UNIVERSITY OF
CAMBRIDGE

Auto-tuning Spark with Bayesian optimisation

Ross Tooley
Churchill College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the Computer Science Tripos, Part III

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: rjt80@cam.ac.uk

May 28, 2021

Acknowledgements

I would like to send my utmost thanks to Eiko Yoneki and Sami Alabed for supervising me. They have given me advice and support throughout my project despite the pandemic preventing us from ever meeting!

Declaration

I Ross Tooley of Churchill College, being a candidate for the Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11,822

Signed: Ross Tooley

Date: May 28, 2021

Abstract

Computer systems are difficult to auto-tune because they are long running and they have many configurable variables. This project presents a sample-efficient, high-dimensional auto-tuner which uses Bayesian optimisation with a directed-acyclic-graph (DAG) surrogate model. The tuner is specialised for computer systems, and uses a system's architecture, its intermediate metrics and its previous evaluations to tune its performance. On a case study of Apache Spark, my tuner optimised six configurable variables to improve Spark's throughput by 10% over its default configuration after 40 Spark evaluations.

This project contributed a novel implementation of DAG models, including a new method to generate the posterior distribution. I added support for deterministic, first-order optimisation techniques to fit the DAG model and optimise the acquisition function. Furthermore, I added support for batch processing on a GPU, which reduced its computation time by a factor of 10 versus a CPU. The implementation is readily available to other researchers as a pluggable module in the BoTorch software stack.

The Spark case study has shown that the tuner is practical on a real-world problem, provides noise-robust optimisation, and is able to overcome difficult issues such as categorical variables. This case study used an expert-defined DAG to model the performance of each of Spark's sub-systems and used these sub-models to predict Spark's overall performance. This tackled the curse of dimensionality by reducing the dimensionality of each sub-system from six to four. The methodology used to build the Spark DAG model is accessible to systems researchers so they can use DAGs to model the performance of many other computer systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

General-purpose computer systems expose many variables which can be tuned to optimise the system's performance. The optimal configuration of these variables is workload dependent, so every user must tune the system for their own application. There is significant research into auto-tuners to remove the burden of manual tuning but systems can be difficult to auto-tune for the following reasons:

- Systems are long running (on the order of minutes or hours) so it is expensive to evaluate many configurations.
- Systems have many configurable variables so the optimisation problem is high-dimensional.

I have implemented a Bayesian optimisation (BO) auto-tuner with a directed-acyclic-graph (DAG) surrogate model. BO [41] is a popular technique for systems because it can tune the system after evaluating only a few configurations. BO works iteratively, as illustrated in Figure 2.1. It uses observations from previous evaluations to fit a probabilistic surrogate model of the system's performance at each configuration, then optimises this surrogate model to decide the next configuration to evaluate. Popular surrogate models for BO, such as Gaussian processes [38] and Tree-Parzen estimators [7], are limited by the *curse of dimensionality* [43, 45]. The amount of training data needed to cover the configuration space and identify the global optimum grows exponentially with the number of dimensions in the configuration space.

DAG models were originally proposed by Dalibard et al. as part of BOAT [11] and they tackle the curse of dimensionality using two properties of systems:

- A system's architecture provides expert knowledge and can be used to create a structured, modular surrogate model.

- Systems output metrics which can be used to identify its bottlenecks.

DAGs are structured models, which are designed by a system expert to mimic the structure of a computer system. DAGs have sub-models to model of each of the computer system's modular sub-systems. These sub-models depend on each other in the same way that the sub-systems depend on each other. A DAG mitigates the curse of dimensionality because each sub-system depends on a small number of other sub-systems so the dimensionality of each sub-model is lower than that of overall system.

My auto-tuner includes novel contributions. Deterministic first-order optimisation techniques can now be used to both fit the DAG model and to optimise the acquisition function. Supporting these techniques required a novel method to generate the DAG's posterior distribution, as well as tools from the BoTorch [6] software stack. Furthermore, batch processing can now be used to reduce the tuner's computation time on a GPU because the tuner is implemented on the PyTorch [36] framework. The tuner is readily available to other researchers because it is implemented as a pluggable module in the BoTorch [6] software stack.

The efficiency and practicality of my auto-tuner has been demonstrated with a case study on Apache Spark [48, 47, 4], a framework to distribute big-data jobs across a CPU cluster. My tuner found a configuration after 40 iterations that improves Spark's throughput by 10% over its default configuration. In comparison, random search found a configuration that was 8% better, but took 60 iterations; BO with a Gaussian process for the surrogate model also found a 10% better configuration in 40 iterations, but my tuner had narrower error across repeat experiments, meaning that it showed more resilience to noise in Spark's performance.

Due to constrained time to gather expert knowledge for the performance model, the Spark case study only tuned six variables even though Gaussian processes can handle problems of up to ten [45]. Nonetheless, the Spark DAG model indicated that DAGs can mitigate the curse of dimensionality because its sub-models all had four inputs or fewer. This improved convergence on the Spark case study and should provide even more benefit on higher-dimensional problems.

In summary, this project has made the following contributions:

1. A BO tuner with DAG surrogate models which supports model fitting and acquisition optimisation using
    - deterministic first-order optimisation techniques, supported by a novel method to generate the DAG's posterior distribution.

- batch processing on a GPU.

2. A DAG performance model for Spark.
3. A methodology to create DAG models for other systems, and techniques to overcome practical issues in applying the tuner to real-world problems.
4. A performance improvement for a Spark benchmark of 10% over its default configurations using only 40 evaluations.

Terminology The term *parameter* is overloaded in many related works. This dissertation uses *configurable variable* to describe the parameters of a computer system that can be tuned, *parameter* to describe the parameters in a structured surrogate model, and *hyper-parameter* for all other parameters.

Chapter 2

Background

2.1 Bayesian optimisation

Bayesian optimisation (BO) is a sequential, model-based, black-box optimisation technique. The aim of *black-box* optimisation is to find $\mathsf{argmax}_{\mathbf{x}\in\mathbf{X}}(f(\mathbf{x}))$, where some properties of the function f are known but not its precise definition. The only way to learn more about f is by evaluating it at some $\mathbf{x}$ and observing $f(\mathbf{x})+\epsilon$, the true value distorted by noise. In the Spark-tuning case study $\mathbf{X}$ is the set of configurable variables of Spark, $\mathbf{x}$ is a single assignment of these variables known as a configuration, and f is the throughput of the Spark application under this configuration. There will be many sources of noise in a computer system such as background processes.

A *model-based* optimisation technique creates a mathematical surrogate model of f using known properties of f and observations from f, and then optimises the surrogate model instead of f. This works if the optimal configuration of f is also the optimal configuration of the surrogate model, and works efficiently if the surrogate model is easier to optimise than f. A *sequential* optimisation technique iteratively evaluates a new configuration then uses the observations from previous evaluations to choose the next configuration. Altogether, a sequential model-based technique iteratively evaluates of f, uses previous observations to fit a surrogate model of f, and uses the surrogate model to choose the next configuration to evaluate.

There are two metrics used to compare optimisation techniques; firstly the optimality of the best configuration found, and secondly the time spent finding this point. To find the optimal configuration in the least time, a sequential technique must balance exploration of unobserved areas of the configuration space with exploitation of the most promising areas. BO makes this trade-off quantifiable because its surrogate model

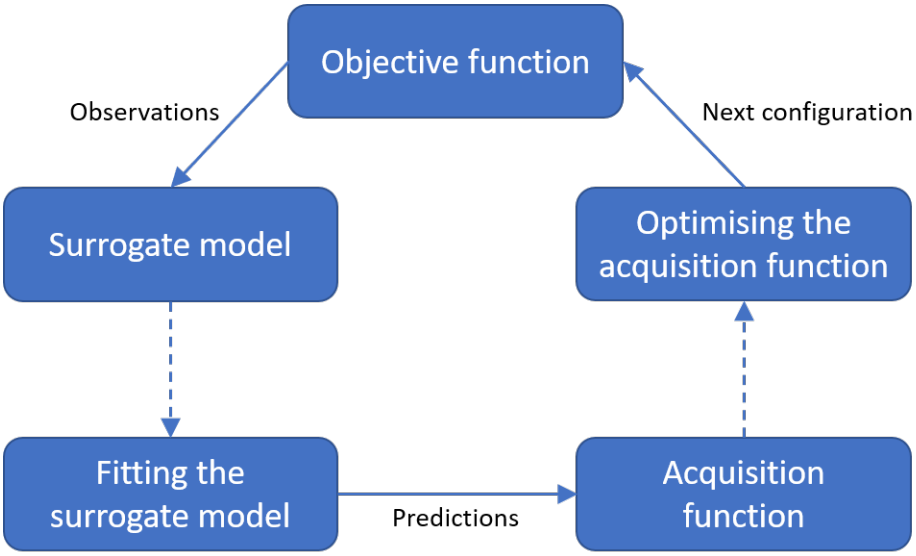

Figure 2.1: The Bayesian optimisation loop

is probabilistic, so unexplored areas of the configuration space will be modelled with wider probability distributions. The surrogate model is used by an acquisition function which scores each configuration on its predicted mean and uncertainty. Different acquisition functions are used for different balances of exploration and exploitation. Finally, the next configuration is chosen by maximising the acquisition function. The full BO loop is summarised by Figure 2.1. For further reading, Shahriara et al. [40] provide an excellent BO review.

2.2 Surrogate models

This project has focused on the BO surrogate model. It has contributed a new implementation of DAG surrogate models with additional features over their original proposal by Dalibard et al. [11]. The motivation for DAG models is to mitigate the *curse of dimensionality* [43, 45].

Curse of dimensionality An accurate surrogate model has the same global optimum as the objective function it is modelling. This is only possible if it is fitted on training data with good coverage of the configuration space. The curse of dimensionality is that the amount of training data required to cover the configuration space grows exponentially with the number of dimensions in the configuration space.

Table 2.1: Comparison of surrogate models for BO

Model	Advantages	Disadvantages
Parametric models	• Quickly fit long-distance trends	• Require known structure of f
Gaussian processes [38]	• Expressive • Flexible	• Fitting is $O(n^3)$ in train-data size [40] • Continuous, non-hierarchical configuration space only
Tree-Parzen estimators [7]	• Fitting is $O(n)$ in train-data size • Categorical and hierarchical configuration space supported	• Less sample efficient than GP [41]
Random forests [29]	• Computationally very cheap • Categorical and hierarchical configuration space supported	• Inaccurately extrapolates uncertainty [40]

DAG models DAG models are structured models designed to mimic the structure of a computer system. A computer system has modular sub-systems and DAGs are composite models with a sub-model for each the system's sub-systems. These sub-systems depend on each other, so DAGs use the performance of one system to help predict the performance of another.

A DAG is defined by a system expert who knows which sub-systems are important to model and how they depend on each other. The sub-models are laid out as the nodes of a directed acyclic graph with the arcs representing the dependencies between sub-models. The sub-models are fitted independently using the metrics output by each sub-system. They do prediction sequentially, using the predictions of child metrics to predict the parent metrics.

DAG models mitigate the curse of dimensionality because each sub-system only depends on a small number of other sub-systems. Each of the DAG's sub-models has fewer dimensions than the overall configuration space so each sub-model requires less training data to identify its optimal input.

Sub-models Theoretically, a DAG model is agnostic to the type of surrogate model used for each sub-system, but my implementation currently only supports one. Table 2.1 lists the potential choices of surrogate model and outlines their advantages and disadvantages.

Parametric models fit the data to a certain structure, for example linear models attempt to fit a linear contribution from each configurable variable. The rest of the models are non-parametric and fit a similarity between configurations rather than a relationship between inputs and outputs. Combining these model types creates semi-parametric models which include a parametric sub-model for the parts of f with known structure and a non-parametric sub-model for the rest. My implementation supports a semi-parametric model with a parametric function to fit the long-distance mean and a non-parametric Gaussian process to fit deviations from the mean. This makes each sub-model flexible, sample efficient and able to incorporate expert knowledge about the sub-system.

2.3 Spark

Spark [48, 47, 4] is a framework to distribute big-data processing across a cluster of multi-CPU machines. Spark parallelises an application, schedules it across the cluster, manages data movement between distributed tasks and implements fault tolerance. Spark has remained popular for over ten years [46] having mostly replaced the MapReduce-based framework Hadoop [12, 14]. Spark processes data faster than Hadoop because it stores data in memory wherever possible. Spark's programming abstraction is also more useful because it supports a chain of map and shuffle operations rather than a single map and reduce.

Spark exposes many configurable variables whose optimal values are application dependent and cluster dependent. For example these variables can control how many parallel tasks to divide the workload into and how many cluster resources to allocate to each task. Spark provides default configuration values [15] but previous works [20, 37] have shown case studies where the optimal values improve the performance by over 20%. However, manual techniques are labour intensive and require expert knowledge of Spark. The aim of this project is to incorporate expert knowledge into an automated tuner so that any Spark-application writer can use an efficient tuner without needing expert knowledge of their own.

This project focuses on Spark jobs that are repeatedly executed on similar batches of data, for example when Spark is used as a stage in a data pipeline. This use case is particularly amenable to sequential-optimisation techniques because there is likely to be one optimal configuration across all batches. BO can tune this application online by using successive executions to explore and exploit the configuration space. The objective metric chosen is *throughput*, which measures the rate at which input records

are processed. This is a useful metric to optimise because it will make executions finish faster and allow the cluster to be freed for other applications.

Spark has high start-up overhead because it distributes computation [30] so most Spark applications are run for many minutes or hours. This motivates the use of BO which has good sample efficiency at the expense of longer model-fitting and acquisition-optimisation times. Furthermore, Spark is a good candidate for structured optimisation because its implementation is clearly documented [16] and it outputs many metrics [17] to show how its sub-systems are performing at run-time.

Chapter 3

Spark performance model

This project tunes Apache Spark [48, 47, 4] as a case study to demonstrate the efficiency and practicality of DAG models. An auto-tuner for Spark must be sample efficient because Spark is used for jobs that run for minutes or hours, and the tuner must be able to scale to high dimensions because Spark has over 30 configurable parameters. Spark has a well-documented architecture which can be used to decompose its performance model into a DAG, and Spark has many run-time metrics which can be used to help predict the overall performance.

This section presents a performance model for Spark and a systematic methodology to create it and encode it as a DAG. This methodology is a classic systems performance analysis, making it accessible for systems researchers to build DAGs for their own systems. The architecture of Spark highlights the bottlenecks in Spark's performance and the metrics that identify these bottlenecks. A data-driven analysis determines how these metrics are affected by the configurable variables and each other, and how the metrics affect the overall performance. The DAG designer adds a node to the DAG for each metric and an arc for each relationship between metrics.

This project has aimed to optimise the `SQL/Aggregation` benchmark from the HiBench suite [22], run on a single-machine cluster. This benchmark stresses key bottlenecks in Spark such as a memory spills and inter-process communication, while avoiding some advanced features like caching, broadcasting and stragglers. Six configurable variables have been tuned on this benchmark, as listed in Table 3.1, which were chosen because previous research [13, 20, 27, 37] indicated that they are significant in most applications. It is likely that the benchmark could be optimised further by tuning more of Spark's 30 configurable variables. Given more time the same methodology could be applied to wider range of Spark benchmarks and a larger number of configurable variables to produce a more general model.

Table 3.1: Spark's configurable variables

Configurable variable	Data type	Range, Step	Default
`executor.cores`	int	[1, 8], 1	1
`executor.memory`	int	[512, 14336], 1	1024
`task.cpus`	int	[1, `executor.cores`], 1	1
`memory.fraction`	float	[0.01, 0.99]	0.6
`shuffle.compress`	bool	{True, False}	True
`shuffle.spill.compress`	bool	{True, False}	True

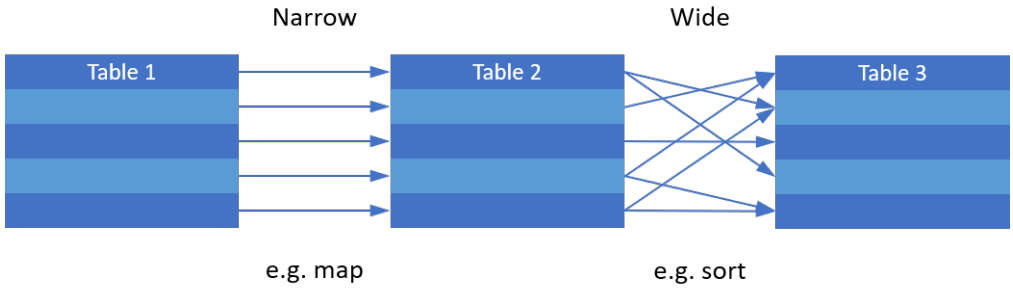

Figure 3.1: Narrow operations are independent across records. Wide operations combine multiple records.

3.1 Architecture of Spark

The first source of expert knowledge of Spark is its documentation [16] which describes its architecture, configurable variables, and metrics. This section summarises the working of Spark, identifies its bottlenecks and discusses how Spark's configurable variables in Table 3.1 can alleviate these bottlenecks when tuned.

Operations The input to a Spark application is a table of data. The Spark application is a sequence of record-wise SIMD operations. Figure 3.1 illustrates the two types of Spark operation. The first are the embarrassingly parallel 'narrow' operations such as `map` and `filter` which process each record independently. The second are the 'wide' operations such as `sort` and `reduce` which combine multiple records and are therefore harder to parallelise. Most Spark bottlenecks are caused by the wide operations because their implementations are far more complex.

Stages Spark breaks the chain of operations into 'stages' which begin with a single wide operation and include all subsequent narrow operations before the next wide operation, as demonstrated in Figure 3.2. Spark parallelises the computation within a stage, but a stage cannot begin until all computation from the previous stage

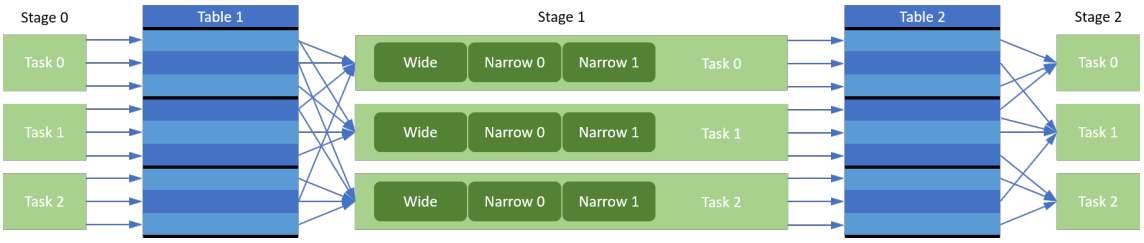

Figure 3.2: Spark separates the application into stages, each of which combines a wide operation and all consecutive narrow operations. Tasks parallelise stages by operating on a subset of the output records.

has completed. At the stage boundaries Spark must serialise all the data from the previous stage into memory, and one major bottleneck in Spark is when this data exceeds the size of memory and must be spilled to disk. Setting the configurable variable `shuffle.compress` to true can reduce disk spills by compressing the previous stage's data, however this is a trade-off with the time spent compressing the data. Furthermore, setting `shuffle.spill.compress` to true compresses data as it is spilt to disk, which exchanges compression to with disk write time.

Tasks The computation within a stage is parallelised into 'tasks' which each operate on a subset of the records. The scheduler distributes tasks to 'executors', which are containers deployed to the hosts in the cluster. Every executor has the same number of cores and amount of memory, as controlled by the configurable variables `executor.cores` and `executor.memory`, and Spark will fill the cluster with as many executors as possible. A greater number of smaller executors can reduce fragmentation of the cluster's resources, but will increase the number of expensive operations fetching data from other executors.

Executors The internals of an executor are illustrated in Figure 3.3. Each task is given a private subset of its executors CPUs but must share its executors memory with other tasks. The variable `task.cpus` controls the number of CPUs allocated to each task. This variable is inefficient if it provides too many or too few CPUs to each task, or if it fragments the executor's CPUs. Finally, the variable `memory.fraction` partitions the executor's memory between execution memory for temporary data structures, and storage memory for accumulating the end-of-stage data. When optimised, it can reduce disk spills and JVM garbage-collection time.

This review of the architecture has highlighted the internal events that are likely to bottleneck Spark. The DAG should include a node to model each of these bottlenecks. The next step is to identify the DAG's arcs by completing a data-driven analysis to

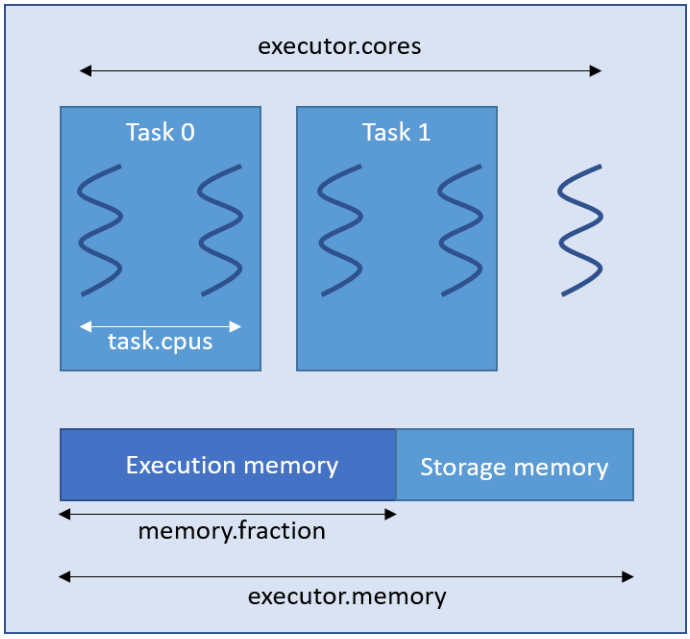

Figure 3.3: An executor multiplexes its CPUs and memory between tasks. The configurable variables affect how the resources are allocated.

determine the relationships between the configurable variables, the metrics and the overall performance.

3.2 Data science to identify dependencies

The architecture of Spark informed the choice of nodes in the DAG by identifying bottlenecks in Spark applications. The next step is to determine the arcs of the Spark DAG by identifying which configurable variables do and do not affect each bottleneck. Arcs allow metrics to be used to help predict the overall performance, and lack of arcs reduces the dimensionality of each sub-model.

Figure 3.4 summarises the metrics used to identify bottlenecks and the relationships found between them. These relationships were found using a data-driven analysis of 128 random configurations of the `SQL/Aggregation` benchmark on the one-machine cluster. Future work on a general-purpose Spark performance model should use observations from multiple benchmarks and multiple clusters to avoid overfitting.

Throughput The overall performance of Spark is measured as its `throughput`. This metrics is inversely proportional to total execution time, which is divided into the two metrics `cpu time` and `non-cpu time`. `Cpu time` includes time spent executing the tasks, including time spent blocking on I/O like spilling to data to disk or

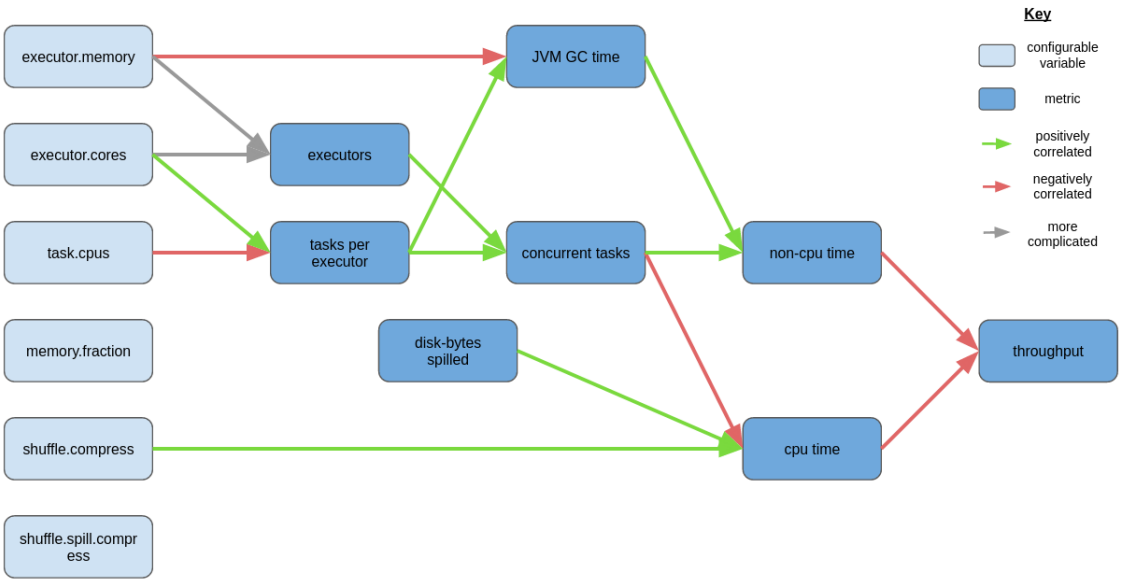

Figure 3.4: Known relationships between Spark's configurable variables, its metrics, and its overall performance metric, throughput.

reading data from other executors. `Non-cpu time` includes everything else such as the executor's garbage collection and Spark's scheduling. The analysis in Figure 3.5 shows that the most important factor in determining `cpu time` and `non-cpu time` is the metric `concurrent tasks`: the number of tasks that can execute in parallel across the cluster. With more `concurrent tasks`, `cpu time` decreases whereas `non-cpu time` increases. The former is likely because fewer cores are blocked on I/O at once, and latter because scheduling more tasks in parallel is harder.

Concurrent tasks The number of `concurrent tasks` is straightforward to predict. It is equal to `executors` $\times$ `tasks per executor`, which respectively are the number of executors Spark creates and the number of tasks that can run concurrently in each executor.

Executors The number of `executors` is determined by `executor.cores` and `executor.memory` because Spark always creates as many executors as it can fit into the cluster.

Tasks per executor `tasks per executor` equals $\lfloor$`executor.cores/task.cpus`$\rfloor$ because each executor runs as many tasks concurrently as it has CPUs for.

Non-CPU time As well as `concurrent tasks` there is one other important metric to determine `non-cpu time`, which is `JVM GC time`: the time executors spend running

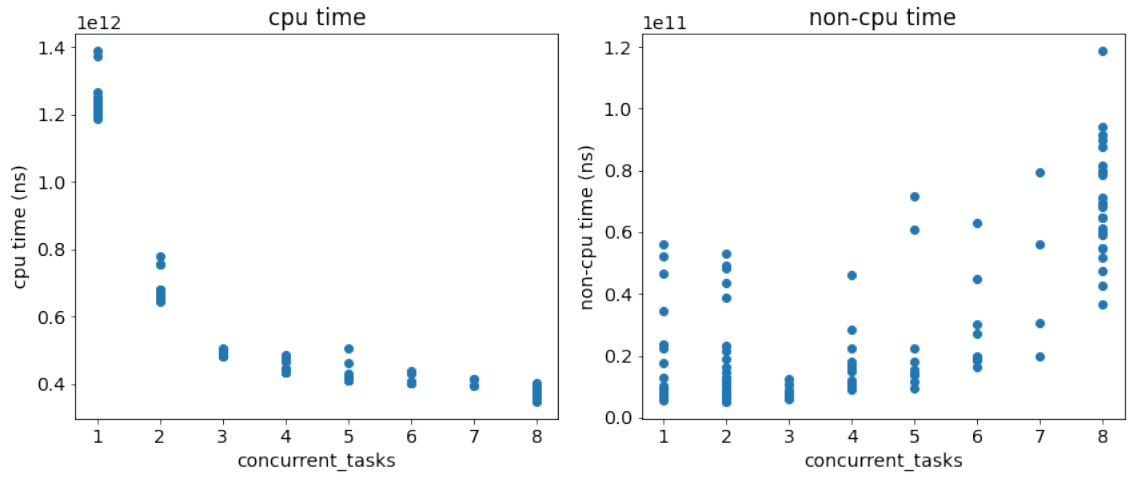

Figure 3.5: Scatter plots showing the effect of `concurrent tasks` on `cpu time` and `non-cpu time` in the test evaluations.

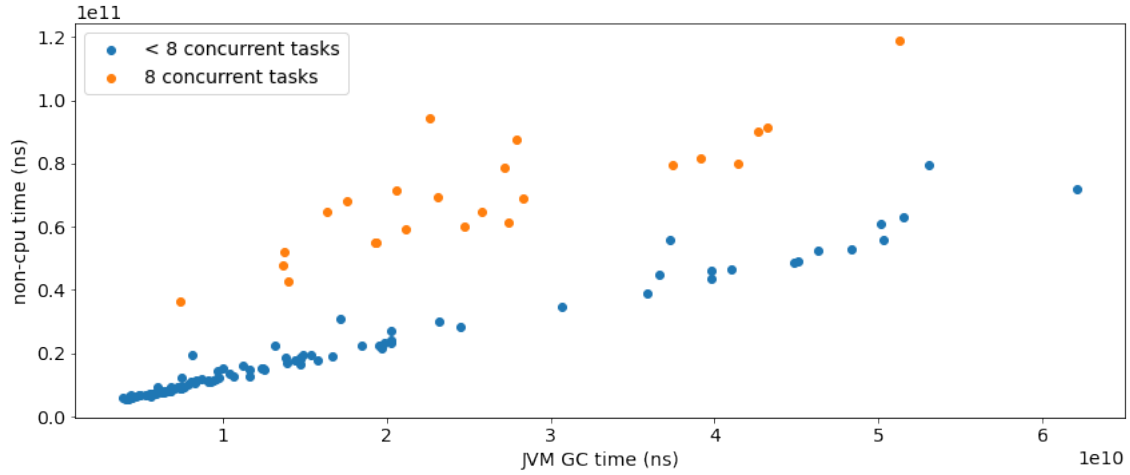

Figure 3.6: Scatter plots showing the effect of `concurrent tasks` and `JVM GC time` on `non-cpu time` in the test evaluations.

the garbage collector. As shown by Figure 3.6, `non-cpu` time is dominated by `JVM GC time`, meaning that the two are linearly correlated, with an additional increase when the number of `concurrent tasks` rises to 8.

JVM GC time. In turn, `JVM GC time` is dependent on `executor.memory` and `tasks per executor`, as shown by Figure 3.7. There are fewer garbage collection when `executor.memory` is larger because the memory fills less often. The executors in the `SQL/Aggregation` benchmark use Java's parallel garbage collector [35] which stops the world when it runs a collection. `JVM GC time` is markedly higher when `tasks per executor` is greater than one, which is likely an artefact of stopping multiple threads rather than a single one.

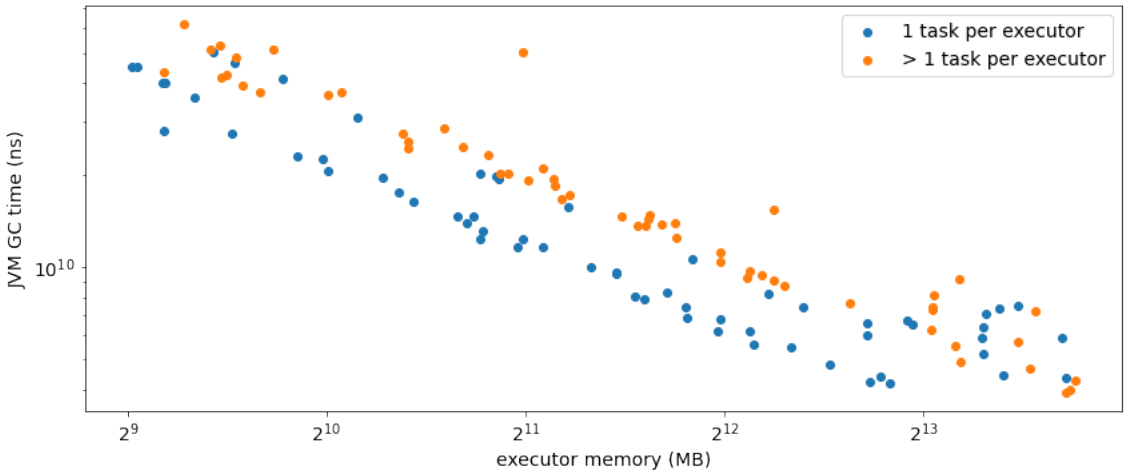

Figure 3.7: Scatter plots showing the effect of `tasks per executor` and `executor.memory` and `JVM GC time` in the test evaluations.

CPU time The other factors that determine `cpu time` are `disk-bytes spilled` and `shuffle.compress`. As shown by Figure 3.8, `cpu time` increases if any data is spilled to disk because this is a slow, blocking operation. Disk spills can be avoided by compressing data written to memory using `shuffle.compress`, but Figure 3.9 shows that this is a trade-off with increased `cpu time`.

Disk-bytes spilled The final metric is `disk-bytes spilled`, however, the test evaluations have shown that there is a high degree of uncertainty when predicting its value. The decision to spill or not to spill appears to be affected by sources of random noise in the system. If a spill occurs then it is usually a large spill, which is presumably because a large spill is only slightly more expensive than a small spill. This behaviour is non-Gaussian and therefore hard to model with Gaussian processes. The ramifications of this are discussed in Section 3.3.

Summary This completes the data-driven analysis. It has revealed the exact relationships between Spark's configurable variables and its metrics within the `SQL/Aggregation` benchmark and informs the choice of arcs in the Spark DAG model. Future work could generalise this data analysis to more benchmark applications, more clusters and more configurable variables in order to find a general-purpose performance model for Spark.

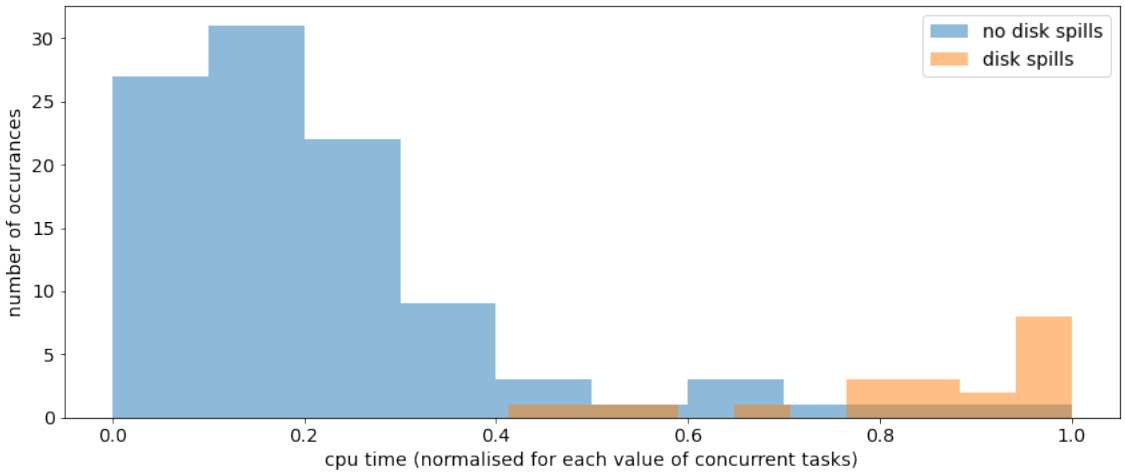

Figure 3.8: Histogram showing the effect of `disk-bytes spilled` on `cpu time` in the test evaluations. Data has been normalised to account for the effect of `concurrent tasks`.

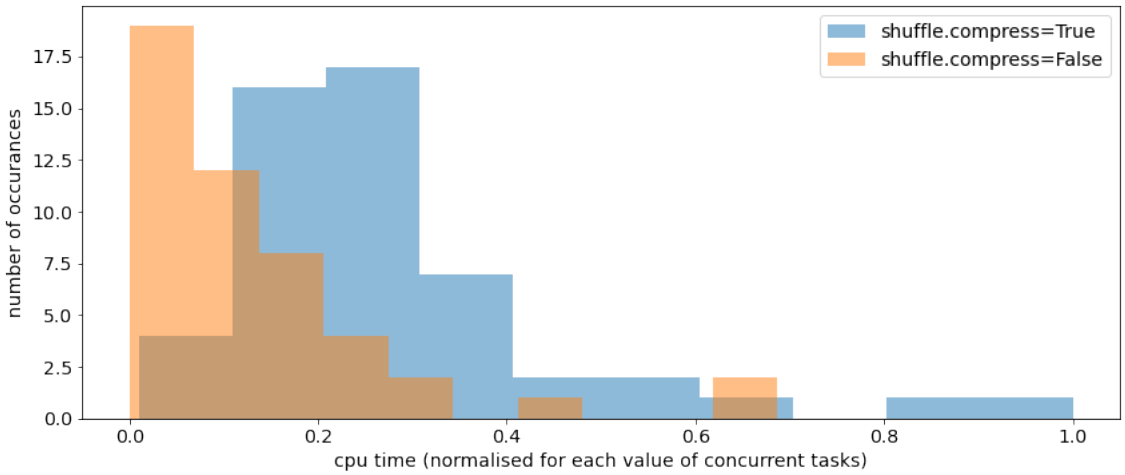

Figure 3.9: Histogram showing the effect of `shuffle.compress` on `cpu time` in the test evaluations. Data has been normalised to account for the effect of `concurrent tasks`.

3.3 Spark tuning vs. mathematical optimisation

This section discusses difficulties in applying the mathematical DAG-BO tuner to the concrete problem of Spark. The issues arise when any abstract assumptions of the tuner are broken in the real-world problem. Some of these issues can be mitigated, often with help of the Ax [23] library.

Categorical variables The sub-models in the nodes of my DAGs are Gaussian processes. These expect real, continuous input variables in order to calculate a distance between each point, but five of the six configurable variables in my Spark benchmark are discrete (three integers, two boolean). To solve this, Ax is used to transform these to continuous variables before passing them to the model and discretise them before passing them back to Spark. The integers are converted to floats of the same value and the booleans are mapped `True` $\rightarrow$ 1.0, `False` $\rightarrow$ 0.0. The same is done for all metrics because metrics become the inputs for other metrics' sub-models.

This solution causes some artefacts that affect the behaviour of BO. There is no training data between the discrete values, so the Gaussian processes will have greater uncertainty in these regions, which makes them more likely to be chosen when optimising the acquisition. The tuner may to escape these regions if it cannot gain more information about them.

Standardising configurations and metrics The hyper-parameters of the Gaussian processes in my DAG model use BoTorch's [6] default initial values. These work best when the input variables are normalised to the range $[0, 1]$ and the output variable are standardised to a mean of 0 and a standard deviation of 1. Therefore, Ax's built-in transformations are used to normalise Spark's configurable variables and standardise its metrics.

Unfortunately, it is less intuitive to define parametric models between normalised inputs and standardised metrics. As shown by example in Equation 3.1, standardising y to y' and normalising x to x' means applying a linear transformation to y and x. Consequently, the simple relationship between y and x in Equation 3.1 requires far more parameters when standardised in Equation 3.4 which makes it harder for a user to define and also harder to fit.

$$y = e^x \tag{3.1}$$

$$y' = m_1 y + c_1 \tag{3.2}$$

$$x' = m_2 x + c_2 \tag{3.3}$$

$$y' = \frac{e^{\frac{x'-c_2}{m_2}} - c_1}{m_1} \tag{3.4}$$

Erroneous inputs BO assumes that the configuration space is a hyper-cube and that every configuration within it is well defined, however this is assumption is rarely true in practise. Previous work has applied ad-hoc workarounds, such as specialised models to deal with hierarchical configuration spaces [7], manually or automatically removing erroneous configurations from the configuration space [3, 27], or by replacing error values with exceptionally bad legal values.

It is particularly difficult to cope with errors in a DAG model. If the computer system throws an error then it might return no metrics, or a subset of its metrics, or incorrect metrics. It may not be possible to replace these incorrect or missing metrics with an exceptionally bad value because it may not be known whether the metric's worst value is its minimum or maximum or a mid-range point. The data-driven analysis showed that the `concurrent tasks` metric inhibits performance when it gets too high and also when it gets too low.

This project chooses to avoid erroneous inputs as a robust solution is likely to require significant future work. The configuration space is trimmed to avoid any erroneous configurations. This requires the linear constraint `task.cpus` $\leq$ `executor.cores`, and once again Ax has built-in functions to enforce this.

Non-Gaussian metrics Some Spark metrics exhibit non-Gaussian qualities, for example `disk-bytes spilled`. When a Spark program is on the cusp of spilling to disk, it will either spill nothing or it will do a large disk spill to avoid many smaller disk spills. The distribution of `disk-bytes spilled` is one-sided as it has a high probability of spilling zero bytes and zero probability of spilling negative bytes. It is also non-Gaussian because it has two modes, one at zero and one at the size of the large disk spill. My implementation only supports modelling metrics using Gaussian process with parametric means, which is a poor approximation of `disk-bytes spilled` so slows down convergence. This motivates future work to support heterogeneous sub-models such as Tree-Parzen estimators [7] to provide better models for non-Gaussian metrics.

Lastly, some of Spark's metrics have flat areas where changing the configuration slightly has no effect on the metric. For example, the number of `executors` will be limited by `executor.cores` or `executor.memory` so increasing the non-limiting variable has no effect on its value. Flat areas inhibit first-order optimisation techniques because the gradient is not informative in deciding the optimal direction. BoTorch's [6] built-in multi-restart optimisation techniques help mitigate this issue by optimising the acquisition function from many starting points. This technique also helps avoid local optima.

3.4 Incomplete expert knowledge

It is unlikely that an expert can define every dependency between the configurable variables and metrics, so a DAG model must be resilient to incomplete expert knowledge. This section categorises the ways that a DAG model can be incomplete, gives an intuition about how each will affect tuning and indicates how a DAG designer should cope with incomplete knowledge. A quantitative evaluation of their effects on tuning time and optimality is left to future work due to lack of project time and resources.

Missing inputs In this situation, a metric has a dependency on a configurable variable which is not registered in the DAG, either as a direct arc or via multiple arcs. The metric's sub-model will interpret the effect of this configurable variable as noise. Given sufficient tuning time the tuner will search the noise, but it will take longer because it cannot learn the relationship with the missing input. A designer can avoid this by measuring the sensitivity of the metric to each configurable variable over training data and adding arcs directly from the significant variables to the metric. This may skip metrics that would otherwise decompose the dependency into sub-models, but it will ensure all significant variables are included.

Insignificant inputs In the opposite situation, a dependency may be registered in the DAG between two nodes that do not depend on each other. The effect of this will not be so destructive because the sub-model will learn a constant relationship between the two. It does unnecessarily increase the dimensionality of the sub-model but it is the safer option when unsure about a dependency.

Correlation without causation A mathematical model fits any correlation, so it can learn a relationship between nodes that only appear to depend on each other. One example is a reversed arc where the dependency is registered in the wrong direction. Another example is a confounding variable, where two metrics are correlated because they both depend on the same configurable variable, but actually have no dependency on each other. The false correlation is likely to break down for some configurations, for example where one value of the child metric corresponds to two values of the parent metric. Just like the missing inputs problem, the tuner will require more tuning time to find the optimal value using incorrect dependencies. To avoid this, a designer should be able to justify the correlations seen in the training data using expert knowledge of the system.

Incorrect parametric models The designer can specify a parametric model for each sub-model to improve the tuner's sample efficiency. If the parametric model is wrong then it will diverge from some areas of the input space. The non-parametric part of the model can only resolve the divergence if it is small. A large divergence may prevent the model-fitting algorithms from converging, or it may force the tuner into a local optimum because the incorrect parametric model steers the tuner away from the global optimum. The DAG designer can use the default parametric model to fit a constant mean if they are unsure of the correct parametric model to use.

In summary, DAG models are resilient to failure in most cases of incomplete knowledge, although they will take more time to converge. The structure of the DAG is a type of Bayesian prior and priors lose their impact as more data is added. Missing arcs force the model to search the noise and surplus arcs force the model to search a larger configuration space. A DAG with no expert knowledge is a single node, which is simply equivalent to a Gaussian process with a constant mean.

3.5 Spark DAG model

This project has presented a performance model for Spark which was created using a systems-analysis methodology. The documentation of Spark's architecture was used to understand the likely bottlenecks in Spark's performance, and Spark's metrics were used to identify when these bottlenecks occur. A data-driven analysis was completed to determine the relationships between Spark's configurable variables, metrics and overall performance. The last step is to encode this performance model as a DAG

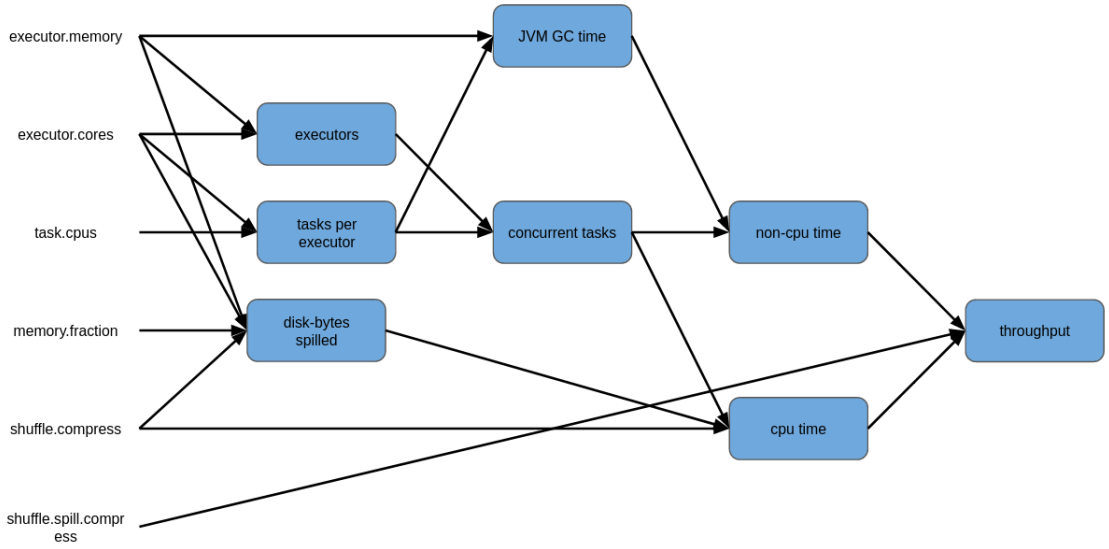

Figure 3.10: Visualisation of the nodes and arcs in the Spark DAG model.

model, and this DAG model is illustrated in Figure 3.10.

The DAG looks similar to Figure 3.4. Every metric is given a node containing a Gaussian process to predict that metric. Every relationship between metrics is given an arc, which uses the child metric to predict the parent metric. No sub-model in the DAG has more than 4 inputs, demonstrating the DAG model's ability to mitigate the curse of dimensionality. Furthermore, the code to create this DAG using my Python API is shown Figure 3.11, which shows how simple it is to turn a performance model into a DAG model.

There are two cases of incomplete knowledge. First, the effect of `shuffle.spill.compress` on any metric is unknown and it may be an insignificant variable. To resolve this, a direct arc has been added from `shuffle.spill.compress` to the objective metric, `throughput`, which allows its effect to be incorporated into the model without adding dimensionality to any other metric. Second is the `disk-bytes spilled` node, for which no reliable model has been found to predict its value. This node is included to reduce the dimensionality of the `cpu time` model despite the chance that it makes incorrect predictions. The test evaluations and Spark's architecture indicate four configurable variables that it is sensitive to, so direct arcs have been added from these variables to `disk-bytes spilled`. This keeps the dimensionality low, but means that any effects due to the other two configurable variables will be lost as noise.

Every node in the DAG uses the default kernel and mean for the Gaussian process as discussed in Section 4.7. Specialised parametric means were tried in initial experiments but they required a large number of parameters which made them difficult to

```
class SparkDag(Dag, DagGPyTorchModel):
  def define_dag(self) -> None:
    # configurable variables
    ec = self.register_input(name="executor.cores")
    em = self.register_input(name="executor.memory")
    tc = self.register_input(name="task.cpus")
    mf = self.register_input(name="memory.fraction")
    sc = self.register_input(name="shuffle.compress")
    ssc = self.register_input(name="shuffle.spill.compress")

    # metrics
    e = self.register_metric(name="executors", dependencies=[ec, em])
    tpe = self.register_metric(name="tasks_per_executor", dependencies=[ec, tc])
    dbs = self.register_metric(name="disk_bytes_spilled",
                               dependencies=[ec, em, mf, sc])
    jgc = self.register_metric(name="jvm_gc_time", dependencies=[em, tpe])
    ct = self.register_metric(name="concurrent_tasks", dependencies=[e, tpe])
    ncpu = self.register_metric(name="non_cpu_time", dependencies=[jgc, ct])
    cpu = self.register_metric(name="cpu_time", dependencies=[dbs, ct, sc])

    # objective metric
    t = self.register_metric(name="throughput", dependencies=[cpu, ncpu, ssc])
```

Figure 3.11: The Spark DAG model, defined using my Python API.

fit. They also converged to local optima, implying that they were not accurate for the entire configuration space.

Chapter 4

Upgrading DAG models

This project contributes a novel implementation of DAG models, a type of BO surrogate model originally proposed by Dalibard et al. [11]. My implementation is built from scratch as a module within the BoTorch software stack [6]. This has allowed it to outsource the rest of the BO loop to other libraries and add novel features to DAG tuners by leveraging recent research from those libraries. The additions are deterministic, first-order optimisation techniques to fit the model and optimise the acquisition, and batch processing to reduce computation time on a GPU. The Spark DAG model in Section 3.5 was defined using my implementation of DAGs and demonstrates how a user can instantiate a DAG model for their own system.

- Section 4.1 motivates DAGs and describes how they are fitted and optimised. These ideas are taken directly from Dalibard et al. [11] whereas the rest of the chapter is my own work.
- Section 4.2 shows how my API abstracts away all complexity from the user.
- Section 4.3 introduces my modular implementation, including how it leverages the BoTorch software stack [6].
- Sections 4.4, 4.5 and 4.6 discuss the novel features within my implementation.
- Section 4.7 gives reasonable default values for my implementation's hyperparameters.

4.1 Mechanics of DAG models

Originally proposed by Dalibard et al. [11], DAGs are specialised models for predicting the performance of computer systems. This section describes their intuitive design,

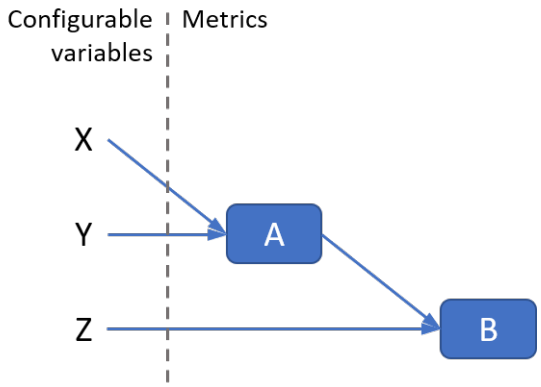

Figure 4.1: A simple DAG model with three configurable variables which predicts two metrics. Furthermore metric A is used to predict metric B.

and explains how model fitting and prediction remains efficient despite such large models.

Computer systems expose run-time metrics, and a DAG uses these metrics to help predict the system's overall performance. Each node of a DAG model is a sub-model which predicts the performance of one metric. Each arc specifies a metric's dependency on one of the configurable variables or other metrics. An expert defines the shape of the DAG, using their knowledge of the system architecture to decompose the system into sub-systems that depend on each other. Figure 4.1 shows a simple example DAG where metric A is predicted using configurable variables X and Y, then metric B (the overall performance) is predicted using metric A and configurable variable Z.

Model fitting and prediction are implemented by the DAG library. The model is fitted on a set of observations, where each observation contains the input value of each configurable variable and the output value of each metric, e.g. $(x_i, y_i, z_i; a_i, b_i)$ in the simple DAG example. Each of the DAG's sub-models are fitted independently, e.g. sub-model A is fitted on observations $(x_i, y_i; a_i)$ and sub-model B on $(z_i, a_i; b_i)$. This makes model fitting fast despite the large number of model parameters to be fitted, because fitting a single model is exponential in its number of parameters whereas fitting multiple sub-models is linear in the number of sub-models. The sub-models can also be fit in parallel since they are independent.

A DAG model predicts the value of every metric given an input configuration. In the simple DAG example, with the configuration (x_i, y_i, z_i), model A predicts a_i from (x_i, y_i) and model B predicts b_i from (z_i, a_i). Each sub-model does its prediction in topological order, and again the cost of prediction is linear in the number of sub-models. One complexity is that the sub-models are Bayesian, so they output a

```
class ExampleDag(Dag, DagGPyTorchModel):
    def define_dag(self) -> None:
        X = self.register_input(name="X")
        Y = self.register_input(name="Y")
        Z = self.register_input(name="Z")
        A = self.register_metric(name="A", dependencies=[X, Y], mean=None)
        B = self.register_metric(name="B", dependencies=[Z, A], mean=None)
```

Figure 4.2: The example DAG, defined using my Python API.

distribution describing the probability of the metric's value given the input configuration and the training data. Since the sub-models expect discrete inputs, the arcs take samples from this probability distribution and forward the samples to the next sub-model.

Both Dalibard et al. and my implementation use a semi-parametric model at every node which fits the long-distance trends with a parametric mean function and fits any deviations from this trend using a Gaussian process [38]. Theoretically, DAGs could support heterogeneous sub-models including Tree-Parzen estimators [7] or random forests [29], but this is left to future work.

In summary, an expert uses a DAG model to decompose their system into dependent sub-systems, and the DAG uses the metrics from each sub-system to predict the system's overall performance. A DAG can mitigate the curse of dimensionality because each sub-model is likely to have fewer inputs than the overall model, and because the each sub-model is likely to be modelling a simpler relationship.

4.2 API

My implementation abstracts away the complexity of creating sub-models, fitting them and using them to make predictions. It presents a declarative Python API to the end user which only requires them to define the structure of the DAG model. All other functions of the model and wider BO loop are handled by library code.

As an example, the code in Figure 4.2 uses the API to define the example DAG from Figure 4.1. The user registers the system's configurable variables X, Y and Z, registers the metrics A and B, and states the dependencies of each metric. Each registered metric then becomes a node of the DAG and each dependency becomes an arc. The API enforces early binding of dependencies, which ensures that the resulting graph is acyclic.

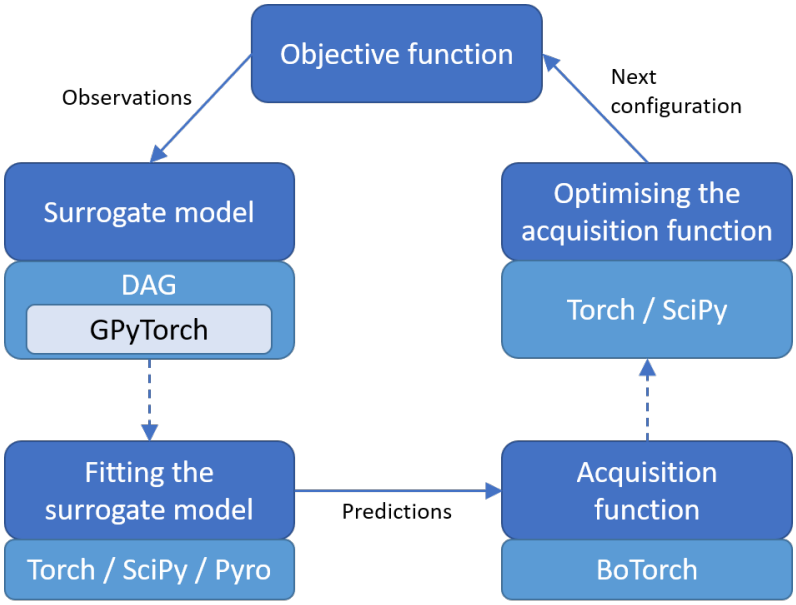

Figure 4.3: How the components of the BO loop are divided between the libraries in the BoTorch stack.

An optional `mean` can be added to each metric. By default, each metric is modelled using a Gaussian process with a constant mean, but this optional parameter allows the user to add a parametric mean function to the Gaussian process.

4.3 Integration with the BoTorch software stack

My implementation of DAG models is a module within the BoTorch software stack [6]. This stack contains libraries for each component of BO, and it is designed for modularity and extensibility so that new research can be swapped into the stack or extend it. My implementation exclusively implements the mechanics of DAG models, and outsources the implementation of the DAG's sub-models and the wider BO loop to libraries in the BoTorch software stack. Figure 2.1 shows the BO loop and Figure 4.3 shows how this is divided between the different libraries in the BoTorch stack, with the DAG model module slotting in as the surrogate model. This section briefly describes the purpose and benefits of each library.

Each sub-model in a DAG is a Gaussian process, and these are implemented by GPyTorch [19], a specialised library containing many optimisations for Gaussian processes. GPyTorch is built on PyTorch [36] so supports batch processing to speed up computation on a GPU and auto-grad for gradient-based optimisation techniques. GPyTorch supports many model-fitting algorithms, including deterministic first-order techniques from SciPy [44] such as `L-BFGS-B`, stochastic first-order techniques from

PyTorch such as `Adam`, and Bayesian methods from Pyro [9] such as Markov-chain Monte Carlo or stochastic variational inference.

BoTorch [6] provides efficient implementations of acquisition functions. Again, these implementations support batch processing and auto-grad via PyTorch so that the acquisition functions can be optimised by first-order optimisers from SciPy or PyTorch. BoTorch is also the glue of the stack, and provides minimal APIs to connect the libraries together. The Ax [23] library is a wrapper around the BoTorch stack with tools to handle practical issues in the real world. This includes massaging the variables so they are continuous, standardised, and have no missing data.

There are other libraries for BO which could have been used for this project. Spearmint [41] is designed to be easy to use, GPyOpt [5] is also designed for extensibility, GPflowOpt [26] is built on TensorFlow [1], ProBO [34] is designed for structured BO. The BoTorch stack was chosen because it is designed to be easy to add a new module like DAGs, and because it is scalable and implements many recent advances in BO.

In summary, the implementation of each sub-model, and the wider BO loop is outsourced to the BoTorch software stack. This takes advantage of optimisations and new research in this system stack, and makes the DAG model into a pluggable module available to other researchers. This module handles the mechanics of DAG models, such as fitting each sub-model independently, and using one metric to help predict another. The next sections discuss the novel ideas in my implementation, which are primarily ways to take advantage of the new research in the BoTorch software stack.

4.4 Reparametrising the posterior distribution

The BoTorch software stack has support for acquisition optimisation using deterministic, first-order methods. In order for these to be stable, the surrogate model's posterior distribution must be reparametrised [25]. The first feature added to my implementation is a reparametrised posterior distribution for DAG models so that my auto-tuner can use deterministic, first-order methods to optimise the acquisition.

Posterior distribution A surrogate model for BO is probabilistic. When it makes a prediction about a new configuration, it outputs a probability distribution over all possible values of the objective function, known as the posterior distribution.

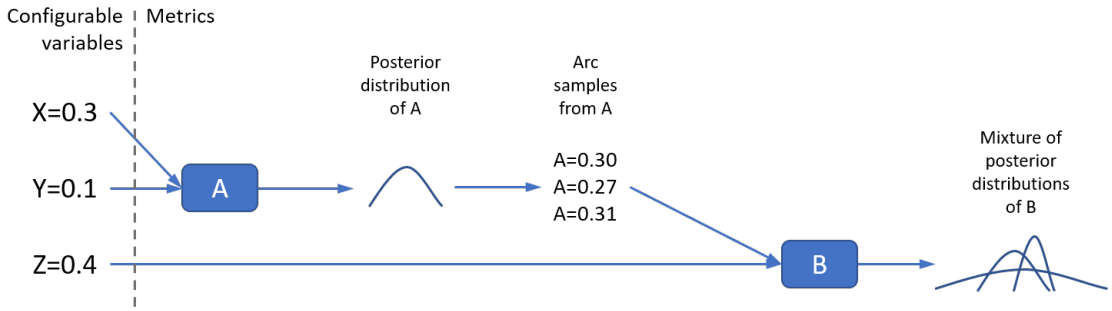

Figure 4.4: Child metrics are used to help predict parent metrics. This is implemented by sampling from the child metric's posterior distribution, and using each sample to create a new posterior distribution for the parent.

Reparametrisation trick A reparametrised distribution is a distribution expressed as a deterministic transformation of the unit normal distribution [25]. For example, a normal distribution with mean μ and standard deviation σ would be reparametrised as

$$N(\mu, \sigma) = \mu + \sigma \cdot N(0, 1) \tag{4.1}$$

For deterministic optimisation to be stable, the posterior distribution of the surrogate model in BoTorch must be reparametrised. This is straightforward for Gaussian processes because their posterior distributions are Gaussians, but this project required a novel implementation because the posterior distributions of DAG models are mixtures of Gaussians, due to an internal technique called arc sampling.

Arc sampling An arc is the connection in the DAG from a 'child' metric to a 'parent' metric, and it forwards the prediction of the child so it can be used to predict the parent. This is complex to implement because each sub-model in the DAG is probabilistic so the child outputs a posterior distribution even though the parent expects a discrete input. To resolve this disparity, the arc takes samples from the child's posterior distribution, known as arc samples. For each arc sample, the parent produces a distinct posterior distribution, thereby producing a mixture of posterior distributions across all arc samples. This process is illustrated in Figure 4.4. The mixture is a Monte-Carlo approximation of the true posterior, which converges as the number of arc samples increases.

Gaussian approximation My implementation does not directly reparametrise the mixture of posterior distributions as there was no practical method found to do this. It first approximates the mixture of posterior distributions as a Gaussian and then reparametrises the Gaussian. This approximation is similar to how a Gaussian process

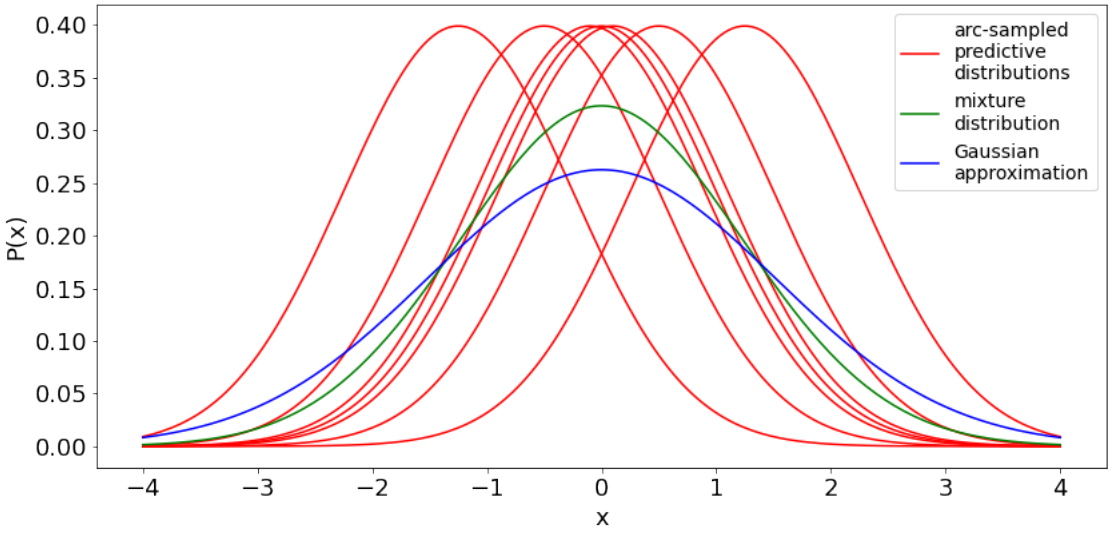

Figure 4.5: The creation of the posterior distribution. The arc-sampled posterior distributions are combined into a mixture distribution, which is then approximated as a Gaussian distribution.

approximates the posterior distribution as a Gaussian. The Gaussian is given the same mean, μ, and variance, σ^2, as the as the mixture distribution, and it is easily reparametrised using Equation 4.1. μ and σ^2 are calculated using the means and variances of each posterior distribution in the mixture, μ_i and σ_i^2. Each posterior distribution has equal weighting in the mixture, since each arc sample is drawn at random, hence the standard equations are:

$$\mu = \frac{1}{N}\sum_{i}^{N}\mu_i \tag{4.2}$$

$$\sigma^2 = \frac{1}{N}\left(\sum_{i}^{N}(\sigma_i^2 + \mu_i^2)\right) - \mu^2 \tag{4.3}$$

In summary, the DAG's posterior distribution is approximated as a Gaussian distribution so that it can be easily reparametrised. A graphical example is shown in Figure 4.5. This enables BoTorch to implement quasi-Monte-Carlo acquisition functions which make deterministic, first-order acquisition-optimisation methods stable enough to use.

4.5 Fitting the model with first-order methods

Dalibard et al. [11] fit DAG models with Bayesian inference, using the sequential Monte-Carlo algorithm. By leveraging the BoTorch [6] stack, my implementation also supports fitting DAGs using a wide range of inference techniques including Bayesian inference, maximum-likelihood estimation, and variational inference.

A DAG contains semi-parametric sub-models which each include a parametric model to fit the long-distance mean and a non-parametric Gaussian process to fit deviations from the mean. Each sub-model is fitted independently while the parametric and non-parametric models within a sub-model are fitted simultaneously. The fitting process aims to find the optimal values of the parametric model's parameters and the non-parametric model's hyper-parameters.

Bayesian inference outputs a posterior distribution to describe the probability of any parameter value being optimal. Maximum-likelihood estimation is a frequentist approach which finds a discrete parameter value that maximises the likelihood of the model fitting the data. Variational inference is an umbrella term covering approximate forms of both of these techniques. Bayesian inference and variational Bayesian inference are available in the BoTorch stack from the probabilistic programming library Pyro [9]. For maximum-likelihood estimation, GPyTorch [19] provides the functionality to compute exact or approximate forms of the model's likelihood, and SciPy [44] and PyTorch [36] provide deterministic and stochastic algorithms to optimise it.

Every parameter in my DAG models is a PyTorch parameter, including the hyper-parameters of the Gaussian processes and the parameters of the parametric model. This means that each model supports auto-grad to access the gradients and maximise the likelihood using first-order optimisation methods. As discussed in Section 4.7 deterministic first-order methods are used by default because they are faster than Bayesian inference and less sensitive to their hyper-parameters than stochastic first-order methods.

In summary, my implementation has added support for first-order model-fitting methods using auto-grad information from PyTorch [36].

4.6 Batch processing

A major benefit of building on PyTorch [36] is out-of-the-box support for batch processing to parallelise SIMD instructions on a GPU. This is already used in GPy-

Torch [19] for their batch-processed Gaussian-process regression algorithm called BBMM, in BoTorch [6] to batch the Monte-Carlo samples in the acquisition functions, and also in BoTorch to run multi-restart optimisation as a batch of individual optimisations.

My implementation benefits from GPyTorch's and BoTorch's uses of batching, and also uses batch processing to efficiently implement arc samples in DAGs. Arc samples are samples taken from the posterior distribution of the child metric and used to generate a mixture of posterior distributions of the parent metric. PyTorch batch processing is used to draw the samples and compute the mixture of posterior distributions in a single batch. By creating the mixture distribution as a batch it is also more efficient to compute its Gaussian approximation using Equations 4.2 and 4.3.

There are however some parts of DAG models which cannot be batched because they are not SIMD. While sub-model fitting can be parallelised, it cannot be batched because the sub-models are not necessarily the same type of model and they do not operate on the same shape of input. Heterogeneous parametric models, where each input is modelled with a different relationship, cannot be batched across inputs, but homogeneous parametric models can. This leaves users a design choice to improve the model's fit or its speed.

4.7 Default hyper-parameters

There are many hyper-parameters within the DAG model and wider BO loop. To lighten the burden on the user, my implementation provides sensible default values for these hyper-parameters which are also used in the Spark case study. This section discusses the default values and why they were chosen.

Gaussian-process kernel Each sub-model in my DAGs is a Gaussian process which has a swappable kernel. The choice of kernel is important as it limits the type of functions that can be fit. Many metrics of computer systems will not be smooth, such as when the metric has an upper bound at a resource limit. For this reason the default is the `Matern` kernel, implemented by GPyTorch [19], which has a smoothness parameter ν to allow the kernel to fit non-smooth functions. By default, $\nu = \frac{5}{2}$ because this produces sample functions that are twice differentiable, which is sufficient for most optimisation techniques [41].

Gaussian-process mean If the user is unable to provide a parametric mean function, my implementation defaults to a constant mean function with a single learnable parameter c with a unit normal prior. This allows the Gaussian process to learn any bias empirically, as it is equivalent to empirically setting a non-zero prior on the mean [39].

Model fitting and acquisition optimisation To fit the model and also to optimise the acquisition, my implementation uses `L-BFGS-B` from the SciPy [44] library by default. This is a deterministic algorithm which approximates Newton's method using only first-order gradients. Alternatively, stochastic first-order methods from PyTorch [36] could have been used to fit the model and optimise the acquisition, or Bayesian methods such as MCMC from Pyro [9] to fit the surrogate model.

A model fitted with Bayesian parameters is generally more robust as it also models the uncertainty in each parameter, making it well suited to BO. Furthermore, MCMC builds the posterior distribution out of samples, meaning that it integrates naturally with arc samples, but in early experiments Pyro's MCMC was found to be prohibitively slow, meaning that model-fitting time became the dominant part of tuning time. First-order methods are faster, and deterministic first-order method were chosen because they are less sensitive to their own hyper-parameters than stochastic methods [6].

Arc samples Arc samples are a Monte-Carlo method for approximating a metric's posterior distribution using a DAG. Increasing the number of arc samples improves the approximation and makes the acquisition-optimisation function more stable at the expense of more computation time. The number of arc samples was chosen empirically, with results shown in Figure 4.6. In this experiment, the Spark DAG model was fitted on the ten bootstrap configurations from the convergence experiment in Section 5. All other hyper-parameters were set to default, and the same CPU and GPU were used as in Section 5.

The top graph of Figure 4.6 shows that the posterior distribution was more stable when more arc samples were used. For this graph, the fitted model was used to generate 20 posterior distributions of Spark throughput at its default configuration, each with a new random arc-samples seed. The graph shows the uncertainty in the mean and standard deviation across these posterior distributions.

The bottom graph of Figure 4.6 shows that the time to optimise the acquisition increased linearly as the number of arc samples increased. It shows the mean and stan-

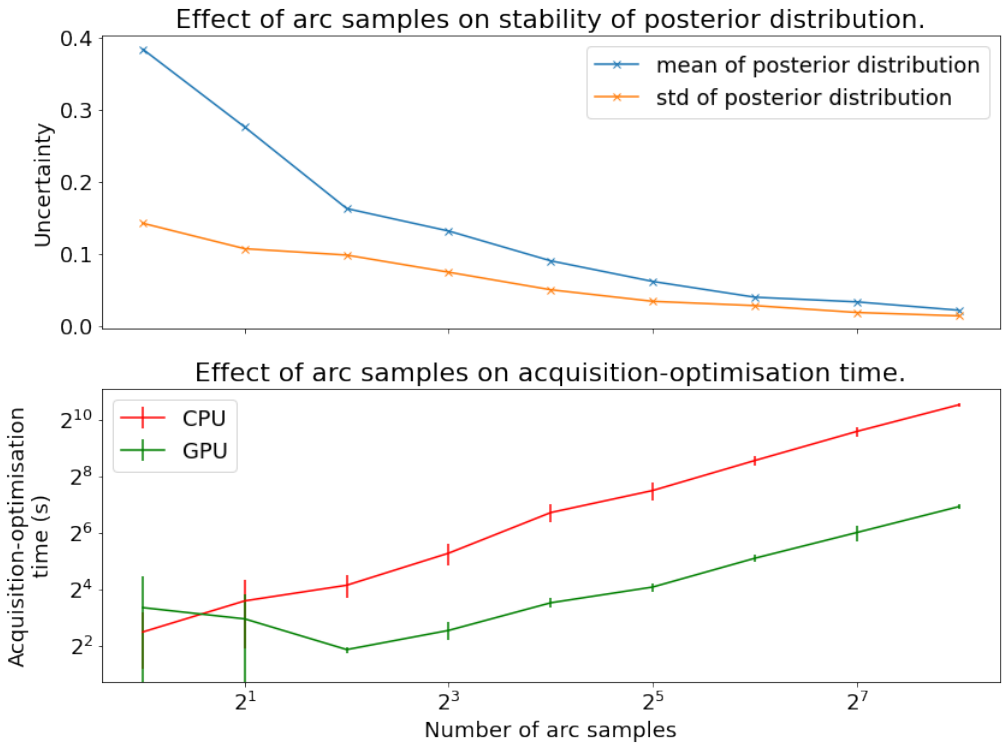

Figure 4.6: Increasing the number of arc samples improves the approximation of the posterior distribution at the expense of longer acquisition-optimisation time.

dard distribution of acquisition-optimisation time across 5 repeats. This experiment also demonstrates that batch-processing on a GPU massively sped up acquisition-optimisation time when using a large number of arc samples. The rate of change was $6\frac{\text{seconds}}{\text{arc sample}}$ when using a CPU but only $0.5\frac{\text{seconds}}{\text{arc sample}}$ when using a GPU, and the GPU optimised the acquisition 12 times faster when using 256 arc samples.

These results confirm that the posterior distribution converges with increasing arc samples, but that this convergence has diminishing returns and has linear cost in acquisition-optimisation time. The default number of arc samples used in the Spark DAG model is set to 64 because the empirical stability only fractionally improved between 64 and 256 samples, and the CPU acquisition-optimisation time was still less than Spark's evaluation time. It is likely that users will benefit from setting this hyper-parameter manually to fit their own hardware and DAGs, because the uncertainty in the posterior distribution also increases with more layers in the DAG.

Acquisition function The default acquisition function is noisy expected improvement, implemented by BoTorch [6]. Expected improvement computes the probability of a configuration improving over the best configuration so far, weighted by the size of that improvement. Various convergence properties have been shown for expected

improvement [40] making it good for online tuning such as my Spark case study. The extended version of expected improvement for noisy objective functions is used to mitigate the many sources of noise in computer systems.

There are hyper-parameters in the external libraries, such as the initial hyper-parameters of the GPyTorch kernel, the number of Monte-Carlo samples in the BoTorch acquisition function, and the number of restarts in BoTorch's multi-restart acquisition function. The libraries provide their own sensible default values and my auto-tuner uses these defaults.

4.8 Summary

This projected has contributed a new implementation of DAG models, a BO surrogate model originally proposed by Dalibard et al. [11]. A DAG model uses a system's metrics to help predict its overall performance, which overcomes the curse of dimensionality because each metric depends on a small number of configurable variables and other metrics. Each metric is modelled by a Gaussian process with optional mean. These sub-models are fitted independently, and do prediction sequentially, keeping the DAG efficient despite its large size. My implementation has added novel features to support first-order deterministic optimisation techniques and batch processing on a GPU.

My implementation uses deterministic first-order optimisation by default to fit the model and optimise the acquisition. This is much faster than Bayesian inference and less sensitive to its hyper-parameters than stochastic methods. To support this optimiser, the method to create the posterior distribution has been redesigned. The mixture of posterior distributions, created from a batch of arc samples, is now approximated as a Gaussian of the same mean and variance. This produces a posterior distribution which supports the re-parametrisation trick to make acquisition optimisation more stable; auto-grad to enable the use of first-order optimisation; and batch processing to reduce computation time on a GPU.

Chapter 5

Evaluation

This project has made the following contributions

1. A BO tuner with DAG surrogate models which supports model fitting and acquisition optimisation using
    - deterministic, first-order optimisation techniques, supported by a novel method to generate the DAG's posterior distribution.
    - batch processing on a GPU.
2. A performance model for Spark, incorporated into a DAG model.
3. A methodology to create DAG models for other systems, and techniques to overcome practical issues applying the tuner to real-world problems.
4. A performance improvement for a Spark benchmark of 10% over its default configurations using only 40 evaluations.

These contributions have been confirmed by evaluating my DAG tuner over the Spark case study and by comparing it against two baseline tuners. The performance of each tuner was evaluated using two criteria: firstly, how optimal is the best configuration found; and secondly, how long has it taken to find this configuration. The baseline tuners were random search [8] and BO using a Gaussian process as the surrogate model (GP) [6].

The DAG tuner found a configuration 10% faster than Spark's default configuration after evaluating 40 configurations. In comparison, the random tuner found a configuration 8% faster after 60 iterations, while the GP tuner also found a configuration 10% faster after 40 iterations but with more variance across repeat experiments. The DAG tuner had higher computation cost than GP, but this was mitigated in the Spark

case study using pipelining, and the DAG tuner's computation time could have been reduced by using a GPU.

5.1 Set-up

Benchmark The objective function used in this experiment was the `SQL/Aggregation` benchmark from HiBench [22], executed on a one-machine cluster. This benchmark aggregates a table of data into groups, which is familiar operation in big-data pipelines such as creating statistics from a batch of data. It uses high memory and exchanges intermediate data between executors so it can be bottlenecked by disk spills and inter-process communication. As discussed in Section 3 this benchmark was chosen to be a single, simple application on a single, simple cluster to reduce the proportion of project time spent gathering expert knowledge.

This benchmark is long running, taking a minimum of 8 minutes, which motivates the need for a sample-efficient tuner. It also permits use of a computationally expensive tuner such as the DAG tuner because the tuner's computation of the next configuration can be pipelined with Spark's evaluation of the previous point. Furthermore, the best configuration found for this benchmark improved its performance by 10% over Spark's default configuration, and 70% over the worst configuration, motivating the need to tune this benchmark.

Configurable variables and objective The performance metric optimised in this case study was throughput. `SQL/Aggregation` processes and aggregates a table of records in a database and throughput is the number of records divided by the job length. It is important to optimise throughput in order to finish the job faster and free resources for other jobs.

Spark has 30 configurable variables but this project was to limited to tuning six of them to reduce time spent creating a performance model. The six are shown in Table 3.1, and were chosen because existing research [13, 20, 27, 37] has shown that most Spark applications are sensitive to them. With more time, the same methodology could extend the performance model to more configurable variables.

DAG tuner My DAG tuner used the Spark DAG model from Section 3.5 as the surrogate model for BO. All the hyper-parameters for the DAG model and the BO loop were set to default, as specified in Section 4.7, and Spark's variables and metrics were transformed as per Section 3.3 before they were passed into and out of the tuner.

Baseline tuners The DAG tuner was compared against two baseline tuners. Section 6 discusses additional tuners qualitatively but those were not used in the evaluation because of constrained time and resources. The first baseline was random search, which Bergsta et al. [8] have identified as a good baseline tuner. By outperforming random search, the DAG model has shown that it directs the search towards promising areas of the configuration space.

The second baseline tuner, GP, also uses BO, except that its surrogate model is a Gaussian process. Outperforming the GP tuner has demonstrated the benefit of using the Spark DAG model as the surrogate model instead of a Gaussian process. The GP tuner was taken from BoTorch [6] and had all hyper-parameters set to BoTorch's defaults. Again, Spark's variables and metrics were transformed as per Section 3.3 because the GP tuner suffers exactly the same issues applying a mathematical optimisation technique to a concrete problem like Spark.

Initialisation Model-based optimisation techniques require a set of initial observations to bootstrap the model. All convergence experiments were bootstrapped with the same ten observations of ten configurations generated using a quasi-random Sobol sequence. This number was a trade-off between coverage of the configuration space and wasted iterations for the tuners to optimise. Ten was chosen to mimic previous research [3, 6] which used $2m+n+2$ configurations, where m is the number of continuous variables and n is the number of discrete variables with a narrow range. A Sobol sequence was used instead of a random set for better coverage of the configuration space.

Hardware The CPU used was a 8-core Intel `i7-3770` with 16GB of DRAM running Ubuntu `20.04`. The GPU used was an NVIDIA Tesla K80 with 12GiB of GDDR5 memory running CUDA `11.2`. In the convergence experiments, all tuners were run on the CPU due to limited access to the GPU. The computational requirements of each tuner were measured on both the CPU and GPU.

5.2 Convergence

The tuners were compared on how well they converge to an improved configuration for the `SQL/Aggregation` benchmark, both on how optimal this configuration is and how many configurations they evaluate before converging. Figure 5.1 shows the convergence of each tuner up to 60 evaluations, starting after the ten bootstrap evaluations.

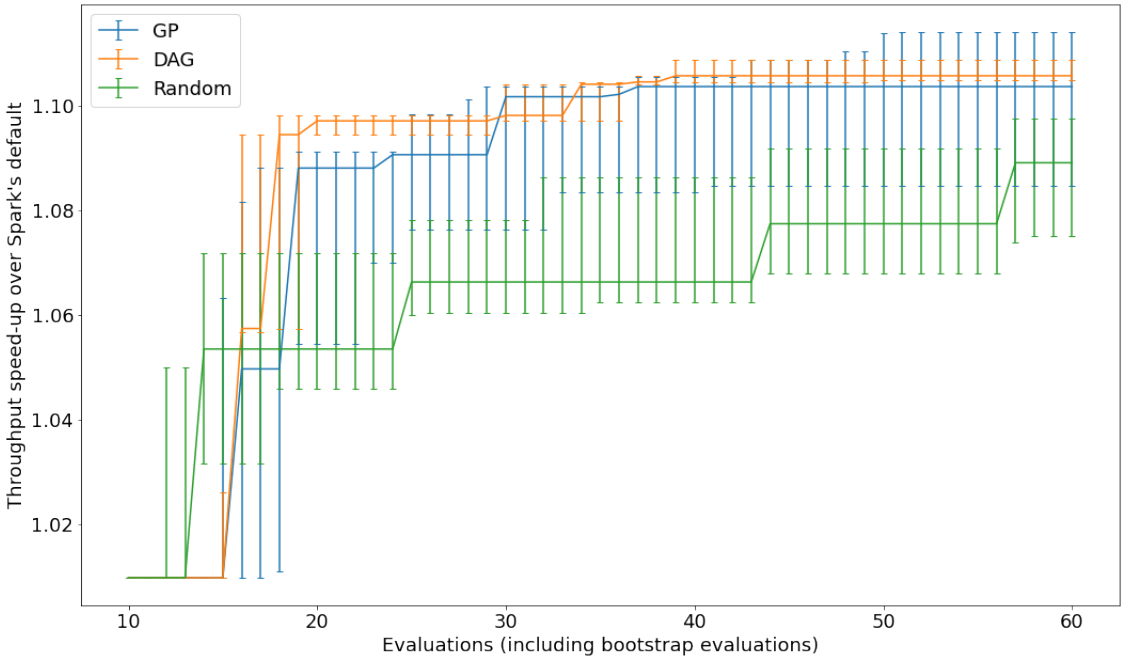

Figure 5.1: Convergence rate of each model: the best configuration so far, after each evaluation. 5 repeat experiments with median and inter-quartile range shown.

Each experiment was repeated 5 times and the median and inter-quartile range are graphed.

The DAG tuner was the best performing tuner and it found a much better configuration than Spark's default configuration. It found a configuration after 40 observations (including bootstrap observations) which improvement throughput by 10% over Spark's default configuration. In comparison, the median speed-up achieved by the GP tuner was also 10% in 40 observations, however, the inter-quartile range of the GP tuner was much wider than that of the DAG tuner. This implies that the DAG model is more resilient to random noise from the Spark observations and to its internal random seeds, and this makes it a more reliable tuner. Lastly, both the DAG tuner and the GP tuner far exceeded the performance of random search, demonstrating that both types of model successfully directed the search to promising areas of the configuration space.

There are many reasons why these results are exciting. Spark is a complex system with a large, mature codebase, yet an accurate performance model has still been possible using a typical systems analysis. This indicates that the same methodology can be applied to many other systems used by systems researchers instead of just theorists. Furthermore, these results have shown that the Spark DAG model is resilient to difficult issues in the case study like categorical variables and non-Gaussian metrics.

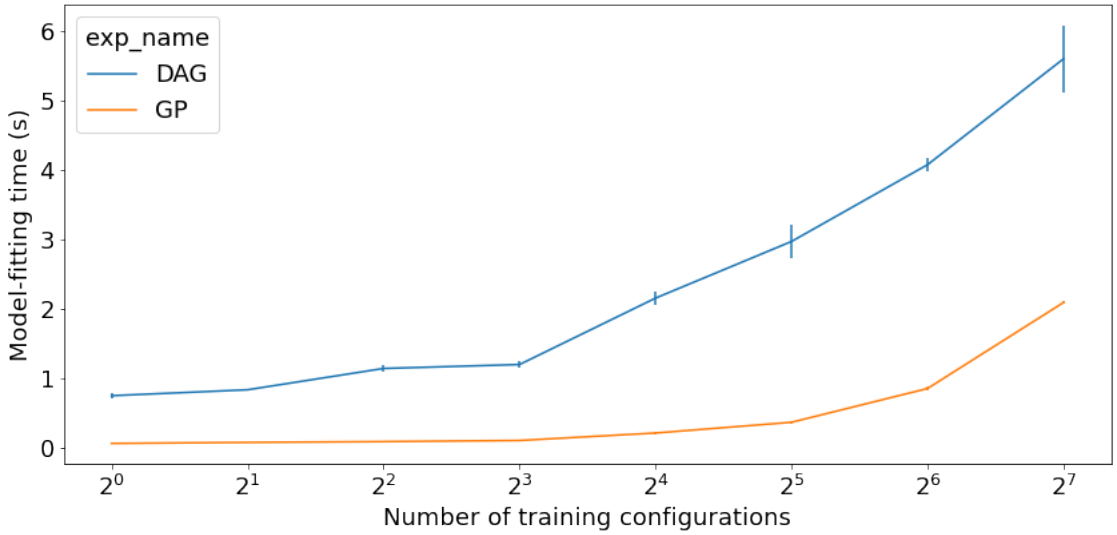

Figure 5.2: How the CPU model-fitting times of the DAG and GP models scale with the number of training configurations. 5 repeats with mean and standard deviation shown.

The case study has also shown that DAG models can mitigate the curse of dimensionality because none of the sub-models in the Spark DAG models have more than four inputs, so input space of each sub-model can be covered with fewer evaluations. The empirical results show that this is a useful property over six dimensions, and it is likely to become more important on case studies with even more dimensions where Gaussian processes begin to struggle.

5.3 Computational requirements

In the Spark case study, the DAG tuner required more computation time than the GP tuner to choose the next configuration, but this did not increase total tuning time because this computation was pipelined with the evaluation of the previous configuration. Pipelining only prevents an increase in tuning time if the tuner's computation time is less than the objective function's evaluation time. This section empirically evaluates the computation time of the DAG and GP tuners to show that DAGs can also be used on faster case studies. The experiment compared the computation times of the DAG and GP models on data from the Spark benchmark, how they scale as the number of training configurations increases, and how they are accelerated on a GPU using batch processing.

Computing the next configuration is separated into two steps, fitting the model on the previous observations and optimising the acquisition function to generate the next

configuration. Figure 5.2 shows how the model-fitting times of the Spark DAG and GP models increased with the number of training configurations. In this experiment, the sub-models of the DAG were fitted sequentially, although the DAG's fitting time could be reduced further by fitting them in parallel.

Both models' fitting times had similar scaling properties, increasing exponentially as the configurations double. The DAG added a clear overhead, although this overhead reduced as the number of configurations increased. This may be because each node of the Spark DAG model has lower dimensionality than the GP, and this benefit began to outweigh the additive cost of fitting each sub-model when fitting many configurations.

The acquisition-optimisation time is shown in Figure 5.3. The DAG tuner was compared to the GP tuner, both with one arc sample to show the minimum possible acquisition-optimisation time, and also with 64 arc samples (the default number). Even with one arc sample the DAG imposed a clear overhead because of the additive cost of making predictions from multiple sub-models. Using 64 arc samples then increased the peak CPU acquisition-optimisation time from 8s to 350s because acquisition-optimisation time increases linearly with the number of arc samples (see Figure 4.6). However, batch processing on a GPU is very beneficial for long acquisition-optimisation times, and using a GPU reduced acquisition-optimisation time by a factor of ten on the DAG with 64 arc samples.

It is also interesting to mention how the acquisition-optimisation time scaled with the number of training configurations. Across all models time decreased between 32 and 64 points, and this might be because there was sufficient information to quickly identify an optima. This behaviour might be specific to my case study, or might indicate that acquisition-optimisation time has an upper bound.

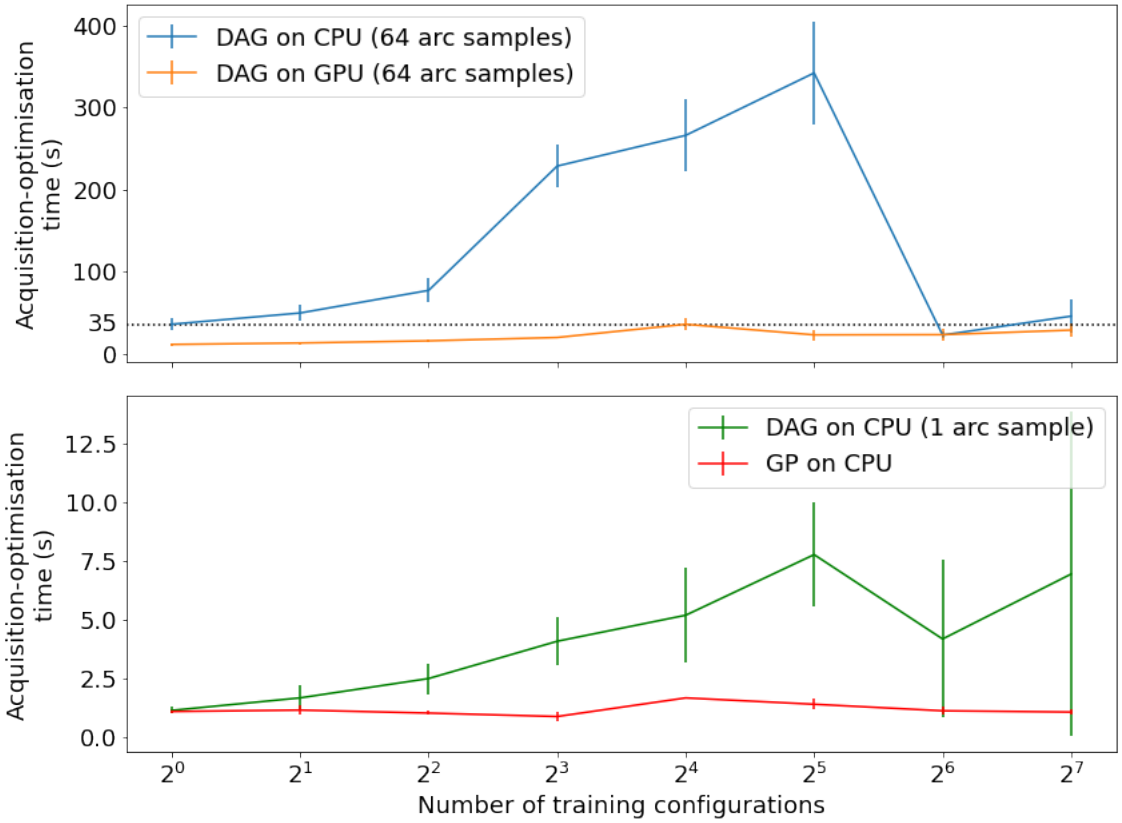

Figure 5.3: How the acquisition-optimisation time of each model scales with the number of training configurations. 5 repeat experiments with mean and standard deviation shown.

Chapter 6

Related work

There has been considerable recent research in tuning computer systems. Spark is regularly used as a case study, so this work is of particular relevance. There have also been successes in tuning other systems and the methods used in those works can be compared to this project.

Manual Spark tuning The most basic tuning resource, and the mostly widely used by Spark users, is Spark's tuning hint sheet [18]. Gounaris and Torres [20] and Petridis et al. [37] also propose methodologies to do manual tuning more quickly. These methodologies have achieved 20% speed-ups over Spark's default configuration on some benchmarks, which motivates the need for tuning, but manual methods inherently place a burden on the user, require the user to learn some expert knowledge of the system and aren't transferable to other systems.

Automated Spark tuning Huawei [49] present an automated tuner for Spark (and other systems) which uses divide-and-diverge sampling to explore the configuration space, and recursive-bound-and-search to exploit the most promising areas. They also try a model-based method using Gaussian processes, but show that it struggles to make accurate performance predictions because of Spark's high dimensionality. Fekry et al. [13] tune Spark using BO with Gaussian processes, but over a smaller number of configurable variables. The first stage of their tuning pipeline is sensitivity analysis which cuts out any insignificant variables. Unlike Fekry et al. DAG models mitigate the curse of dimensionality while also retaining all configurable variables. Sensitivity analysis could be used in future work to help build DAG models automatically by trimming arcs between metrics that do not depend on each other.

Performance models of Spark Venkataraman et al. [42] build an expert performance model for Spark which more accurately predicts its performance than a Gaussian process. Rather than tuning iteratively, this model is trained once with many random configurations on a smaller Spark workload, then it predicts which configuration will be fastest for the production workload. In future, this work could be combined with mine using multi-fidelity optimisation [24], an iterative technique which takes advantage of different evaluation functions, some of which are cheaper to evaluate but less insightful. Kunjir and Babu [27] present an expert model for Spark which is used to enhance BO with Gaussian processes. Like DAG models, their model uses Spark's metrics to help predict the overall performance, but unlike a DAG they have no way to predict the value of run-time metrics, so they are limited to using metrics which can be deterministically predicted at compile time. A DAG model can include all of Spark's metrics and therefore incorporate more expert knowledge.

Wider systems tuning There have also been successes for automated tuning techniques in other computer systems. Alipourfard et al. [3] use BO to select the best cloud configuration for a given workload, and Hsu et al. [21] improve this work by using the VM's metrics to predict the overall performance. This is the same strategy as used by Kunjir and Babu [27] and by BOAT [11], but Hsu et al. can only use metric values from previously run configurations whereas DAG models are able to use the predicted values of the new configuration. Recently, Alabed and Yoneki [2] have also used a system's run-time metrics to tune it more efficiently. They tune RocksDB using multi-task Gaussian-process BO [10]. Multi-task learning simultaneously optimises multiple objectives in the same configuration space, which can be more efficient than single-objective optimisation because the multi-task surrogate model learns correlations between the objective functions. This is a similar idea to DAG models, except that DAGs explicitly defines the dependencies between metrics whereas a multi-task model assumes they may all depend on each other.

Reinforcement learning Reinforcement learning has been used by Li et al. [28] and Mirhoseini et al. [33, 31, 32] to optimize database queries, scheduling of neural networks, and chip placement in hardware design. Reinforcement learning approaches require thousands of evaluations to learn policies, so these approaches are only suitable for very short-running systems. A lot of real-world systems, especially those distributed across multiple machines, have long start-up times for each evaluation that make them unsuitable to reinforcement learning.

Chapter 7

Conclusion

7.1 Future work

Heterogeneous sub-models Some of the metrics in the Spark case study did not follow a Gaussian distribution, so modelling them with Gaussian processes added uncertainty to the model. The Tree-Parzen estimator [7], among others, can fit more general distributions so adding support for multiple types of sub-model would reduce uncertainty in DAGs.

DAG architecture search Designing a DAG is heavy human investment which is worthwhile when the DAG is reused on many optimisation problems, such as a single DAG model for all Spark programs. This human cost would be avoided if the DAG were designed automatically from training data and would make DAGs applicable to one-off problems. This is a similar idea to neural architecture search [50].

Multi-fidelity optimisation My auto-tuner learns by repeatedly evaluating the objective application on a production-scale workload, but some knowledge could be gained in less time by evaluating a small-scale training workload instead. Venkataraman et al. [42] successfully learn a performance model exclusively from training workloads, and multi-fidelity BO [24] extends this further by learning from both training and performance workloads. Alternatively, the training workloads could be used for DAG architecture search and the performance workloads used for fitting the DAG.

7.2 Summary of work

This project has presented a novel auto-tuner for computer systems and shown it outperforming baseline tuners on a concrete case study of Spark. The tuner uses Bayesian optimisation (BO) with directed-acyclic-graph (DAG) surrogate models. My new implementation of DAG models supports deterministic, first-order optimisation techniques to fit the model and optimise the acquisition. It also supports batch processing on a GPU to reduce the computation time of the tuner. This has been implemented using a novel method to generate a DAG's posterior distribution and by leveraging libraries from the BoTorch [6] software stack.

The efficiency and practicality of my tuner has been demonstrated on a case study of Spark. This case study demonstrated a methodology to create a DAG performance model and overcome practical issues such as categorical variables. The methodology is accessible to systems researchers and could be reused on a wide range of systems-optimisation problems in future. Researchers can quickly include DAG models in their own work because my implementation is available as a BoTorch module.

The DAG tuner optimised six Spark configurable variables to provide a 10% throughput speed-up over Spark's default configuration using only 40 evaluations. This was the same speed-up and tuning time as achieved by BO with a Gaussian process, but the DAG tuner had a narrower error range across repeat experiments. This demonstrates its additional resilience to random noise from Spark. Random search only achieved an 8% speed-up in 60 iterations, which shows that the DAG model actively directed the search to promising areas.

The DAG tuner required more computation time than the GP tuner, although pipelining prevented this from increasing the total tuning time in the Spark case study. This computation time was reduced by a factor of 10 on a GPU meaning that the DAG tuner can also tune shorter-running systems efficiently.

Due to the curse of dimensionality, BO with Gaussian processes struggles on problems with more than ten configurable variables [45]. This project only evaluated a case study with six configurable variables but it has indicated that DAGs can mitigate the curse of dimensionality. No sub-model in the Spark DAG model had more than four input dimensions, and DAGs for other systems should also be low-dimensional using the methodology from Section 3. This produced strong convergence results on a six-variable problem and should become more important on high-dimensional problems where Gaussian processes struggle.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Sami Alabed and Eiko Yoneki. High-dimensional Bayesian optimization with multi-task learning for RocksDB. *arXiv preprint arXiv:2103.16267*, 2021.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big-data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[5] The GPyOpt authors. GPyOpt: A Bayesian optimization framework in Python. `http://github.com/SheffieldML/GPyOpt`, 2016.

[6] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. BoTorch: A framework for efficient Monte-Carlo Bayesian optimization. In *Advances in Neural Information Processing Systems 33*, 2020.

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (NIPS 2011)*, volume 24. Neural Information Processing Systems Foundation, 2011.

[8] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[9] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[10] Edwin V Bonilla, Kian Chai, and Christopher Williams. Multi-task Gaussian-process prediction. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008.

[11] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, pages 479–488, 2017.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. To tune or not to tune? In search of optimal configurations for data analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2494–2504, 2020.

[14] Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org/`, 2021.

[15] Apache Software Foundation. Apache Spark configuration. `https://spark.apache.org/docs/latest/configuration.html`, 2021.

[16] Apache Software Foundation. Apache Spark documentation. `https://spark.apache.org/docs/latest/index.html`, 2021.

[17] Apache Software Foundation. Apache Spark moniting. `https://spark.apache.org/docs/latest/monitoring.html`, 2021.

[18] Apache Software Foundation. Apache Spark tuning. `https://spark.apache.org/docs/latest/tuning.html`, 2021.

[19] Jacob Gardner, Geoff Pleiss, Kilian Q Weinberger, David Bindel, and Andrew G Wilson. GPyTorch: Blackbox matrix-matrix Gaussian-process inference with GPU acceleration. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[20] Anastasios Gounaris and Jordi Torres. A methodology for Spark parameter tuning. *Big data research*, 11:22–32, 2018.

[21] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. Arrow: Low-level augmented Bayesian optimization for finding the best cloud VM. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 660–670. IEEE, 2018.

[22] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[23] Facebook Inc. Ax. `https://github.com/facebook/Ax`, 2021.

[24] Kirthevasan Kandasamy, Gautam Dasarathy, Jeff Schneider, and Barnabás Póczos. Multi-fidelity Bayesian optimisation with continuous approximations. In *International Conference on Machine Learning*, pages 1799–1808. PMLR, 2017.

[25] Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[26] Nicolas Knudde, Joachim van der Herten, Tom Dhaene, and Ivo Couckuyt. GPflowOpt: A Bayesian optimization library using TensorFlow. *arXiv preprint – arXiv:1711.03845*, 2017.

[27] Mayuresh Kunjir and Shivnath Babu. Black or white? How to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1667–1683, 2020.

[28] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[29] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomForest. *R news*, 2(3):18–22, 2002.

[30] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[31] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[32] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*, 2020.

[33] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[34] Willie Neiswanger, Kirthevasan Kandasamy, Barnabas Poczos, Jeff Schneider, and Eric Xing. ProBO: Versatile Bayesian optimization using any probabilistic programming language. *arXiv preprint arXiv:1901.11515*, 2019.

[35] Oracle. The parallel collector. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html`, 2021.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learn-

ing library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[37] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data*, pages 226–237. Springer, 2016.

[38] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[39] Carl Edward Rasmussen and CK Williams. Gaussian processes for machine learning, vol. 1, 2006.

[40] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[41] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944*, 2012.

[42] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[43] Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer, 2005.

[44] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020.

[45] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

[46] Alex Woodie. A decade later, Apache Spark still going strong. `https://www.datanami.com/2019/03/08/a-decade-later-apache-spark-still-going-strong/`, 2019.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[48] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[49] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.

[50] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.