	17.41%	12.65%	9.89%	11.34%
SVM	2.61%	20.01%	12.04%	19.20%	13.38%	12.05%	13.21%

Table 15: Type1 MAPE for standalone regressor using dataset D_2

Observations

- If we try to compare column-wise the tables 13 and 15, we have approximately the same order of magnitude for the MAPE calculated on each job, with slightly better results when we use the full data (dataset D_1).
- By using only the common configurations, job 4 has a relatively high T1 MAPE (compared to other jobs). Later on, we'll see if we train a regressor on a mixture of jobs on the common configurations, if job 4 is capable of leveraging information coming from other jobs.
- We can reject the hypothesis H_1, since even if we have exactly the same number of configurations across the different workloads, we still have lowest values of errors for job 1. This means that job 1 is an easy job to predict.

4.3 Experiment 2: Onehot encoding

4.3.1 Design

In this section we discuss a baseline approach where each workload is represented by its job id (categorical) in an onehot encoding fashion. So, we fit the regressor on $(C_j^i, E^i) \rightarrow l_j^i$, where E^i represents the onehot encoding of the job i.

This experiment is designed for 3 purposes:

- To examine whether when we train a regression algorithm on multiple jobs, its error in predicting one job is lower than the error induced by a standalone regressor.
- To check how well the baseline approach can leverage generalities across workloads by only using categorical features (Onehot encoding) (to predict unseen workloads).
- To confirm whether having more data improves or not Type1 and/or Type2 errors.

4.3.2 Experimental setup

Using the onehot encoding, each workload is considered to be equidistant from any two other workloads. In other terms, the distance between two different workloads is the same. Under this encoding schema, the workload vectors look like:

$$E1 = (1, 0, 0, 0, 0, 0)$$

$$E2 = (0, 1, 0, 0, 0, 0)$$

$$E3 = (0, 0, 1, 0, 0, 0)$$

$$E4 = (0, 0, 0, 1, 0, 0)$$

$$E5 = (0, 0, 0, 0, 1, 0)$$

$$E6 = (0, 0, 0, 0, 0, 1)$$

We train on some configurations of familiar jobs, and predict on:

- New configurations of the same familiar jobs, so we get the T1 MAPE.
- New configurations of some new unfamiliar jobs, so we get the T2 MAPE.

So we define 6 different training settings inside which we leave out one job for evaluating the T2 MAPE. LO_i is the training setting in which we leave out job i when we train the regressor.

This experiment is also done under the bulk training framework which means we train the regressor directly on all training data from the familiar jobs along with the unique sample from the unfamiliar job.

4.3.3 Results and Analysis using dataset D_1 (all configurations)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Bagging	4.71%	2.27%	2.59%	2.67%	2.78%	2.76%	2.96%
ExtraTrees	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
GradientBoosting	8.89%	4.25%	5.07%	5.03%	5.38%	5.31%	5.66%
KNN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RandomForest	4.43%	2.16%	2.47%	2.57%	2.66%	2.62%	2.82%
SVM	10.42%	5.46%	5.86%	6.09%	6.47%	6.39%	6.78%

Table 16: Training MAPE for baseline regressor with onehot encoding (dataset D_1)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Bagging	13.15%	6.39%	7.54%	7.91%	7.96%	7.87%	8.47%
ExtraTrees	13.42%	6.32%	7.13%	7.77%	7.87%	7.78%	8.38%
GradientBoosting	12.01%	5.64%	6.62%	6.81%	6.99%	6.88%	7.49%
KNN	13.68%	7.82%	8.44%	8.48%	9.13%	8.99%	9.42%
RandomForest	12.91%	6.14%	7.21%	7.65%	7.70%	7.59%	8.20%
SVM	12.40%	6.28%	6.79%	7.23%	7.41%	7.34%	7.91%

Table 17: Type1 MAPE for baseline regressor with onehot encodings (dataset D_1)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Bagging	26.38%	21.27%	45.32%	13.14%	10.78%	12.32%	21.53%
ExtraTrees	25.59%	17.99%	35.77%	14.09%	11.19%	12.41%	19.51%
GradientBoosting	26.22%	16.74%	19.69%	12.21%	11.07%	11.45%	16.23%
KNN	78.60%	20.44%	35.51%	20.35%	20.47%	13.17%	31.42%
RandomForest	29.16%	19.08%	44.36%	13.67%	10.39%	12.18%	21.47%
SVM	76.02%	18.12%	18.87%	11.37%	10.97%	13.53%	24.81%

Table 18: Type2 MAPE for baseline regressor with onehot encodings (dataset D_1)

Observations

- By observing Table 17, we see that GradientBoosting outperforms all other algorithms in all training settings, and achieve on average, a Type 1 error as low as 7.49%. Also from Table 18 GradientBoosting outperforms on average the other regression algorithms and achieve on average a Type2 MAPE of 16.23%
- The worst errors in terms of Type 2 MAPE are obtained in the training settings LO_3 which means that Job 3 is quite different from other jobs, and by training on those jobs, we can't predict very well latencies corresponding to Job3. This may be due to the overlapping sliding window that is used in job 3 and which is not used in other jobs as in section 3.1. Furthermore, if we check the feature importance with the RandomForest regressor as in figure 9, we see that the most important feature is the categorical feature corresponding to job 3: That means that there is a discrimination between job 3 and all other jobs.
- The bad performance of RandomForest regressor and some other ensemble methods compared to the performance of GradientBoosting in Type2 error may be linked to the sampling in all ensemble method. In other words, when building the trees, all ensemble algorithms used except Gradientboosting are trained on subsets of the full training data, while in gradient boosting, each tree is trained using information from all the data. And since the sample corresponding to the unfamiliar job is unique,

then there's a little chance that it's going to appear enough in the fitted trees of other ensemble methods.

Now let us add more profiling information by checking the detailed Type1 MAPE obtained with the GradientBoosting algorithm, as well as feature importance obtained with 2 different regressors.

	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG	Standalone
Job 1	-	2.21%	2.56%	2.16%	2.26%	2.17%	2.27%	2.03%
Job 2	14.18%	-	13.51%	13.71%	14.20%	13.81%	13.88%	14.63%
Job 3	12.14%	11.86%	-	12.77%	11.57%	12.54%	12.18%	14.57%
Job 4	9.90%	9.75%	9.91%	-	10.43%	10.69%	10.14%	10.25%
Job 5	11.01%	11.14%	11.23%	11.29%	-	11.05%	11.14%	13.32%
Job 6	10.72%	9.89%	10.69%	10.86%	10.86%	-	10.60%	11.99%

Table 19: Detailed T1 MAPE for Baseline experiments with GB regressor (dataset D_1)

Each row in the table 19 represents the errors obtained when we try to predict on a particular job across the different training settings. The last column of the table 19 represents the Type1 MAPE obtained with standalone regressors trained with GradientBoosting on all available configurations. It's the 4th row of the Table 13. We observe slight improvement when we use information from all the regressors ("AVG" column in table 19 compared to "Standalone" column)

Feature importance using RF and GB regressors

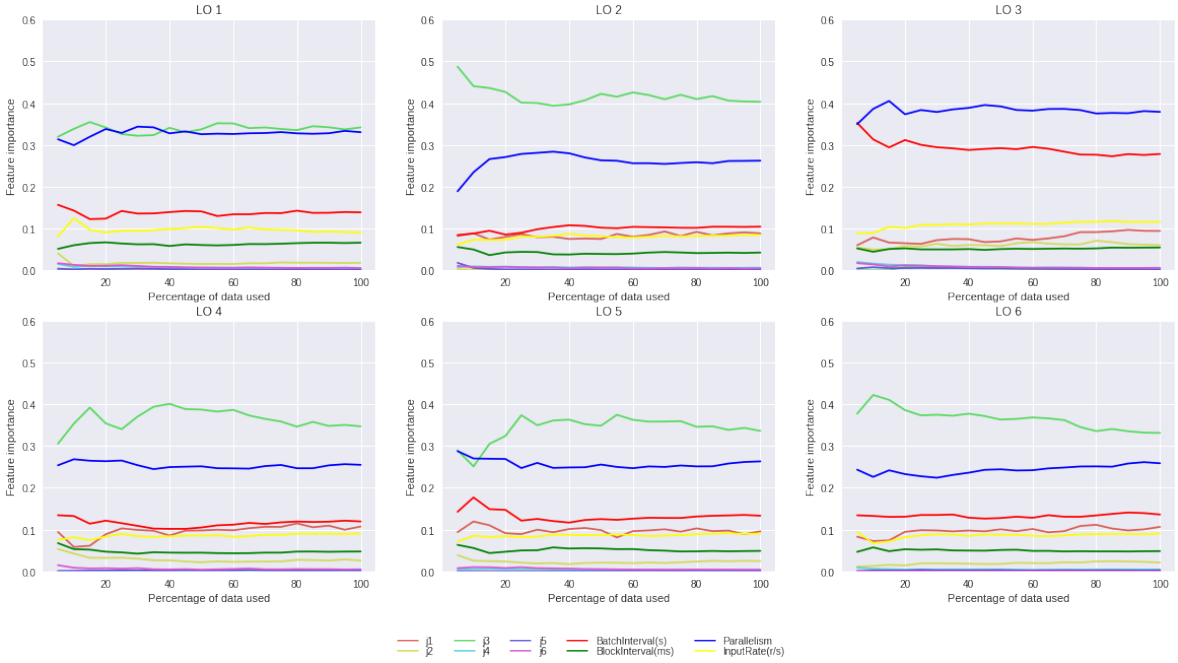

Figure 9: Feature importance using RandomForest regressor

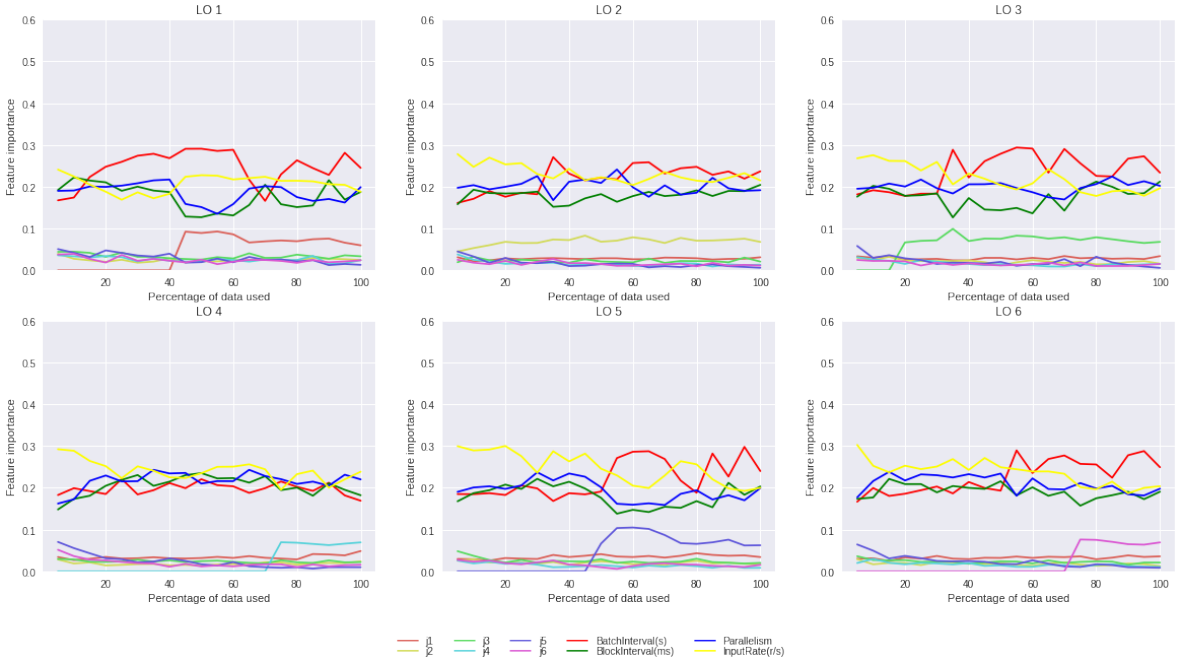

Figure 10: Feature importance using Gradient Boosting regressor

- When using a RandomForest regressor, the most important feature across all training settings (except LO 3) is the categorical bit for job 3. This means that the regressor estimation is highly dependent of the id of the job whether it's job 3 or any other job. The second most important feature with the random forest regressor is the Parallelism, then comes BatchInterval, blockInterval and InputRate and the other categorical bits. This is true for all training settings different than LO3, but in LO3, since the regressor is fitted on 1 sample from job3, then it's not leveraging that feature, and is not well performing while predicting job 3 (Type 2 error of 44.36%).
- When using a GradientBoosting regressor, It's leveraging all the configuration parameters equally (all of the 4 parameters: Parallelism, BatchInterval, BlockInterval InputRate) have an importance around 0.2. Another note is that after those 4 features comes the bit feature corresponding to the currently left out job. For example, in LO1, the most important feature after the 4 configuration parameter is the bit relative to job 1. This means that gradientboosting is leveraging the information it's receiving from the only 1 sample relative to the unfamiliar job. And this is why gradient boosting achieves a very good error in terms of type2 error.

4.3.4 Results and Analysis using dataset D_2 (common configurations)

This experiment is intended to confirm whether having less data in each training setting worsens Type1 and/or Type2 error. It is also intended to see if we can improve error for job 4 by using information from other regressors when we have a limited number of seen samples.

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Bagging	4.60%	4.07%	3.96%	4.14%	4.23%	4.09%	4.18%
ExtraTrees	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
GradientBoosting	6.57%	5.41%	5.85%	5.82%	5.91%	5.95%	5.92%
KNN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RandomForest	4.57%	3.75%	4.02%	4.02%	4.10%	4.01%	4.08%
SVM	8.96%	7.00%	7.76%	7.87%	7.80%	7.69%	7.85%

Table 20: Training MAPE for baseline regressor with onehot encodings (dataset D_2)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Bagging	12.41%	10.71%	10.98%	10.13%	11.31%	11.16%	11.12%
ExtraTrees	12.26%	10.41%	10.67%	9.56%	10.43%	10.54%	10.65%
GradientBoosting	12.71%	10.52%	10.89%	9.07%	9.99%	10.35%	10.59%
KNN	22.30%	21.48%	21.86%	19.65%	21.01%	21.19%	21.25%
RandomForest	12.98%	10.96%	11.16%	10.00%	10.92%	11.52%	11.26%
SVM	12.98%	10.33%	10.04%	8.96%	9.22%	10.78%	10.38%

Table 21: Type1 MAPE for baseline regressor with onehot encodings (dataset D_2)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Bagging	32.86%	18.18%	42.50%	14.19%	11.10%	11.39%	21.70%
ExtraTrees	37.66%	18.77%	40.92%	15.50%	11.95%	10.82%	22.60%
GradientBoosting	20.66%	17.73%	30.04%	15.00%	11.48%	10.37%	17.54%
KNN	59.83%	23.42%	42.10%	28.58%	23.52%	15.60%	32.17%
RandomForest	34.57%	18.43%	42.44%	15.05%	10.62%	10.55%	21.94%
SVM	32.33%	19.88%	22.74%	11.22%	12.44%	11.90%	18.42%

Table 22: Type2 MAPE for baseline regressor with onehot encodings (dataset D_2)

Observations

- Having more training data improves significantly Type 1 MAPE: When we have more training data as in Table 17, the T1 MAPE are lower for all algorithms and for all training settings than those we had in Table 21 where we trained on less data (Except LO_1 for Bagging and ExtraTrees)
- Having more training data cannot improve Type2 MAPE since the additional training data is not related to the unseen job over which Type2 MAPE is calculated. This can be confirmed by the fact that we don't have any trend by comparing Tables 22 and 18.
- Job 4 can be predicted with a Type2 MAPE of 15% with GradientBoosting regressor (Table 22) vs 18.35% with standalone regressors (Table 15), which confirms once again that baseline experiments can leverage additional information from other jobs to improve one job's predictions!

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG	Standalone
Job 1	-	4.08%	4.46%	3.83%	3.34%	3.33%	3.81%	5.75%
Job 2	11.41%	-	10.68%	10.06%	11.70%	10.83%	10.94%	15.42%
Job 3	11.37%	11.45%	-	10.23%	10.81%	11.32%	11.04%	13.38%
Job 4	16.20%	15.30%	15.96%	-	15.26%	15.07%	15.56%	18.35%
Job 5	14.46%	12.38%	13.71%	12.16%	-	11.19%	12.78%	14.95%
Job 6	10.12%	9.38%	9.63%	9.08%	8.85%	-	9.41%	14.5%

Table 23: Detailed T1 MAPE for Baseline experiments with GB regressor (dataset D_2)

Observations:

- If we have only a limited number of configurations over which we train the regressors, then baseline approach provides significant improvements on type1 error obtained from standalone regressor, that is by leveraging data coming from other jobs to improve one job's predicted value.

4.3.5 Pros and Cons

Pros

- The time required to fit a Gradient Boosting regressor with the actual data is relatively small compared to fitting a neural network

Cons

- One needs to know explicitly the id of the job he's dealing with. If we don't have a clue about the job id, and we have only collected observations, we're not able to predict a latency by using this model.
- If we have n different jobs, we'll have n different binary sparse features, and thus this approach can't scale.

5 Deep Learning models

5.1 Embedding

5.1.1 Design

This experiment aims to study whether an embedding neural network architecture can bring better performance than the baseline regressors. A quick review on training neural network and selecting the best hyper-parameters can be found in Appendix (section 9.3). The idea of embedding comes from the neural network architectures used in recommendations: there's a need to embed the information relative to a user represented by its id, in a vector, and there's as well the need to embed the information relative to the item by its id in deep recommender systems. Similarly, in the context of learning workload characteristics, there's a need in representing each workload by a vector describing its characteristics, so that we can compute some distance and similarity measures over different workloads.

The architecture used in the embedding approach is represented in the figure 11. Note that "FC" is an abbreviation for "Fully-Connected" and means a fully-connected layer. In the figure 11, we showed 3 fully connected layers, but in reality, we may have more or less fully-connected layers according to what depth we obtain during hyper-parameter selection.

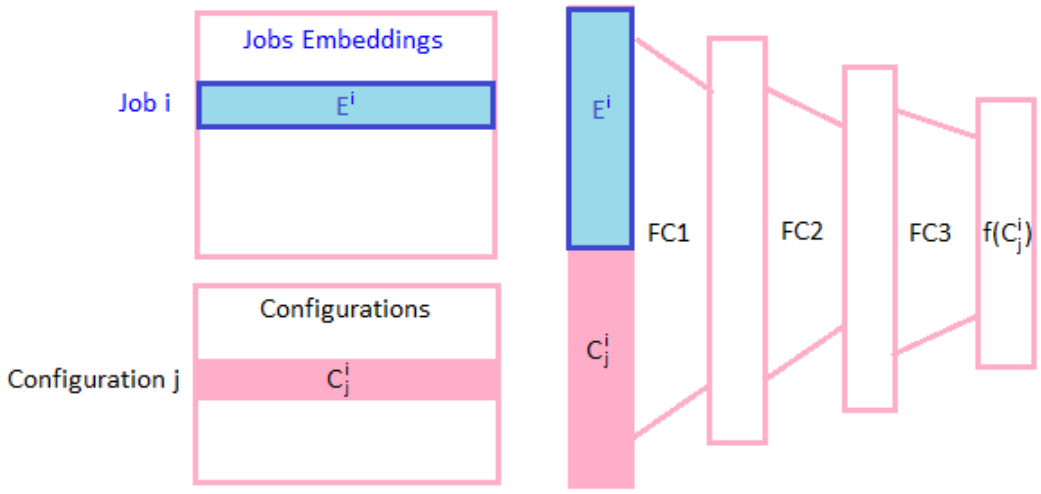

Figure 11: Embedding Deep Learning Architecture

- Shape of the embedding matrix: $(n \times p)$
- Shape of the configuration matrix: $(m \times 4)$

with n is the total number of jobs (=6 in our benchmarks), p is the dimension of the embedding determined later on by cross validation, and m is the total number of training points (it depends from each training setting).

The input of the neural network architecture is (i, c_j^i) and the output is $f(c_j^i)$, where i is the id of the job. This neural network architecture consists of:

- An embedding layer (weight matrix is E) that provides for a given job id i a vector E^i (ith row of the embedding matrix E)
- A concatenation layer: a layer that concatenates the embedding vector E^i of the current job i, with the configuration vector c_j^i
- Several fully connected layers whose first input is the concatenation of E^i with c_j^i (a vector of dimension $= p + 4$) and last output is $f(c_j^i)$

Note: E^i contributes implicitly to f: $f(E^i, c_j^i) \equiv f(c_j^i)$ but E^i is omitted since it's not fixed, but it's learnt while training the neural network.

The embedding matrix having n=6 rows is learned during the back-propagation. It can be treated like any weight matrix in a neural network. Thus, this embedding matrix is randomly initialized, like any other weight matrix, before the neural network is trained. Then, the embedding vector E^i of a specific job i is updated using gradients of the loss function computed from samples relative to that job id.

If E denotes the embedding matrix, and i denotes the id of the job that is provided as input to the neural network, then mathematically the vector corresponding to the job i can be obtained from the embedding matrix by simply multiplying this matrix with the onehot vector of job i:

$$E^i = onehot(i-1, n)E$$

(we have i-1 instead of i since workload numbering starts at 1 instead of starting at 0).

with $onehot(x, n) = (0, 0, ..., 1, 0, ...0)$ is a vector of size n with zeros everywhere except at position x.

For example, if p=3 (dimension of embedding vector) and k=6 (number of jobs):

$$E = \begin{pmatrix} 0.45 & 0.33 & -0.43 \\ -0.25 & -0.1 & 0.87 \\ 0.13 & -0.22 & 0.67 \\ 0.24 & .32 & -0.9 \\ 0.73 & -0.3 & -.3 \\ 0.6 & -0.2 & 0.1 \end{pmatrix}$$

Then, $E^2 = onehot(1,6) \cdot E = (0,1,0,0,0,0).\begin{pmatrix} 0.45 & 0.33 & -0.43 \\ -0.25 & -0.1 & 0.87 \\ 0.13 & -0.22 & 0.67 \\ 0.24 & .32 & -0.9 \\ 0.73 & -0.3 & -.3 \\ 0.6 & -0.2 & 0.1 \end{pmatrix} = (-0.25, -0.1, 0.87)$

More details about the maths behind learning embedding layer weights is provided in section 9.4 in appendix.

5.1.2 Experimental setup

The loss function to minimize during the backpropagation is (if we don't use pre-training):

$$\boxed{loss = MSE(f(c_j^i), l_j^i)} \tag{5.1.2.1}$$

However, if we want to leverage the other collected traces O_j^i, we need to pre-train the embedding matrices as well as the weights for the fully connected layers, we need first to minimize this loss function:

$$\boxed{loss' = MSE(f(c_j^i), \phi_j^i)} \tag{5.1.2.2}$$

(with c_j^i the scaled version of C_j^i, and ϕ_j^i is the scaled version of O_j^i).

These experiments are done under the bulk training framework, with online retraining: This means that the regressor has been fitted on all training data from familiar jobs along with the unique sample from unfamiliar job before doing any prediction.

We tried to select the best hyperparameters by doing a 5 fold cross validation (with *early_stopping* set to False according to the recommendations in [4]), and by training on 5000 epochs using Adam's optimizer. Multiple optimization steps are done during each epoch on mini-batches of size 32. The hyper-parameters have been selected from the following possible values using a grid search:

```
activation: {relu, linear}
depth: {2, 3, 4, 5},
dim_embedding: {3, 5, 8, 12, 20}
nh: {20, 50}
pyramid_like: {True, False}
learning_rate: {0.1, 0.01, 0.001}
```

A quick definition of each optimized hyper-parameter is shown below:

- activation: The activation at the output of each layer (except the embedding layer)

- depth: The depth of the neural network (total number of layers excluding the embedding layer).
- dim_embedding: The dimension of the embedding vector (the vector that will code the workload invariant charateristics).
- nh: The number of hidden units (neurons) inside each layer.
- learning_rate: The descent step used by the Adam's optimizer when doing backpropagation.

The best hyper-parameters (that gave the lowest type1 error) are shown in table 24.

activation	*depth*	*dim_embedding*	n_h	*pyramid_like*	*learning_rate*
`relu`	5	3	20	`False`	0.001

Table 24: Selected hyper-parameters for the embedding architecture

We first show the results we obtain without pre-training embedding's weight matrices and biases using the full observations, and later on, we compare results with what we obtain if we pre-train.

We expect from the beginning not to have a better performance when we pre-train using all observations since these observations include metrics that may not be directly linked or associated with latency, or noise perhaps.

We used Keras 1.2.1 running over tensorflow 0.12.1 to implement this neural network architecture.

5.1.3 Results and Analysis (using dataset D_1, no pre-training)

We first show results obtained without enabling *early_stopping*, and then we show the results we obtain by enabling *early_stopping* with patience parameter set to 100 epochs, and threshold set to 1%. (Monitor error on a validation split and stops training if validation error hasn't decreased by more than 1% during the last 100 epochs).

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Number of epochs	5000	5000	5000	5000	5000	5000	-
Training MAPE	9.97%	5.49%	5.61%	6.17%	6.50%	6.71%	6.74%
T1 MAPE	13.04%	6.12%	6.87%	6.98%	7.28%	7.09%	7.90%
T2 MAPE	58.19%	17.70%	33.74%	18.55%	10.77%	19.71%	26.44%

Table 25: Embedding approach (no early stopping)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Number of epochs	523	310	406	309	265	416	-
Training MAPE	11.48%	6.55%	6.45%	7.27%	8.04%	7.84%	7.94%
T1 MAPE	12.08%	6.69%	7.00%	7.65%	8.05%	8.08%	8.26%
T2 MAPE	18.53%	16.96%	22.64%	17.43%	11.32%	13.04%	16.65%

Table 26: Embedding approach results with early stopping enabled

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Job 1	-	3.20%	2.91%	3.24%	3.43%	3.08%	3.17%
Job 2	14.42%	-	13.60%	14.37%	14.81%	15.38%	14.51%
Job 3	11.64%	13.48%	-	14.17%	14.19%	14.27%	13.55%
Job 4	10.38%	11.16%	10.96%	-	11.00%	11.75%	11.05%
Job 7	10.45%	11.53%	10.96%	10.98%	-	13.00%	11.38%
Job 8	11.05%	10.53%	11.29%	10.59%	10.65%	-	10.82%

Table 27: Detailed T1 MAPE for embedding regressor

Observations:

- If we disable early stopping, the embedding regressor overfits and doesn't generalize over unseen jobs, and this is why it yields bad T2 error.
- If early stopping is enabled, approximately 500 epochs is enough for the regressor to learn from the dataset D_1.
- Embedding regressor's performance over familiar (type1 error) and unfamiliar (type2 error) jobs are very similar to those obtained with baseline regressor with onehot-encoding. (We have the same order of magnitude)

5.1.4 Results and Analysis (with and without pre-training)

In the case of pre-training, after we train the architecture using the loss function in 5.1.2.2, we fine tune the weights and embeddings for latency prediction by minimizing the loss in 5.1.2.1, we start from the obtained embedding, weight and biase matrices except those at the top layer, which necessarily will be randomly initialized because of shape change.

In both cases, we repeat the experiments and average the results over 50 runs that differ in the initialization matrices and biases (using different random seeds). We then compare between the results obtained from embedding and those obtained from gradient boosting with onehot encodings.

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	12.27%	6.79%	6.95%	7.46%	7.83%	7.59%	8.15%
T1 MAPE	12.83%	6.89%	7.47%	7.80%	8.01%	7.81%	8.47%
T2 MAPE	20.14%	18.42%	21.05%	15.46%	11.40%	16.09%	17.09%

Table 28: Embedding results over 50 runs (No pretraining) (dataset D_1)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	11.48%	6.47%	6.53%	6.74%	7.13%	7.00%	7.56%
T1 MAPE	12.35%	6.54%	7.13%	7.26%	7.52%	7.37%	8.03%
T2 MAPE	24.30%	17.29%	21.54%	15.62%	11.58%	15.20%	17.59%

Table 29: Embedding results over 50 runs with pretraining (dataset D_1)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	8.79%	4.29%	4.98%	5.00%	5.40%	5.35%	5.64%
T1 MAPE	12.00%	5.74%	6.53%	6.72%	6.97%	6.86%	7.47%
T2 MAPE	24.90%	17.66%	21.98%	12.92%	11.34%	12.22%	16.84%

Table 30: GradientBoosting regression results over 50 runs (onehot encoding, dataset D_1)

Observations:

- For training and type1, Gradient Boosting outperforms the embedding model (with pretraining) which itself slightly outperforms the embedding model (without pretraining). As for type2 errors, we can't see any consistent behavior across the different training settings
- Pre-training doesn't really improve the model. From an optimization point of view, pretraining the weights and embeddings modifies the point from which the minimization of the latency loss (as in 5.1.2.1) starts. In addition, when we pre-train the embedding matrix, we give equal importance, in the pre-training phase, to latency and other metrics that may be just noise.

Euclidean distances between workload embeddings Since the same neural network architecture is used to learn the embeddings for the workloads and to predict the latency, then the obtained embeddings will have some semantics according to the latency. Below we show the euclidean distances between the workload embeddings obtained from all training data in dataset D_2 (without any job left out) sorted in increasing order.

```
[('4_5', 0.26831388),
 ('4_6', 0.4225446),
 ('2_5', 0.49871534),
 ('1_5', 0.57910305),
 ('5_6', 0.60182083),
 ('2_4', 0.60704762),
 ('1_4', 0.73227113),
 ('1_2', 0.93102801),
 ('2_6', 0.94576234),
 ('1_6', 1.0215058),
 ('3_6', 1.139717),
 ('3_5', 1.1606787),
 ('3_4', 1.2609017),
 ('2_3', 1.2691787),
 ('1_3', 1.5471139)]
```

The distance between jobs 4 and 5 is the smallest, which means that these two jobs are very similar in terms of impact on the latency for different configurations. However, the distance between jobs 1 and 3 is the highest, which means these two jobs are very dissimilar. We also notice that in general, the distance between job 3 and any other job is high which confirms once again that job 3 is different from other jobs. The figures 12, 13, and 14 represent the variation of the latency (target) with respect to the configuration shown for: all jobs, job 4 and 5, and job 1 and 3 respectively.

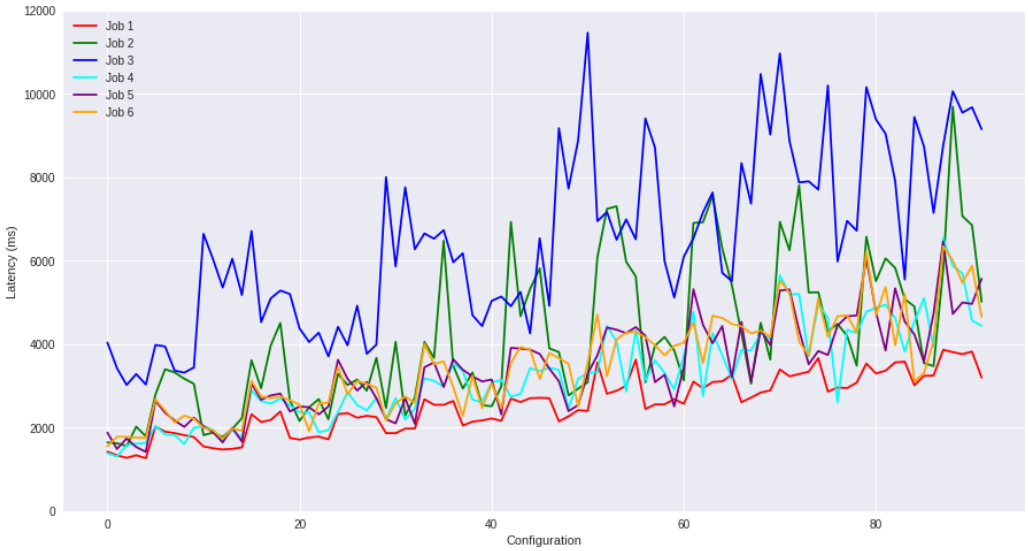

Figure 12: Latencies obtained with all jobs using common configurations

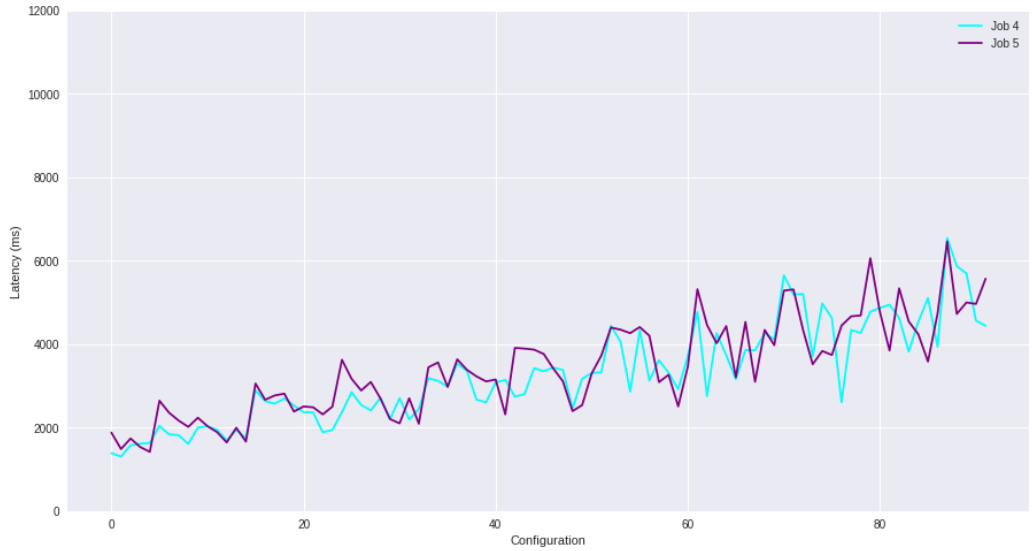

Figure 13: Latencies obtained with jobs 4 and 5 using common configurations

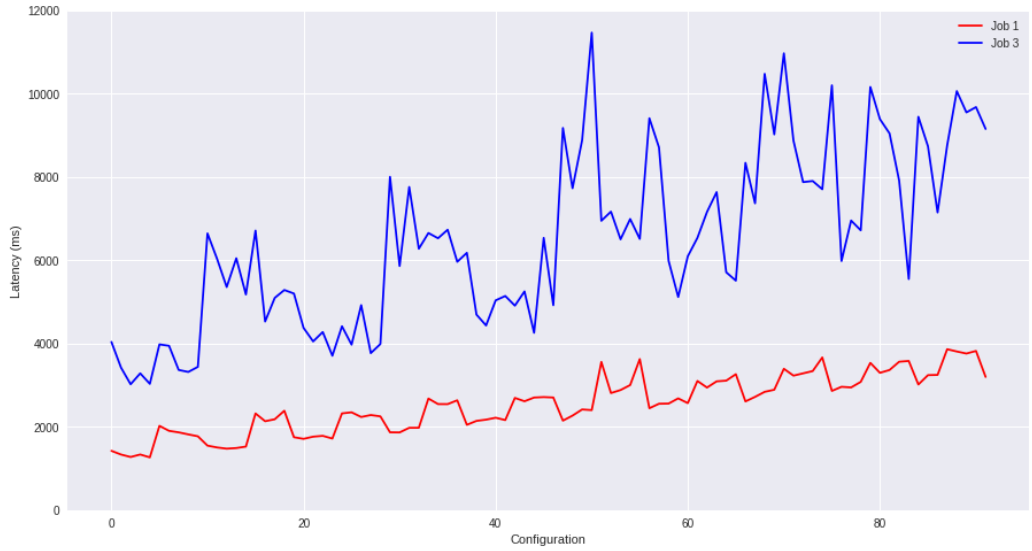

Figure 14: Latencies obtained with jobs 1 and 3 using common configurations

5.1.5 Pros and Cons

Pros

- Coupling between the workload characteristics extraction and the regression task: we have one single architecture that generates the encodings and predict the latency.

- Semantics: Embedding approach gives us some semantic for different workloads by calculating distances/similarities between the different workload vectors.
- Invariance property satisfied: we have a unique workload descriptor per data flow program.
- Incremental online learning is supported.
- Leveraging additional information: we have the ability to pretrain the embedding matrix by using the same embedding architecture we use for predicting the latency, except the top of the network which is replaced to output a vector of the same dimension as the vector of observations.

Cons

- Doesn't support online prediction for new jobs.
- Need of much more data than any approach to better generalize over new jobs.
- Long training time due to the huge space of parameters to be learnt (weights and biases).
- Requires the explicit knowledge of the job id.

5.2 AutoEncoder

5.2.1 Design

The auto-encoder approach consists of two steps: Feature extraction (done using a neural network) and Regression. The feature extraction part is done using a deep learning architecture shown in figure 15 to extract for each job i, its workload characteristics vector E^i, and the regression part is done using a Gradient Boosting regression algorithm that is fitted on $(c_j^i, E^i) \rightarrow l_j^i$

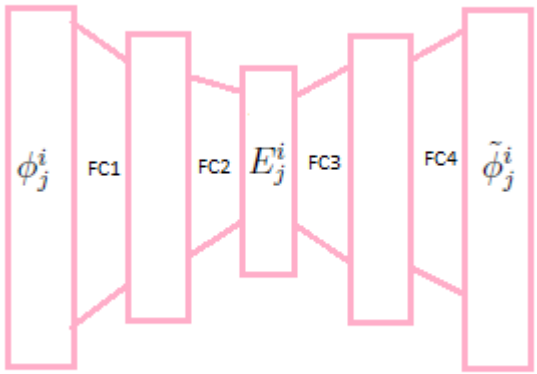

Figure 15: An Auto-Encoder Deep Learning Architecture with depth=2

- Initial input of the NN architecture: $\phi_j^i = MinMaxScale(O_j^i)$
- Intermediate Output: E_j^i (encoding vector)
- Final Output: $\tilde{\phi}_j^i$ (reconstruction of ϕ_j^i)

$$\text{Loss} = \sum_{i \in I_{train}} \sum_{j=1}^{J_i} \frac{1}{|I_{train}|} \frac{1}{J_i} ||\phi_j^i - \tilde{\phi}_j^i||^2 \tag{5.2.1.1}$$

with J_i is the number of configurations for job i, and I_{train} is the set of jobs used in the training setting.

Since an auto-encoder as shown in figure 15 is a non linear dimensionality reduction technique, then for each different observation O_j^i (or ϕ_j^i if scaled) of a particular P^i, we get a different "encoding" E_j^i. And since for the new unfamiliar workload, when we want to predict latency for a configuration C_j^i, we won't have its corresponding encoding E_j^i (it's like the

chicken and egg problem), then we are forced to find a way to characterize each workload by a unique vector E^i. The most simple way to do so, is to take the centroids among all previously observed samples:

$$E^i = \frac{1}{J_i} \sum_{j=1}^{J_i} E^i_j \tag{5.2.1.2}$$

5.2.2 Experimental setup

We explain in this section how did we choose the hyper-parameters of both the auto-encoder neural network architecture and the hyper-parameters of the GradientBoosting regressor. In the encoding part of the architecture, the number of neurones in each layer is divided by two as we move forward. However, in the decoding part, the number of neurones gets multiplied by 2. So, the encoding part has a pyramid-like shape and the decoding part has as well an inverse pyramid-like shape.

We tried to select the best hyperparameters by doing a 5 fold cross validation (with *early_stopping* set to False according to the recommendations in [4]), and by training on 1000 epochs using Adam's optimizer. Multiple optimization steps are done during each epoch on mini-batches of size 32. The hyper-parameters have been selected from the following possible values, such that we minimize the reconstruction error expressed in 5.2.1.1.

```
activation: {relu, linear}
depth: {2, 3, 4, 5}
dim_encoding: {3, 5, 8, 12, 20}
nh: {20, 50, 100}
learning_rate: {0.1, 0.01, 0.001}
```

activation	*depth*	*dim_encoding*	n_h	*learning_rate*
`relu`	4	3	100	0.001

Table 31: Selected hyper-parameters for the auto-encoder architecture

Note that the hyper-parameter *depth* in the table 31 is tricky here: It represents the depth of the encoding part (or decoding part) and not the full depth of the whole architecture. We kept the same hyper-parameters for the GradientBoosting regressor that we obtained by CV in baseline experiments. These hyper-parameters are shown again in the table 32.

learning_rate	*loss*	*max_depth*	*max_features*	*n_estimators*
0.1	`lad`	4	5	500

Table 32: Best hyper-parameters for GradientBoosting regressor

We used tensorflow 0.12.1 to implement the auto-encoder neural network architecture. This experiment is done under 2 frameworks which mainly differ when it comes to type 2 error (However, for training and type1 errors, we won't see any difference):

- Bulk training framework (also called online retraining): where both the auto-encoder and the Gradient Boosting regressor are trained on all the training data from familiar jobs in addition to the unique sample from the unfamiliar job.
- Online prediction framework: which means that we use the auto-encoder and Gradient Boosting regressor already trained on samples from the familiar jobs, to immediately predict latency corresponding to new configurations from the unfamiliar job, after the encoding for that new job is obtained from the seen traces.

We show the results obtained with the two datasets D_1 and D_2.

5.2.3 Results and Analysis using dataset D_1 (all configurations)

The results in the tables represent what we obtained with the auto-encoder approach by training the gradient boosting regressor on centroids of encodings of each job (obtained from the auto-encoder). The error values shown are obtained by repeating the experiments 50 times (for different random seeds for the auto-encoder), and by taking the average score in each training settings.

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	8.44%	4.21%	4.93%	4.91%	5.33%	5.26%	5.51%
T1 MAPE	12.05%	5.66%	6.51%	6.73%	6.92%	6.81%	7.45%
T2 MAPE	30.47%	23.06%	46.64%	16.36%	12.64%	18.45%	24.60%

Table 33: Auto-Encoder regression results over 50 runs (online prediction framework on D_1)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	9.00%	4.33%	5.32%	5.18%	5.34%	5.63%	5.80%
T1 MAPE	11.94%	5.70%	6.62%	6.78%	6.93%	6.92%	7.48%
T2 MAPE	22.92%	22.13%	24.28%	15.42%	14.90%	16.80%	19.41%

Table 34: Auto-Encoder regression results over 50 runs (online retraining framework on D_1)

Observations:

- By analyzing table 33, we can see that the new job is drastically different from the workloads we've seen, and this is the case in LO_2 and LO_3 training settings. However,

in LO_4, LO_5 and LO_6, the regressor has already seen a similar workload. (Since workload 4 and 5 are similar, and workload 5 and 6 are exactly the same with some difference in IO operations). As for LO_1, since the workload 1 has a sliding window whose size depend of φ, then it's quite different from all the other workloads, and this is why T2 MAPE is high if we directly try to predict over this job (table 33).

- By comparing results in table 33 with results obtained in table 34 we come up with the conclusion that the regressor we're using (gradient boosting) is a robust regressor. From one single observed sample on which it's trained, it improves drastically the overall T2 performance.

5.2.4 Results and Analysis using dataset D_2 (common configurations)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	6.33%	5.25%	5.83%	5.71%	5.70%	5.65%	5.74%
T1 MAPE	12.67%	10.43%	10.81%	9.80%	10.33%	10.72%	10.79%
T2 MAPE	51.43%	30.07%	45.59%	19.50%	20.30%	21.03%	31.32%

Table 35: Auto-Encoder regression results over 50 runs (online prediction framework on D_2)

Training Settings	LO_1	LO_2	LO_3	LO_4	LO_5	LO_6	AVG
Training MAPE	7.01%	6.24%	6.64%	6.33%	6.33%	6.51%	6.51%
T1 MAPE	12.31%	9.85%	10.26%	9.44%	9.85%	10.64%	10.39%
T2 MAPE	25.41%	23.53%	36.06%	16.01%	16.82%	15.45%	22.21%

Table 36: Auto-Encoder regression results over 50 runs (online retraining framework on D_2)

Observations:

- By comparing results in table 33 and 35, we can notice that having more data improves a little bit Type1 MAPE, but improves significantly Type2 MAPE in the online prediction framework.
- Deep Learning needs a lot of data to give good performance.

5.2.5 Pros and Cons

Pros

- Ability to predict without retraining: if the auto-encoder is already trained, whenever a new job comes, we can immediately extract its encoding from the current observations

without having to know explicitly the job id. Then we use the extracted encoding along with the configuration to predict immediately the latency.

- Fair generalization power compared with the other approaches, and this generalization power can be even improved by collecting more traces

Cons

- Difficulty of encoding quality assesment: As any unsupervised task, it's difficult to tell whether the current encoding is a good one or not, especially that the encoder is decoupled from the regression on the final target (latency). Thus, the obtained encodings (by averaging) won't contain semantic information regarding to latency as the encodings obtained from the embedding approach.
- Violation of invariance property: by construction, the encodings are obtained from observations which depend highly from the configuration of the current job. Thus, the encodings themselves depend of the configuration and are not invariant with respect to the job id.
- Information loss: we lose a huge amount of information certainly after averaging the encodings with respect to each job. This information loss comes from two reasons:
  - Information loss due to averaging (in datasets D_1 and D_2).
  - Information loss due to the lack of homogeneity in the configurations being averaged among each job (in dataset D_1).

6 Qualitative and Quantitative Comparision

6.1 Qualitative comparision

To wrap things up, we provide the major differences between the approaches we've studied in this master thesis: Baseline approach (using GradientBoosting Regressor with onehot encoding of job ids), the Embedding approach and the Auto-Encoder approach.

	Baseline (1hot, GB)	Embedding	AutoEncoder
Workload descriptors	No	Yes+	Yes-
Amount of info used	$(id, C_j^i) \rightarrow l_j^i$	• pretrain $(id, C_j^i) \rightarrow (O_j^i)$ • train $(id, C_j^i) \rightarrow l_j^i$ • prediction $(id, C_j^i) \rightarrow l_j^i$	• $O_j^i \rightarrow E_j^i \rightarrow O_j^i$ • $\{E_j^i, j \in J\} \rightarrow E^i$ • $(C_j^i, E^i) \rightarrow l_j^i$
Scalability	$O(Cn^2)$ (space complexity). For every job, we add a column, and for every configuration we add a row	$O(Cn)$	$O(Cn)$
Training cost for every new job	for every single job added, needs global retraining	For every job added incremental or global retraining is needed	Not necessarily
Prediction over old jobs	High	High	High
Prediction over new jobs	Medium	Medium	Medium+ ? (Find a same default config)

Table 37: Major differences between different regression approaches

In the big O notation of space complexity of the different approaches, C denotes the average number of configurations per job, and n denotes the total number of jobs. Note: This space complexity denotes the space complexity regarding to the training data, and not regarding to the regression model. (In DL, we have to store as well the weights and biases, and in GB we have to store the different trees and their nodes)

We should highlight that the baseline approach is impractical because of:

- The need to retrain the regressor to predict over new jobs.
- The problem of scalability: at each time we add a new job, our training matrix introduces a new column as well as the new rows corresponding to configurations of that job.

We also have to keep in mind that the embedding approach is superior to the auto-encoder approach in one thing: the coupling between the feature extraction and the regression on

the latency, and this is what gives semantics to the obtained workload characteristics vectors with the embedding approach. However, the auto-encoder remains the only approach that provides an online prediction framework.

6.2 Quantitative comparision

A summary of the empirical results obtained with the different regression approaches is provided in Table 38

	Baseline	Embedding(retraining)	AE (retraining)	AE (online prediction)
Time cost	3 s	7min 48s	8 mins 22s	0.6 s
Training MAPE	5.64 %	7.56 %	5.81%	5.51%
T1 MAPE	7.47%	8.03%	7.48%	7.45%
T2 MAPE	16.84 %	17.59%	19.41%	24.6%

Table 38: Average MAPE for different approaches using dataset D_1

The time cost row of this table shows the amount of time required before getting the predicted values over new jobs. In the case of baseline, embedding and AE (online retraining), this time includes the time to train the regressor (and/or the neural network). However, in AE (online prediction), the trained neural network and trained regressor can be used to get the predictions without the need to retrain.

From this table, we can notice that:

- Auto-encoder comes with a negligible cost in the online prediction framework, while baseline have modest cost because it requires retraining.
- With the current datasets, the cost of training a neural network remains higher than the cost of training a gradient boosting regressor.
- The power of prediction over existing jobs is high over all techniques (by observing T1 MAPE).
- We have some real issues when we try to predict over new jobs. Deep Learning approaches gave similar or poorer results as the baseline in terms of type2 error. We suppose that this may be due to several reasons:
  - Perhaps we need more data to get a better performance in training the neural network in the embedding approach.
  - The collected traces used in the auto-encoder don't contain useful information that help predicting the latency.
  - The collected traces used in the auto-encoder contain useful information for learning the latency, but our auto-encoder is not able to distill these information.

7 Related Work

This section goes through related work in deep learning and/or performance modeling:

7.1 Contractive Auto-Encoder

The auto-encoder do not solve the problem of having invariant encodings per job because:

- It's a dimensionality reduction technique: if input data, corresponding to different configurations of one job, have a high variance, then we expect to see this variance in the obtained representation (encoding) too.
- The loss function we're minimizing is the mean squared error between the input (observations vector) and reconstruction obtained from the encoding. In order to have a small value of the loss function, the observations need to be perfectly reconstructed, and thus the variance in the input will necessarily induce a variance in the encoding.

The contractive auto-encoder is a variation of the auto-encoder that attempts to let the extracted features from the input more robust by making them locally invariant in many directions of change of the raw input. The contractive auto-encoder also doesn't solve our problem in obtaining invariant encodings with respect to different configurations of one particular job.

Despite the fact that the obtained representation better captures the local directions of variation dictated by the data, corresponding to a lower-dimensional non-linear manifold, while being more invariant to the vast majority of directions orthogonal to the manifold, the obtained encodings will still have an variance in the directions of the manifold.

7.2 Traditional Query Optimizer

In DataBase Management Systems (DBMS), traditional query optimizer tries to determine the most efficient way to execute a given query by considering possible execution plans according to the running time. The query optimizer is a single objective optimizer that focuses on the query's running time by doing IO and CPU analysis. A query execution plan is characterized by how data will be collected from database, using which data-structures and by which order they will be accessed.

In our work, we can't build such models because we don't have fixed operators for dataflow systems. Furthermore, we have to learn from observations coming from the same environment to extract workload characteristics. Moreover, in our work we're using a multi-objective optimization framework: we don't focus only on running time, but we also consider monetary cost and throughput.

7.3 Ottertune

The paper [16] proposes a way of tuning configuration parameters of a DBMS. Tuning DBMS differs from tuning workflow systems (like Spark) by:

- Queries written in DBMS are based on small algebra, however in workflow models, we can run arbitrary analytics and user-defined functions are wrapped in a dataflow program.
- Existing workload descriptors have been developed by each vendor of a DBMS system. A workload is characterised using the runtime statistics recorded while executing it. For example, MySQL's InnoDB engine provides statistics on the number of pages read/written, query cache utilization, and locking overhead. So the feature engineering problem for workload description is solved based on vendor specific information over which dimensionality reduction to remove redundancy and correlation is done. Then clustering is applied to create meaningful groups and finally a Lasso regression for feature selection.

  However, in our proposed models for tuning workflow systems, feature engineering is done simultaneously with feature selection for high-quality prediction. In addition, the traces we're collecting consist of metrics regarding the workflow system (Spark streaming) plus some system metrics. Hence, we can't treat deep learning as a blackbox, but a framework within which we have to determine feature engineering and selection for performance modeling and prediction.
- OtterTune performs in the space of single-objective optimization: The DBA tells the tuning manager what metric (throughput or latency) to optimize when selecting a configuration. Given a specific objective, it performs two steps:
  - find the most similar workload based on Euclidean distance, in a dynamic mapping manner
  - recommend a better configuration in each execution period to improve the target metric.

  Our work performs in the space of multi-objective optimization, and provides :
  - the full skyline to maximize the information to the user or
  - a set of methods to find the most suitable points on the skyline for a user population.

  However, a limitation of our current work is that the optimizer fully trusts the model prediction and makes optimization decisions based on the predicted performance.

8 Conclusion and Future Work

In this research project we aimed to answer questions regarding what additional benefits deep learning offers compared with the baseline regressor. Indeed, we showed that deep learning offers an incremental prediction framework (using embedding architecture) or online prediction framework (using auto-encoder along with a gradient boosting regressor) that are not available in the baseline approach. But there is a tradeoff between using the online prediction framework and having good performance, since of course retraining improves results. That said, the online prediction framework gave us acceptable generalization power over unseen jobs.

We also wanted in our research to answer the question about if it's possible to learn meaningful workload descriptors. We showed in fact that in the embedding approach where workload extraction and latency regression are coupled, we could get workload characteristics vectors that have some semantics and achieve fair accuracy. As for the auto-encoder, we believe that the way we're extracting the workload descriptors needs to be reconsidered. We suppose that the reason behind the poor encoding comes from taking centroids of encodings per job id, in order to satisfy the invariance property. This is why we think that the auto-encoder wasn't able to show its generalization power because the way we collected data was not right (having a small amount of workloads with too many configurations).

Therefore, we intend in our future work to generate a lot of workloads and collect few traces per workload. By having for example 800 new workloads with a default configuration, then we can train an auto-encoder on that common configuration to get an invariant encoding without having to use the centroid of different encoding vectors. But choosing the default configuration is yet another research issue to address.

9 Appendix

9.1 A review on cross validation

The cross validation is a technique used in machine learning to select the hyperparameters of an estimator. A V-fold cross validation consists of splitting data into V folds and at each time select one fold to keep out of the data, and train on the remaining folds. A test error is calculated on the fold that has been left out.

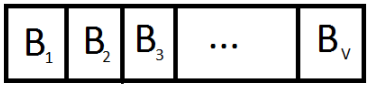

Figure 16: Cross validation folds

Generally $V = 5$ or 10

If $V = n_{train}$, then we call this particular case "Leave one out". The greater the V is, the better estimation of the generalization error we obtain, but the longer time it takes to approximate this generalization error.

Let's denote by $\hat{g}_h^{(-l)}$ the estimator (or regression function) that was trained using hyperparameters h on configurations coming from $B_i, i \in \{1, 2, ...V\} \setminus \{l\}$ to predict the latency. If $L^{(l)}$ represents the type1 error evaluated on fold B_l, then the cross validation error estimate for the current hyperparameters setup h is given by:

$$CVEE(h) = \frac{1}{V} \sum_{k=1}^{V} L^{(k)}(\hat{g}_h^{(-k)})$$

Finally, for different setups of the hyperparameters $h \in h_1, h_2, ...h_H$, (with H the number of possible combinations of the hyperparameters) we select $\hat{h}$ (which is a particular setup of the hyperparameters, on which the CVEE was calculated) that gives the lowest CVEE:

$$\hat{h} = argmin_h CVEE(h) = argmin_h \frac{1}{V} \sum_{k=1}^{V} L^{(k)}(\hat{g}_h^{(-k)})$$

9.2 A review on some regression algorithms and their hyper-parameters

Most of the content of this section is extracted from scikit-learn's website with some slight modifications.

- **K-Nearest Neighbors (KNN)** regressor: The k-nearest neighbors algorithm is a non-parametric method that can be used for regression. In KNN regression, the target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set. KNN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until regression. The KNN algorithm is among the simplest of all machine learning algorithms. The tuned parameters are:
  - *n_neighbors*:Number of neighbors from which the target will be predicted. This parameter was chosen from $\{3 + 2i, i \in \{1, 2, ...20\}\}$ (all odd numbers between 3 and 43)
  - *weights*:weight function used in prediction. Possible values: `uniform` or `distance`. If `uniform` is chosen, then all points in each neighborhood are weighted equally. If `distance` is chosen, then points are weighted by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away. During cross validation, this parameter was chosen from {`uniform`, `distance`}
- **SVM** regressor [15]: In machine learning, support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. In regression the method is called support vector regression (SVR). The model produced by support vector classification depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by SVR depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction. The important hyper-parameters that were tuned:
  - *kernel*: specifies the kernel type to be used in the algorithm. It must be one of `linear`, `poly`, `rbf`, `sigmoid`, `precomputed`. During cross validation, the kernel was chosen from {`rbf`, `linear`}
  - *C*: Penalty parameter of the error term. The C parameter tells the SVM optimization how much to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points. For very tiny values of C, you should get misclassified examples, often even if your training data is linearly separable. (when the kernel is linear). C was chosen from {1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100, 1000, 1e4, 1e5, 1e6}
  - *epsilon*: in the epsilon-SVR model: It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.
- **Bagging**: A Bagging regressor is an ensemble meta-estimator that fits base regressors each on random subsets of the original dataset and then aggregate their individual

predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. The base estimator we used in our experiments was a decision tree, and the two main important hyper-parameters that we tuned are:

  - *n_estimators*: The number of base estimators in the ensemble. It was chosen from {50, 200, 500, 1000}
  - *max_features*: The number of features to draw from X (training matrix) to train each base estimator. It was chosen from {1, 2, ..., 9, 10} (since cross validation was done using data with onehot encoding of job id)

- **Random Forest** [6]: A random forest is a meta estimator that fits a number of decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement. The important hyper-parameters that we tuned:

  - *n_estimators*: The number of trees in the forest. It's chosen from {50, 200, 500, 1000}
  - *max_features*: The number of features to consider when looking for the best split. max_features was chosen from {1, 2, ..., 9, 10}
  - *max_depth*: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. (min_samples_split is by default=2). Max_depth is chosen from {None, 5, 10, 15, 20, 25, 30, 35, 40, 45}

- **Extra-trees**: It's a variant of the random forest regressor that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the randomly selected features and the best split among those is chosen. We also tuned the hyperparameters: *n_estimators*, *max_features*, *max_depth* for this regressor from the same spaces of possible values.

- **GradientBoosting (GB)** [7]: GB builds an additive model in a forward stage-wise fashion. It allows for the optimization of arbitrary differentiable loss functions. This algorithm is an iterative algorithm. In each stage a regression tree is fit on the negative gradient of the given loss function or what we call pseudo residuals so that it greatly improve score on examples it was currently not predicting so well, without messing up the other examples too much. Gradient Tree Boosting or Gradient Boosted Regression Trees (GBRT) can be seen as a generalization of boosting to arbitrary differentiable loss functions. The advantages of GBRT are:

  - Natural handling of data of mixed type (= heterogeneous features)

- Predictive power
- Robustness to outliers in output space (via robust loss functions)

The main disadvantage of GBRT is scalability: due to the sequential nature of boosting it can hardly be parallelized. The important hyper-parameters that we tuned by a grid search cross validation procedure are:

- *n_estimators*: The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance. This parameter was chosen from {50, 200, 500, 1000}
- *max_features*: The number of features to consider when looking for the best split. If None, then max_features=n_features. This parameter was chosen from {1,2,3, ... 9, 10}
- *max_depth*: maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. This parameter has an important impact on the performance and must be tuned for best performance; the best value depends on the interaction of the input variables and was chosen from 3, 4, 5, 10
- *loss*: `ls`, `lad`, or `huber`. `ls` refers to least squares regression. `lad` (least absolute deviation, equivalent to the least absolute error) is a highly robust loss function solely based on order information of the input variables. `huber` is a combination of the two. This parameter was chosen from {`ls`, `lad`, or `huber`}
- *learning_rate*: learning rate shrinks the contribution of each tree by learning_rate. There is a trade-off between *learning_rate* and *n_estimators*. This parameter was chosen from {0.001, 0.01, 0.1}

Let's highlight now on the main differences between the ensemble regressors. This is illustrated in table 39. Since we have sampling in RandomForest regressor, Bagging and ExtraTrees, then we expect not to have a good type2 error for these algorithms if we compare with respect to GradientBoosting.

Random Forest	**Extra Trees**	**GradientBoosting**	**Bagging (Tree based)**
Samples are drawn with replacement, and size of subsample is = size of full training sample	Samples are drawn with replacement, and size of subsample is = size of full training sample	No subsampling is done, and all data are used for building each tree. (subsampling is done in stochastic gradient boosting)	Samples are drawn with replacement, and size of subsample is by default = size of full training sample, but can be changed though
For each split inside a tree, a random subset of features is used	For each split inside a tree, random splits are drawn for each of the randomly drawn features and the best split among those is chosen	For each split inside a tree, a random subset of features is used	For each tree, a random subset of features will be used
Building different trees can be parallelized	Building different trees can be parallelized	The algorithm is sequential, and it can't build different trees in a parallel manner	Building different trees can be parallelized

Table 39: Major differences between ensemble regression algorithms

9.3 A review on Deep Learning [4]

This chapter is a summary of the practical recommendations on training neural network given in the paper [4].

- A neural network can be represented by a graph with 3 types of nodes:
  - Input nodes (no computation)
  - Internal nodes
  - Output nodes

  Nodes correspond to elementary operations (addition, multiplication, and non linear operations such as neural network activation functions). The flow graph is directed and acyclic. Each of its nodes is associated with numerical output which is the result of application of the computation.

  In addition of associating a numerical output o_a to each node a of the graph, we can associate a gradient: $g_a = \frac{\partial L(z,\theta)}{\partial o_a}$
  - o_a is computed using predecessor's output o_p of predecessor nodes.
  - g_a is computed using the gradients of the successor node s of a

Chain rule: $g_a = \sum_s g_s \frac{\partial o_s}{\partial o_a}$

- The neural network learning mechanism relies on the backpropagation principle which uses the chain rule to compute the gradients in a backward manner.
- Gradient can be computed manually or through automatic differentiation.
- We have 2 types of hyperparameters:
  - Hyperparameters of the optimization
  - Hyperparameters of the model itself

9.3.1 Tuning hyperparameters of the optimization

We'll only list the hyper-parameters of the optimization that we tuned in our deep learning models.

- **Learning rate** ϵ_0:

  recommendation: $10^{-6} < \epsilon_0 < 1$ to be used with inputs mapped to $(0, 1)$ interval. Default is 0.01. Learning rate value is crucial. If we have the time to tune only 1 hyperparameter, then this is the hyperparameter to tune. In practice, we start with a large value and if the training criterion diverges, we try again with 3 times smaller learning rate, and so on...until no divergence is observed. Note that we didn't deal ourselves with any learning rate schedule. It's the optimizer we choose (Adam Optimizer) that takes care of the learning rate schedule.

- **Minibatch size B**

  B = 32 is a good default value, and we used this value in both deep learning approaches (embedding and auto-encoder). Larger B implies faster computation (with appropriate implementation), but requires visiting more examples in order to reach the same error (since there are less updates per epoch)

  $$\theta^{(t)} = \theta^{(t-1)} - \epsilon_t \frac{1}{B} \sum_{t'=Bt+1}^{B(t+1)} \frac{\partial L(z_t, \theta)}{\partial \theta}$$

  B = 1 corresponds to to online gradient descent; Note that online gradient is an unbiased estimator of the generalization error gradient B = training set size corresponds to standard (batch) gradient descent

  When B increases, we get more multiply-add operations per second, if we take advantage of parallelism, or matrix-matrix multiplications (instead of separate matrix-vector multiplications), often gaining a factor of 2 in overall training time. But, when B increases, the number of updates per computation done decreases which slows down convergence (in terms of error vs number of multiply-add operations performed). This parameter impacts training time and not so much test performance, and this is why we didn't tune it.

- **Number of training iterations T** (measured in minibatch updates)

  This parameter is optimized using the principle of early stopping. It helps avoid overfitting (that may also be due to other hyperparameters).

  Note that it might be useful to turn early stopping off when analyzing the effect of individual hyperparameters.

  Practical implementation:

  - Save $\hat{T}$ for which the validation error (on an out of sample dataset) was the minimum and such that for $T < \hat{T} + \tau$ (or for $T < \hat{T} * \tau$) the validation error hasn't decreased by more than a certain threshold. τ here represents a constant related to patience
  - To avoid the overhead of early-stopping, one may calculate the validation error not after each update, but after seeing N examples (or a multiple of N) with N as large as the validation set size.

  Some definitions: 1 epoch = 1 iteration through the whole training set. Faster convergence has been observed if the order in which the minibatches are visited is changed for each epoch. (Efficient if data training holds in memory)

9.3.2 Tuning hyperparameters of the model and training criterion

- **number of hidden units**: n_h

  - n_h= size of the neural network layer
  - if n_h is larger than the optimal value, this does not hurt generalization performance much, but will induce more computations ($O(n_h^2)$) in a FC architecture)
  - We can set different values of n_h for different layers
  - In general, over complete layers work better than undercomplete layers, and using the same size for all layers generally worked better than pyramid like (or upside pyramid) sizes.

- **Neuron non linearity**

  Typical neuron output $s(a) = s(w'x + b)$, with:

  - x: vector of inputs into the neuron.
  - w: vector of weights
  - b: offset (bias term)
  - s: scalar non linear function

  Most commonly used non linearities for hidden units:

  - Rectifier max (a.k.a relu) : $max(0, a)$

  - Sigmoid: $\frac{1}{1+\exp(-a)}$
  - hyperbolic tangent: $\frac{\exp(-a)-\exp(a)}{\exp(-a)+exp(a)}$

  For output (or reconstruction units), hard neuron non linearities like the rectifier do not make sense, because when the unit is saturated, (e.g. $a < 0$ for the rectifier), and associated with a loss, no gradient is propagated inside the network, i.e. there's no chance to correct the error. Although, we've used the `relu` activation, we need to pay attention of the saturation at the outputs.

- **Weights initialization scaling coefficient** Biases can be generally initialized to 0, but weights need to be initialized carefully to break the symmetry between hidden units of the same layer. Recommendation: sample weights from a uniform $(-r, r)$ distribution where:
  * $r = \sqrt{\frac{6}{\text{fan-in+fan-out}}}$ for hyperbolic tangent units. This is also called Glorot uniform initialization or Xavier uniform initialization.
  * $r = 4\sqrt{\frac{6}{\text{fan-in+fan-out}}}$ for sigmoid units
- **Random seeds** Sources of randomness:
  - random initialization
  - sampling examples

  Choice of random seed has a slight effect on results: but since we have enough computing power, then we tried to get results using different seeds.
- **Preprocessing**
  - Rescale the inputs of the neural network using a standard scaler or a MinMax scaler (to have them mapped in $[0, 1]$
  - Non linearity: logarithm or square root.

9.4 Maths behind learning embedding vectors E^i

This section is intended for two purposes:

- To show the expression of the update that is done on embedding matrix after a training step using SGD.
- To prove that if a sample corresponds to job i, then this sample will only affect the embedding vector E^i relative to job i, and won't affect $E^{i'}$ with $i' \neq i$

To simplify things, we'll do the computation on the simple neural architecture shown in figure 17.

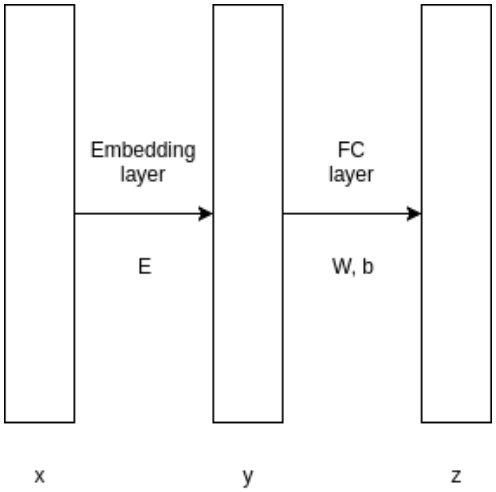

Figure 17: Simplified embedding architecture

- $x = (x_1, x_2, ..., x_n)$ is the input vector of the neural network of shape $(n, 1)$. It represents the onehot encoding of the job id. (A vector with zeros everywhere except at one place)
- $y = (y_1, y_2, ..., y_p)$ is the output vector of the embedding layer. This vector is of shape $(p, 1)$.
- $z = (z_1, z_2, ..., z_d)$ is the output vector of this architecture. It is of shape $(d, 1)$.
- E is the embedding matrix of shape (n, p). (It's the weight matrix of the embedding layer). $(E = [e_{ij}])$
- W is the weight matrix of the fully-connected layer and is of shape (p, d). $(W = [w_{ij}])$
- $b = (b_1, b_2, ..., b_d)$ is the bias matrix of the fully connected layer and is of shape $(d, 1)$

$$E = \begin{pmatrix} e_{11} & e_{12} & ... & e_{1p} \\ e_{21} & e_{22} & ... & e_{2p} \\ ... & ... & ... & ... \\ e_{n1} & e_{n2} & ... & e_{np} \end{pmatrix} W = \begin{pmatrix} w_{11} & w_{12} & ... & w_{1d} \\ w_{21} & w_{22} & ... & w_{2d} \\ ... & ... & ... & ... \\ w_{p1} & w_{p2} & ... & w_{pd} \end{pmatrix}$$

$E^i = (e_{i1}, e_{i2}, ..., e_{ip})$ represents the ith row of the matrix E and corresponds to the embedding vector of job i.

The loss function to be minimized during back-propagation is $L(z)$ (scalar function). During back-propagation, the weights of the neural network architecture are updated in SGD by using gradients of the loss function as follow (with η being the learning rate)

$$W_{(t+1)} = W_{(t)} - \eta \nabla_W L(z)$$

$$E_{(t+1)} = E_{(t)} - \eta \nabla_E L(z)$$

This last equation can be expanded and written as n equations:

$$E^i_{(t+1)} = E^i_{(t)} - \eta \nabla_{E^i} L(z) \quad \forall i \in \{1, 2, ..., n\}$$

We will prove that if the sample over which the stochastic gradient descent step will be applied corresponds to job i, then only $\nabla_{E^i} L(z) \neq 0$ (in general) and $\forall i' \neq i, \nabla_{E^{i'}} L(z) = 0$ and thus no change will affect embedding vectors of other jobs.

If we ignore the activations, we can write:

- $y = E^T \cdot x \quad (y_i = \sum_j e_{ji} \cdot x_j)$
- $z = W^T \cdot y + b \quad (z_i = \sum_j w_{ji} \cdot y_j + b_i)$
- $\nabla_z L = (\frac{\partial L}{\partial z_1}, \frac{\partial L}{\partial z_2}, ..., \frac{\partial L}{\partial z_d})$
- $\nabla_y L = (\frac{\partial L}{\partial y_1}, \frac{\partial L}{\partial y_2}, ..., \frac{\partial L}{\partial y_p})$
- $\nabla_{E^i} L = (\frac{\partial L}{\partial e_{i1}}, \frac{\partial L}{\partial e_{i2}}, ..., \frac{\partial L}{\partial e_{ip}})$

By using the chain rule, we can write:

$$\frac{\partial L}{\partial y_i} = \sum_{j=1}^{d} \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial y_i} = \sum_{j=1}^{d} \frac{\partial L}{\partial z_j} w_{ij}$$

By using the chain rule too, we can also write:

$$\frac{\partial L}{\partial e_{ij}} = \sum_{k=1}^{p} \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial e_{ij}}$$

But if $k \neq j$, then $\frac{\partial y_k}{\partial e_{ij}} = 0$. So,

$$\frac{\partial L}{\partial e_{ij}} = \frac{\partial L}{\partial y_j} \cdot \frac{\partial y_j}{\partial e_{ij}} = x_i \cdot \frac{\partial L}{\partial y_j}$$

Thus:

$$\nabla_{E^i} L = x_i \cdot \nabla_y L$$

This means that if the sample over which the update will be done corresponds to job i, then $x_i = 1$ and $\forall i' \neq i, x_{i'} = 0$, so:

$$\nabla_{E^i} L = \nabla_y L$$

and $\forall i' \neq i, \nabla_{E^{i'}} L = 0$

9.5 Spark Streaming [1]

This section is an excerpt from the Spark Streaming programming guide [1] with some slight modifications that take into consideration our notation.

9.5.1 Overview

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, one can apply Spark's machine learning and graph processing algorithms on data streams.

Figure 18: Spark Streaming: Input sources and output file systems

Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Figure 19: Spark streaming using Spark engine to process data

Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input

data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs. (RDD stands for Resilient Distributed Dataset and is the basic data structure used by the Spark Engine)

9.5.2 A Quick Example

Let's say we want to count the number of words in text data received from a data server listening on a TCP socket. Consider the following code (in Python)

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

The lines colored in blue correspond to the textual description of the dataflow program that consists of 3 operations: a `flatMap` followed by a `Map` and a `reduceByKey`.

9.5.3 Window Operations

Spark Streaming also provides windowed computations, which allow to apply transformations over a sliding window of data. The figure 20 illustrates this sliding window.

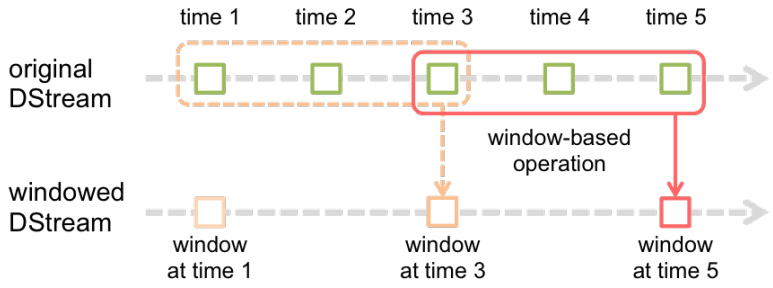

Figure 20: Windowed operations in Spark Streaming

As shown in this figure, every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In this specific case, the operation is applied over the last 3 time units of data, and slides by 2 time units. This shows that any window operation needs to specify two parameters.

- window length: The duration of the window (3 in the figure).
- sliding interval: The interval at which the window operation is performed (2 in the figure).

These two parameters must be multiples of the batch interval of the source DStream ($\varphi = 1$ in the figure).

9.5.4 Performance tuning

Getting the best performance out of a Spark Streaming application on a cluster requires tuning some parameters. Those are the parameters we aim to automatically tune using our multi-objective optimizer.

This section explains a number of the parameters and configurations that can be tuned to improve the performance of an application. Some low level parameters related to memory tuning and garbage collector are out of the scope of our research project. At a high level, two things need to be considered:

1 Reducing the processing time of each batch of data by efficiently using cluster resources.

2 Setting the right batch size (***batch interval*** φ) such that the batches of data can be processed as fast as they are received (that is, data processing keeps up with the data ingestion).

Reducing the Batch Processing Times

There are a number of optimizations that can be done in Spark to minimize the processing time of each batch:

- *Level of Parallelism in Data Receiving*:

  Receiving data over the network (like Kafka, Flume, socket, etc.) requires the data to be deserialized and stored in Spark. If the data receiving becomes a bottleneck in the system, then one should consider parallelizing the data receiving. Note that each input DStream creates a single receiver (running on a worker machine) that receives a single stream of data. Receiving multiple data streams can therefore be achieved by creating multiple input DStreams and configuring them to receive different partitions of the data stream from the source(s). For example, a single Kafka input DStream receiving two topics of data can be split into two Kafka input streams, each receiving only one topic. This would run two receivers, allowing data to be received in parallel, thus increasing overall throughput. These multiple DStreams can be unioned together to create a single DStream. Then the transformations that were being applied on a single input DStream can be applied on the unified stream.

  Another parameter that should be considered is the receiver's ***block interval*** ρ. For most receivers, the received data is coalesced together into blocks of data before storing inside Spark's memory. The number of blocks in each batch determines the number of tasks that will be used to process the received data in a map-like transformation. The number of tasks per receiver per batch will be approximately (φ/ρ). For example, block interval of 200 ms will create 10 tasks per 2 second batches. If the number of tasks is too low (that is, less than the number of cores per machine), then it will be inefficient as all available cores will not be used to process the data. To increase the number of tasks for a given batch interval, one must reduce the block interval. However, the recommended minimum value of block interval is about 50 ms, below which the task launching overheads may be a problem.

- *Level of Parallelism in Data Processing*:

  Cluster resources can be under-utilized if the number of parallel tasks used in any stage of the computation is not high enough. For example, for distributed reduce operations like `reduceByKey` and `reduceByKeyAndWindow`, the default number of parallel tasks is controlled by the `spark.default.parallelism` configuration property. One can pass the level of ***parallelism*** θ as an argument or set the `spark.default.parallelism` configuration property to change the default.

Setting the Right Batch Interval φ

For a Spark Streaming application running on a cluster to be stable, the system should be able to process data as fast as it is being received. In other words, batches of data should be processed as fast as they are being generated.

Depending on the nature of the streaming computation, the batch interval φ used may have significant impact on the data rates λ that can be sustained by the application on a fixed set of cluster resources. For example, let us consider the earlier WordCountNetwork example. For a particular data rate λ, the system may be able to keep up with reporting word counts every 2 seconds (i.e., $\varphi = 2$ seconds), but not every 500 milliseconds. So the batch interval needs to be set such that the expected data rate in production can be sustained.

A good approach to figure out the right batch size for a particular dataflow program is to test it with a conservative batch interval (say, 5-10 seconds) and a low data rate. To verify whether the system is able to keep up with the data rate, one can check the value of the end-to-end delay experienced by each processed batch. If the delay is maintained to be comparable to the batch size, then system is stable. Otherwise, if the delay is continuously increasing, it means that the system is unable to keep up and is therefore unstable. Once one have an idea of a stable configuration, he can try increasing the data rate and/or reducing the batch size. Obtaining the best performance at the lowest cost is a tedious task if someone has to do it manually for each dataflow program, since manually tuning such configurations wastes someone's time and money.

References

[1] Spark streaming programming guide. *https://spark.apache.org/docs/latest/streaming-programming-guide.html*.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. *CQL: A Language for Continuous Queries over Streams and Relations*, pages 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[3] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*, UTLW'11, pages 17–37. JMLR.org, 2011.

[4] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[5] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, pages 153–160, Cambridge, MA, USA, 2006. MIT Press.

[6] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[7] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232, 10 2001.

[8] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1):3–42, April 2006.

[9] 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, pages 506–515, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

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

[12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[13] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

[14] 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, pages 833–840, USA, 2011. Omnipress.

[15] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, August 2004.

[16] 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, pages 1009–1024, New York, NY, USA, 2017. ACM.

[17] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.