BOAT: Building Auto-Tuners with Structured Bayesian Optimization

Valentin Dalibard Michael Schaarschmidt Eiko Yoneki

Computer Laboratory
University of Cambridge
firstname.lastname@cl.cam.ac.uk

ABSTRACT

Due to their complexity, modern systems expose many configuration parameters which users must tune to maximize performance. Auto-tuning has emerged as an alternative in which a black-box optimizer iteratively evaluates configurations to find efficient ones. Unfortunately, for many systems, such as distributed systems, evaluating performance takes too long and the space of configurations is too large for the optimizer to converge within a reasonable time.

We present BOAT, a framework which allows developers to build efficient bespoke auto-tuners for their system, in situations where generic auto-tuners fail. At BOAT's core is structured Bayesian optimization (SBO), a novel extension of the Bayesian optimization algorithm. SBO leverages contextual information provided by system developers, in the form of a probabilistic model of the system's behavior, to make informed decisions about which configurations to evaluate. In a case study, we tune the scheduling of a neural network computation on a heterogeneous cluster. Our auto-tuner converges within ten iterations. The optimized configurations outperform those found by generic auto-tuners in thirty iterations by up to $2\times$.

1. INTRODUCTION

From the number of machines used in a distributed application, to low-level parameters such as compiler flags, managing configurations has become one of the main challenges faced by users of modern systems [6]. Recently, auto-tuning has successfully been applied to address this challenge in a range of areas [26, 3, 4]. Users employ it by exposing their configuration parameters and a performance metric of their application to a black box optimizer which will evaluate many configurations. A popular auto-tuning technique is Bayesian optimization [5], which uses the results to incrementally build a probabilistic model of the impact of the parameters on performance. This allows the optimization to quickly focus on efficient regions of the configuration space. Unfortunately, for many systems, either the configuration space is too large to develop a good model, or the time to evaluate performance is too long to be executed many times.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052662

The problem addressed in this paper is automatically finding efficient configurations in situations where black box optimizers fail. We present **BOAT**, a framework that allows a system developer to build a **B**esp**O**ke **A**uto-**T**uner for their system. The key idea of this work is to allow developers to provide contextual information in the form of a probabilistic model reflecting the system's behavior. At BOAT's core is structured Bayesian optimization (SBO), our extension of the Bayesian optimization algorithm capable of leveraging these probabilistic models to make informed decisions about which configurations to evaluate. Our approach drastically reduces the number of iterations needed to converge, making auto-tuning applicable in a range of new areas.

System developers implement probabilistic models using BOAT's *probabilistic programming* library. Probabilistic programming has recently been proposed as an intuitive way to construct structured probabilistic models [17]. In BOAT, developers use it to expose their understanding of system's executions. Throughout the optimization, runtime measurements from the system are used to perform inference on the model and make it resemble the true underlying behavior.

For example, in our neural network case study, we built an auto-tuner which balances a computational load across a heterogeneous cluster. Our associated model iteratively infers the computational power of machines and uses it to predict a machine's computation time as a function of its assigned load. Similarly, a different part of the model predicts communication time based on the inferred network speed of each machine. Our contributions are:

- We propose Structured Bayesian Optimization (SBO), a novel extension of Bayesian optimization capable of leveraging bespoke probabilistic models to rapidly converge to high-performance configurations (Section 3).
- We present BOAT, a framework to build bespoke auto-tuners (Section 4). BOAT includes an implementation of SBO as well as a probabilistic programming library. We discuss the design of useful probabilistic models in the context of BOAT (Section 5).
- We demonstrate the use of BOAT through two case studies. First, a garbage collection case study (3 parameters) in which we tune the configuration flags of a database to minimize its tail latency. This is a domain simple enough for generic auto-tuners to be amenable. We present it here to show that BOAT brings convergence improvements in these simple settings and to illustrate the construction of probabilistic models. Second, a neural network case study (30+ parameters) in which we optimize the distributed scheduling of a neural network computation on

a heterogeneous cluster. We show that our bespoke auto-tuners converge within ten iterations. Our optimized configurations are up to 2.9× faster than simple configurations an application user may have selected, and 2× faster than the ones found by generic auto-tuners after thirty iterations (Section 6).

2. MOTIVATION

2.1 The need for auto-tuning in systems

Due to the diversity of workloads, modern systems have no "one size fits all" approaches to best execute their computation. They therefore expose many configuration parameters which users must manually tune to maximize performance. This is a difficult task as it requires an understanding of the underlying system execution. Furthermore, changes in the hardware or the workload being executed will affect the system's behavior thus requiring new adjustments.

In this paper, we consider the task of automatically tuning configuration parameters associated with system performance. This can include any parameter for which the optimal value is dependent on the hardware or software context, ranging from low level parameters such as compiler flags or configuration files, to higher level ones like the number of machines used in a computation.

Ideally, generic auto-tuners could be applied to these tasks and determine high performance configurations in reasonable time. In practice, this is rarely the case for two reasons. First, measuring performance is often expensive. In distributed systems, it will take at least a few minutes to get an accurate measure of the "objective function" being tuned. This discards the evolutionary or hill climbing approaches used by auto-tuners like Petabricks [3] or OpenTuner [4], as they usually require thousands of evaluations.

Second, the configuration space of most real world problems is too complex. For example, in our neural network case study, the computation is scheduled on up to ten machines. The configuration space includes at least three parameters per machine, for a total of over thirty parameters. This is too large for the traditional Bayesian optimization approach which is used by optimizers like Spearmint [26] and SMAC [19]. Bayesian optimization tends to require fewer iterations to converge than other optimizers, but fails to work in problems with many dimensions (more than 10) [25].

2.2 Garbage collection case study

As an example, we briefly present our garbage collection case study. The goal is to tune the Garbage collection (GC) flags of a Java Virtual Machine (JVM) based database to minimize its 99th percentile latency. We tune three parameters of the Concurrent Mark Sweep (CMS) collector: the *young generation size* and *survivor ratio* flags, which govern the size of the different sections of the heap, and the *max tenuring threshold* which sets the rate at which objects are promoted between heap sections. We measure the 99th percentile latency of Cassandra [28], a popular JVM-based wide-column store, using the YCSB cloud benchmarking framework [11] with a variety of workloads. The details and results of our experiments are presented in section 6.1.

The behavior of the garbage collection has a high impact on the latency. In one context, setting appropriate values of these three flags reduces the 99th percentile latency from 19ms using Cassandra's default values, to 7ms. This is mostly due to *minor collections* which frequently collect objects in the young generation section of the heap, provoking a "stop-the-world" pause which halts the application. Good configurations will minimize the average duration of these collections as well as their total time.

The small domain of this tuning problem means off-the-shelf auto-tuners are still applicable. In our evaluation, we find Spearmint converges to good configurations in 16 iterations, after four hours of auto-tuning time (15 minutes per evaluation). However, leveraging contextual information can significantly reduce convergence time. Our auto-tuner implemented in BOAT is simple yet converges to within 10% of the best found performance by the second iteration. We use it for illustration throughout this paper. Our neural network case study tackles a more complex tuning problem, with over thirty dimensions, in which case off-the-shelf auto-tuners fail to find good values after thirty iterations.

Algorithm 1 The Bayesian optimization methodology

```
Input: Objective function $f()$
Input: Acquisition function $\alpha()$
1: Initialize the Gaussian process $G$
2: for $i = 1, 2, \dots$ do
3:     Sample point: $\mathbf{x}_t \leftarrow \arg\max_{\mathbf{x}} \alpha(G(\mathbf{x}))$
4:     Evaluate new point: $y_t \leftarrow f(\mathbf{x}_t)$
5:     Update the Gaussian process: $G \leftarrow G \mid (\mathbf{x}_t, y_t)$
6: end for
```

3. STRUCTURED BAYESIAN OPTIMIZATION

3.1 The Bayesian optimization algorithm

We start by reviewing the Bayesian optimization methodology. Bayesian optimization [5] attempts to find the minimum of some objective function $f(\mathbf{x})$ where typically $\mathbf{x} \in \mathbb{R}^D$. To this end, it incrementally builds a probabilistic model which reflects the current knowledge of the objective function. Most of the time, a Gaussian process (GP) is used [22]. GPs are a powerful class of models which can be viewed as an elaborate interpolation tool. Given a dataset of multi-dimensional inputs $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_t\}$ and corresponding real valued observation $\boldsymbol{Y} = \{y_1, y_2 \dots y_t\}$ their prediction for a new input $\mathbf{x}$ is normally distributed.

The overall procedure of the optimization is shown in Algorithm 1. It executes three steps each iteration. First, it performs a numerical optimization to find a point in the configuration space which maximizes an *acquisition function.* Acquisition functions measure how promising a point $\mathbf{x}$ is based on the distribution predicted by the GP at $\mathbf{x}$. They trade-off exploration and exploitation. For example, a popular acquisition function is the *expected improvement* which returns the expected value of the improvement brought by evaluating $f(\mathbf{x})$ over the best value η found so far:

$$\alpha_{EI}(\mathbf{x}) = \mathbf{E}(\max(0,\, f(\mathbf{x}) - \eta))$$

Finding a point $\mathbf{x}_t$ which maximizes the acquisition function can be performed using an off-the-shelf numerical optimization algorithm. Second, the optimization measures the output of the expensive objective function at this point. Third, it updates the Gaussian process with this new measurement.

When compared with other optimization methods, such as evolutionary algorithms, Bayesian optimization tends to converge in fewer iterations. This however comes at the cost of a high overhead per iteration due to the computational complexity of performing the numerical optimization.

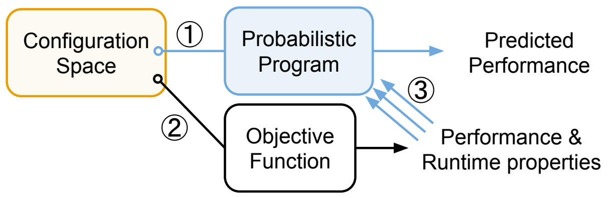


Figure 1: Procedure of Structured Bayesian Optimization

Recently, Bayesian optimization has successfully been used to tune the parameters of machine learning programs [26] and surpass human experts. However, it has so far been unsuccessful at tackling optimizations in high dimensional spaces [25]. This can be attributed to two issues which may occur:

1) The probabilistic model fails to accurately capture the objective function landscape after a reasonable number of iterations. This is due to the *curse of dimensionality*.

2) The numerical optimization algorithm, used in each iteration, fails to converge and find a promising point.

In this work, we extend Bayesian optimization to tackle the first of these issues. In practice, we found both may need to be addressed and our neural network case study also involved improvements to help the numerical optimization converge, but these are out of the scope of this paper.

3.2 Structured Bayesian optimization

Structured Bayesian optimization (SBO) extends Bayesian optimization methodology to take advantage of the known structure of the objective function. It uses a *structured probabilistic model*, provided by the developer, instead of a simple Gaussian process. In BOAT, those models are implemented in our probabilistic programming library.

Figure 1 shows the overall procedure of a structured Bayesian optimization. It is similar to the one of a traditional Bayesian optimization and performs three steps each iteration: (1) It looks for a point in the configuration space with high predicted performance by the probabilistic program. (2) It evaluates the best found point using the objective function and collects runtime measurements (3) It performs inference on the probabilistic program using the resulting observations.

When compared with traditional Bayesian optimization, using a bespoke probabilistic models brings two advantages. First, it captures the user's understanding of the behavior of the system. This drastically reduces the number of iterations needed for the model to converge towards the true objective function. In the context of BOAT, models are implemented in probabilistic programming and can reason about arbitrary data structures – like regular programs – and reproduce complex behaviors. For example, the model in our neural network case study predicts the individual computation time of each machine in a distributed cluster. The total time is predicted to be the maximum of individual times plus a communication cost. It would take many evaluations for a Gaussian process to accurately model the function *max* over multiple inputs, our model does so by default.

Second, using such a model allows us to monitor runtime properties reflected in the model and use them for inference. For instance, in our garbage collection case study, our model predicts the number and average duration of minor collections. After each experiment we parse the garbage collection logs to observe their true value and use them for inference.

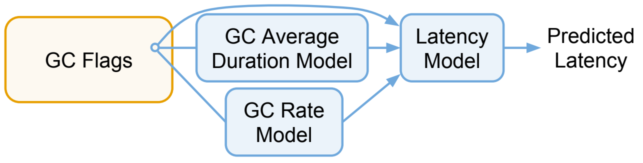


Figure 2: Dataflow of our garbage collection model

The next subsection presents an example of a structured probabilistic model from our garbage collection case study.

3.3 A probabilistic model for GC

The probabilistic model written by the developer should take as input a configuration and predict its performance. Initially, developers should use a generic probabilistic model, effectively running traditional Bayesian optimization, and observe whether the convergence time is acceptable. If it is not, BOAT allows developers to incrementally add *structure* to the model to reduce convergence time. Adding structure is done by making the probabilistic model more similar to the behavior of the system. In the following, we illustrate how this was achieved in our garbage collection case study.

As an initial model, we used a Gaussian process. The GP predicted 99th percentile latencies based on the flag values. This took many iterations to converge, despite the simplicity of the problem. To add structure, we included in the model a notion of *rate* and *average duration* of minor collections. Given flag values, our model predicted both these statistics. It then predicted the latency as a function of the flag values and the statistics. The data flow of our model is shown in Figure 2. When using the model in BOAT, we collected the true value of these statistics from the GC logs after each evaluation and used them for inference. Further, we declared how we believed each of the three model components behaved as a function of their inputs. For example, we noticed the rate of minor collections was inversely proportional to the size of the *eden* memory region in the JVM-heap, which is where objects are initially allocated. This intuition was included in the corresponding model by building a *semi-parametric model* (Section 5.1), which can successfully combine a user-defined trend with empirical data.

In practice, adding only little structure can be sufficient to make the optimization converge in a reasonable time. This is useful as simpler models are able to adapt to a broad variety of behaviors, such as widely different hardware and workloads. This allows the construction of bespoke auto-tuners providing global performance portability.

4. THE BOAT FRAMEWORK

BOAT allows a developer to build a bespoke auto-tuner for their system via SBO. Figure 3 shows the flow of data to constuct and use an auto-tuner.

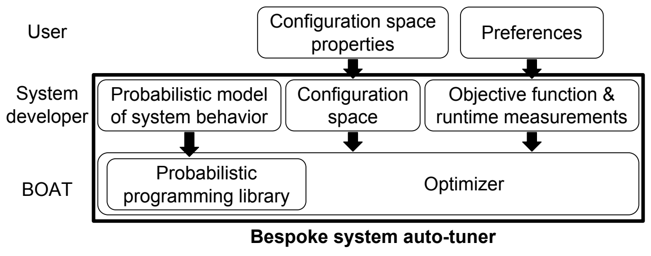
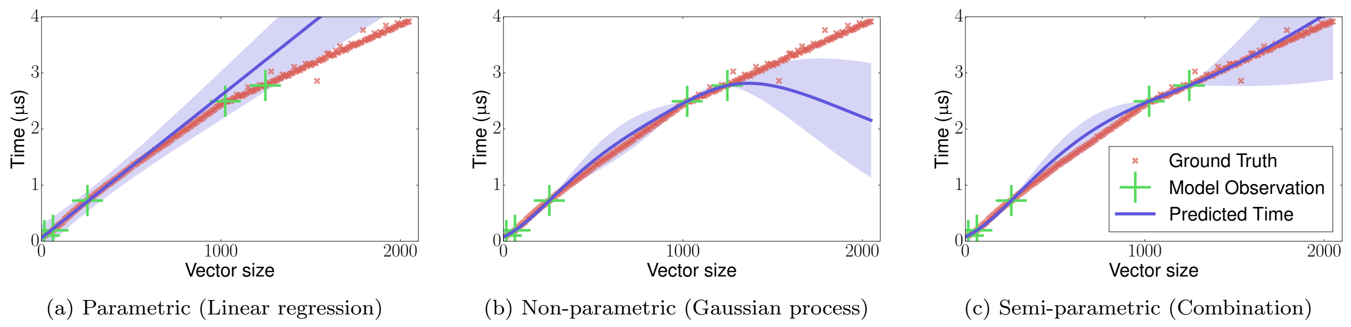

User
Configuration space properties
Preferences
System developer
Probabilistic model of system behavior
Configuration space
Objective function & runtime measurements
BOAT
Probabilistic programming library
Optimizer
Bespoke system auto-tuner

Figure 3: Flow of data when using BOAT



(a) Parametric (Linear regression) (b) Non-parametric (Gaussian process) (c) Semi-parametric (Combination)

Figure 4: Three models predicting the time to insert an element into a sorted vector after five observations.

Application users provide two types of arguments specific to their application. First, the configuration space properties. These have an influence on the set of valid configurations. In a scheduling problem, this could be the list of available machines. Second, their preferences. These define system performance metrics. For example, a user could specify to optimize throughput, tail latency, or set the workload with which the system should be evaluated.

To create a bespoke auto-tuner, a system developer takes these as input to provide three types of information to BOAT:

1) **Configuration space**: The domain of the optimization.

2) **Objective function & runtime measurements**: This specifies how to evaluate a given configuration. For example, this can involve writing configuration values to a configuration file and starting a distributed system along with a benchmark. When a BOAT optimization terminates, it returns the configuration that yielded the best objective function value.

3) **Probabilistic model of system behavior**: The contextual information which allows BOAT to discard regions of low performance and quickly converge.

The first two items are common to all auto-tuners and we do not discuss them further. The next section discusses the design of probabilistic models in the context of BOAT.

5. PROBABILISTIC MODELS IN BOAT

Developers build bespoke auto-tuners in BOAT by declaring a probabilistic model of the system's behavior via BOAT's probabilistic programming framework. A probabilistic program is similar to a simulator. The programmer implements code mimicking the behavior of the process being modeled. The advantage of probabilistic programming is that, in the presence of empirical data, they can perform *inference* to make the simulator's behavior match the observed one. We refer the reader to [12] for a review of probabilistic programming and the details of BOAT's algorithms for probabilistic inference. This section discusses the design and implementation of models in BOAT.

There are two key techniques to building useful models. First, models should be *compartmentalized.* A model should consist of a combination of independent components with each component predicting a single observable value. For example, the garbage collection model, presented in Section 3.3, contains three independent components predicting the rate and duration of collections and the 99th percentile latency. This makes models easy to debug. One can compare each component's predictions with observed values and diagnose which parts of the model fail to converge. Furthermore, this independence can be exploited by the probabilistic framework. This allows the construction of large probabilistic model without the need to pay an exponential inference cost. Section 5.2 shows how such models can be expressed in BOAT. Second, users should make each component a semi-parametric model. We discuss semi-parametric models, their benefits and their implementation in BOAT in the next subsection.

5.1 Semi-parametric models

There are two desirable properties a model should have in the context of SBO:

- It should understand the general trend of the objective function to avoid exploring low performance regions.
- It should have high precision in the region of the optimum, to find the point with highest performance.

Semi-parametric models, which we now describe, can fulfill both properties. They are a combination of *parametric* models and *non-parametric* models. As a running example, we model the average time needed to insert an element into a sorted vector as a function of its length. This has complexity $\mathcal{O}(n)$ but implementations will have runtimes affected by cache effects and other hardware properties. Figure 4 compares the predictions of a parametric, non-parametric and semi-parametric model after observing five points from the dataset. The data was obtained using the `boost::flat_set` data structure and averaged over a million runs.

Parametric models learn a fixed number of parameters. For example, simple linear regression typically learns two parameters, the slope and y-intercept. Parametric models allow developers to specify the expected behavior of the system. In our example, this means specifying that the relationship between length and time is linear and not, for example, quadratic. They however cannot fit subtleties in the data. We fit a linear regression to five data points from the sorted-vector data in Figure 4a. Although the general trend is correct, the model fails to fit all of the data points as they are not strictly linear.

On the other hand, *non-parametric* models learn an unbounded number of parameters that grows with the training data. For example, in the k-nearest neighbor algorithm each training example is memorized so it can be used for prediction. Non-parametric models provide no direct way to specify a general trend. Traditional Bayesian optimization uses Gaussian processes which are non-parametric models. We fit a GP to the same five points from the sorted-vector data in Figure 4b. It succeeds at fitting all of the data points, but fails to grasp the overall trend. In the context of

```
struct GCRateModel : public SemiParametricModel<GCRateModel> {
  GCRateModel() {
    allocated_mbs_per_sec =
     std::uniform_real_distribution<>(0.0, 5000.0)(generator);
    // Omitted: also sample the GP parameters
  }
  double parametric(double eden_size) const {
    // Model the rate as inversly proportional to Eden's size
    return allocated_mbs_per_sec / eden_size;
  }
  double allocated_mbs_per_sec;
};

int main() {
  // Example: observe two measurements and make a prediction
  ProbEngine<GCRateModel> eng;
  eng.observe(0.40, 1024);  // Eden: 1024MB, GC rate: 0.40/sec
  eng.observe(0.25, 2048);  // Eden: 2048MB, GC rate: 0.25/sec
  // Print average prediction for Eden: 1536MB
  std::cout << eng.predict(1536) << std::endl;
}
```

Listing 1: The `GCRate` semi-parametric model.

Bayesian optimization, this can lead to the over exploration of regions with poor performance. This may seems acceptable for our sorted vector example, but the number of these regions grows exponentially with the number of dimensions.

Semi-parametric models combine a parametric model and a non-parametric one. The non-parametric model is used to learn the difference between the parametric model and the observed data. In Figure 4c, we fit the sorted-vector data with a semi-parametric model that simply combines the previous two models. Predictions interpolate all data points, and correctly keep increasing with larger vector sizes. In the context of BOAT, non-parametric models require little effort from developers. Including parametric parts to describe general behavior will help the optimization to converge faster but requires some analysis. In our experience, complex parametric models designed in a narrow setting may also fail to generalize. When building a model's component in BOAT, we hence recommend to start with a non-parametric model and to keep adding structure until the optimization converges in a satisfactory time.

Semi-parametric models in BOAT. Developers extend a `SemiParametricModel` class to declare a semi-parametric model. For example, Listing 1 shows the implementation of the model predicting the rate of garbage collections from our GC case study. A `SemiParametricModel` class must define two functions:

- A constructor, which samples the values of the model parameters from their *prior* distribution. Two types of parameters must be sampled. First, the parameters of the parametric model. For example, Listing 1 samples a single parameter representing the rate at which memory is allocated. Second, the parameters of the Gaussian Process (omitted in Listing 1) which consist of one scale parameter for each input dimension and a variance parameter [22].
- A **`parametric`** function which, for a given input, returns the prediction of the parametric model.

The model can then be used by constructing a probabilistic engine `ProbEngine` templated on the model class. Data can be fed into the model via the **`observe`** function and **`predict`** returns the model's predictions in light of this data. The next subsection shows how BOAT semi-parametric models can be used in the context of a larger model.

```
struct CassandraModel : public DAGModel<CassandraModel> {
  void model(int ygs, int sr, int mtt){
    // Calculate the size of the heap regions
    double es = ygs * sr / (sr + 2.0);// Eden space's size
    double ss = ygs / (sr + 2.0);     // Survivor space's size
    // Define the dataflow between semi-parametric models
    double rate =     output("rate", rate_model, es);
    double duration = output("duration", duration_model,
                             es, ss, mtt);
    double latency =  output("latency", latency_model,
                             rate, duration, es, ss, mtt);
  }
  ProbEngine<GCRateModel> rate_model;
  ProbEngine<GCDurationModel> duration_model;
  ProbEngine<LatencyModel> latency_model;
};

int main() {
  CassandraModel model;
  // Observe a measurement
  std::unordered_map<std::string, double> m;
  m["rate"] = 0.40; m["duration"] = 0.15; m["latency"] = 15.1;
  int ygs = 5000, sr = 7, mtt = 2;
  model.observe(m, ygs, sr, mtt);
  /* Prints distributions (mean and stdev) of rate, duration
     and latency with a larger young generation size (ygs)*/
  std::cout << model.predict(6000, sr, mtt) << std::endl;
  // Print corresponding expected improvement of the latency
  std::cout << model.expected_improvement(
      "latency", 15.1, 6000, sr, mtt) << std::endl;
}
```

Listing 2: The full Cassandra latency model.

5.2 DAG models

In BOAT, `DAGModels` are used to concatenate multiple semi-parametric models into a single global model. Developers implicitly define a directed acyclic graph (DAG) of the flow of data between the semi-parametric components. Listing 2 shows the implementation of the model used in our GC case study, already shown diagrammatically in Figure 2.

There are two properties a `DAGModel` must fulfill. First, all of its semi-parametric components must be included as members of the class. For example, `CassandraModel` contains three members, one for each of its components. Second, it must define a **`model`** function, specifying the flow of data through the model. A **`model`** function takes as input a configuration. It then uses the function **`output`** to propagate data through the semi-parametric components and get their predicted outputs. Each **`output`** call is named by passing a unique string identifier as the first argument.

A `DAGModel` class can be used in a number of ways. First, **`observe`** performs inference on the model components, given a configuration and a dictionary mapping each **`output`** ID to an associated measured value. The underlying inference exploits the *conditional independence* of `DAGModels`: all components can be trained independently given their measured outputs. Second, the function **`predict`** returns the distribution of each **`output`** call for a given configuration. This is useful to debug a model, we can compare each component's prediction with the observed values. Third, one can compute the expected improvement of a configuration over a previously achieved result. Recall from Section 3.1 that expected improvement is an acquisition function used in Bayesian optimization. In SBO, each iteration of the optimization, we select the configuration which maximizes the expected improvement predicted by our structured model.

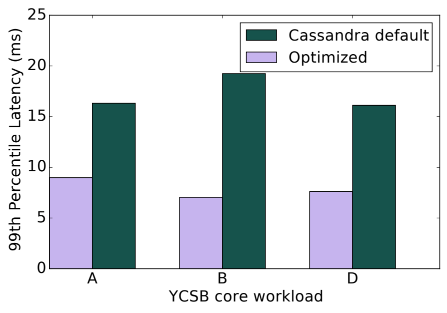


Figure 5: Results for YCSB workloads A, B and D.

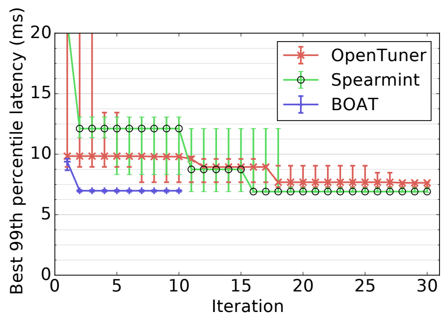


Figure 6: Convergence of the frameworks on workload B.

6. EVALUATION

We demonstrate the use of BOAT through our two case studies: 1) the garbage collection case study in which we tune configuration flags to minimize tail latency, and 2) the neural network case study in which we tune the scheduling of the training of a neural networks on a distributed system. In both, we implemented a probabilistic model of the underlying system behavior, and used it within BOAT to optimize the system configuration in a range of setting. Our evaluation focuses on quantifying two properties:

1. **The benefits of auto-tuning.** Showing that one-size-fits-all configurations yield sub-optimal performances.

2. **The need for a bespoke auto-tuner.** Showing that our auto-tuners reduce convergence time when compared to off-the-shelf optimizers. We compare our performance with OpenTuner [4] which dynamically adapts its optimization algorithm, and Spearmint [26] which implements traditional Bayesian optimization.

6.1 Garbage collection

We start by presenting the results of our garbage collection (GC) case study, as introduced in Section 2.2.

Configuration space. We tune the *young generation size*, *survivor ratio* and *max tenuring threshold* flags, of the CMS collector, which is used by default by Cassandra.

Objective function. We configured a single 8 core node to run Cassandra [28] with a 8 GB fixed heap space to model a medium-sized web application. We measure the latency using the YCSB [11] cloud benchmarking framework on a 24 core machine co-located in the same network. Each experiment was run for 15 minutes.

Model. Our probabilistic model, introduced in Section 3.3, is composed of three semi-parametric models: we predict the rate and average duration of minor collections and their impact on latency. Our analysis showed that the frequency at which major collections occurred was too low to have an impact on 99th percentile latency. The GC rate model was described in Section 3.3. We further found that the duration of minor GCs tends to increase with the size of the *eden* heap region and the max tenuring threshold parameter. The GC duration model uses this intuition in its parametric part. The 99th percentile latency tends to be affected by two properties of GCs: their average duration and the fraction of time spent in GCs. The parametric part of the latency model includes linear penalties for each of these two quantities. These models are too simplistic to capture the full underlying behavior of the computation, but they do grasp the overall trend. This is sufficient to make the BOAT-based optimizer rapidly converge towards high performance areas.

Results. We ran our bespoke auto-tuner for 10 iterations on YCSB workloads A (50% reads, 50% updates), B (95% reads, 5% updates) and D (95% reads, 5% inserts) (workload C has 100% reads and is not GC-sensitive). After each optimization, we re-evaluated our optimized configuration and compared its 99th percentile latency with the default Cassandra configuration. The results are shown in Figure 5. Error bars are too small to be displayed in the figure as standard deviations were consistently below 1 ms (all results averaged over 3 runs). Our optimized configurations outperforms the Cassandra default configuration by up to 63%. Although we run the optimization for 10 iterations, each optimization converges to within 10% of the best found performance by the second iteration.

We found that the optimized configuration used large eden size, making minor collections longer but less frequent. After inspection we noted that this effectively improved the batching of the collection, reducing the total work. Optimized configurations spent well under 1% of their time in stop the world phases, whereas in Cassandra default configuration's case this was around 4%.

Comparison with other auto-tuners. Our previous results show tuning does yield performance improvements for our workloads. We now consider whether generic auto-tuners would be able to yield similar performance in the same timescale. Figure 6 compares our performance with OpenTuner [4] and Spearmint [26] which we ran for thirty iterations. We run each optimization three times. For each iteration, we report the median, min and max of the best 99th percentile latency achieved so far. We see that within two iterations, our auto-tuner consistently finds a high performance configuration. In contrast, it is only at the 16th iteration that one of the other framework's median value reaches a good performance, after four hours of optimization.

6.2 Neural networks

We now present our neural network case study. Neural networks have seen a surge of interest in recent years, and many frameworks have been proposed to facilitate their training. Here, we built a tuner on top of TensorFlow, a recent framework for distributed machine learning [1, 2]. The API offered by TensorFlow to machine learning applications is low-level. Users must manually set which of their available machines should be used and how much work each should do. TensorFlow offers no automated approach to balance workloads. This task is especially difficult in heterogeneous settings, where the optimal load of a machine depends on its

Neural Network name	Input Type	Network Type	Size (MB)	Ops (Millions)	Batch size range
GoogleNet [27]	Image	Convolutional	26.7	1582	$2^6 - 2^9$
AlexNet [20]	Image	Convolutional	233	714	$2^8 - 2^{11}$
SpeechNet [24]	Audio	Perceptron	173	45.3	$2^{13} - 2^{16}$

Table 2: The three neural networks used in our experiments. Size is the size of the parameters which must be transmitted to and from workers each iteration. Ops is the number of floating point multiplications per input. The name "SpeechNet" is introduced by us for clarity, this network was recently proposed for benchmarking [9].

computational power. Further, the synchronization cost of machines can be high, and hence the slowest workers should not be used at all.

Using BOAT, we built a bespoke auto-tuner that balances a TensorFlow workload. Our tuner takes as input a neural network architecture, a set of available machines and a batch size (an algorithmic parameter described in the next subsection). It then performs ten iterations, each time evaluating a distributed configuration, before returning the one with highest performance. The tuning always finishes within two hours which is small compared to the typical training time of neural networks. The next subsection gives a background of the computation used to train neural networks in a distributed setting.

6.2.1 Distributed training of Neural Networks

Stochastic gradient descent. Neural networks are typically trained with backpropagation using stochastic gradient descent (SGD). Each iteration, a random batch of samples from the training set is drawn. The number of samples is called the *batch size*. Using each sample independently, an estimate of the gradient of the parameters of the network with regard to a *loss function* is computed using backpropagation. Gradient estimates are aggregated and used to update the neural network parameters.

Note that higher batch sizes lead to more parallelism, but lower batch sizes can result in better accuracy of the final neural network [20]. We cannot quantify in advance the impact of the batch size on accuracy, as it depends on the training data, but our experiments expose the trade-off between batch size and computational speed.

Distributed SGD. The parameter server architecture [13] typically used for distributed SGD uses two types of tasks:

- **Parameter Server** tasks synchronize the gradients at every iteration and update the parameters. Each parameter server task is associated with a section of the neural network parameters, e.g. the first layer.
- **Worker** tasks compute the gradient estimates. Each worker is assigned a set of inputs. Each iteration, the worker fetches the updated parameter values from the parameter servers. It then computes the gradient estimate using the inputs it has been assigned. Finally, it sends these gradients back to the relevant parameter server.

Typically, stochastic gradient descent is implemented synchronously. A barrier after each iteration forces workers to work on the same parameter values. Some systems implement asynchronous SGD [23] which removes this barrier and lets workers compute gradients on stale parameter versions. This improves computational performance, especially with large numbers of workers, but can hurt convergence and decrease the final result quality [8]. In this case study, we only consider the synchronous version.

Instance Type	# Hyperthread	GPU	# per setting		
			A	B	C
`g2.2xlarge`	8	1 K520	0	1	2
`c4.2xlarge`	8	/	6	3	2
`c4.4xlarge`	16	/	2	3	2
`c4.8xlarge`	36	/	2	3	4
Total			10	10	10

Table 1: Machine and setting specifications.

6.2.2 Tuning distributed SGD

Configuration Space. We tuned the scheduling of a parameter server architecture, implemented in TensorFlow, to minimize the average iteration time. Given a set of machines, a neural network architecture and a batch size, we set:

- Which subset of machines should be used as workers.
- Which (possibly overlapping) subset of machines should be used as parameter servers.
- Among working machines, how to partition the workload. For machines with GPUs, this includes the partition of workload between their CPU and GPUs. Each device gets assigned a number of inputs to process per iteration. The total number of inputs must sum up to the batch size.

There are effectively two boolean configuration parameters per machine setting whether it should be a worker and/or a parameter server and one to two integer parameters per machine, depending on whether it has a GPU, specifying the load. In our experiments, we tune the scheduling over 10 machines, setting 30-32 parameters. In our largest experiment, there are approximately 10^{53} possible valid configurations, most of which lead to poor performance.

Note that we are only tuning system parameters, the computation performed will be the same independently of the configuration used. In particular, the configuration selected does not affect accuracy.

Objective function. To measure the performance of a setting, we performed twenty iterations of stochastic gradient descents. The first few iterations often showed a high variance in performance and hence we report the average time of the last 10 iterations. We found this was enough to get accurate measurements, repeating configurations showed little underlying noise.

6.2.3 Probabilistic model of distributed SGD

The probabilistic model contains several components.

Individual device computation time. For each device – CPU or GPU – on a worker machine, we modeled the time needed to perform its assigned workload. This time should be near linear with respect to the number of inputs, hence we used a semi-parametric model similar to the one of Section 5.1 with a linear parametric model. We fit one individual device model per type of device available (e.g. c4.4xlarge CPU, or Nvidia GPU K520).

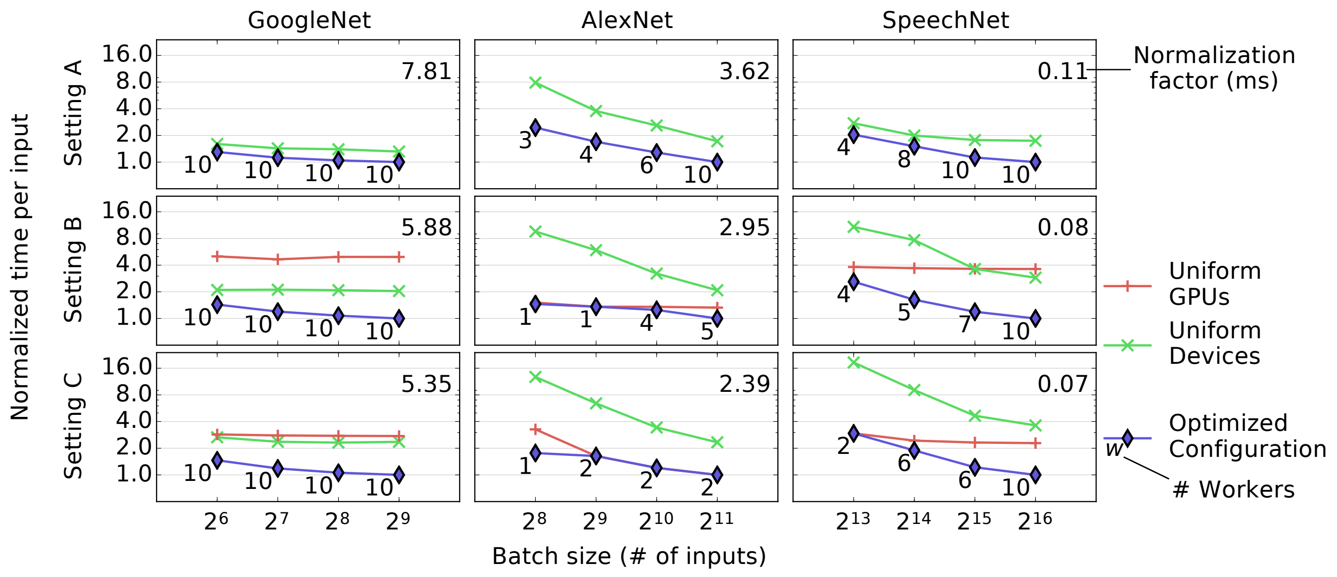


Figure 7: Normalized time per input (lower is better) of simple and optimized configurations on each experiment. Within each sub-graph, results are normalized by the best achieved time per input. This is always the one of the optimized configuration on the largest batch size (the lower right point of each sub-graph). The normalization factor, i.e. the best time per input, is shown at the top right of each sub-graph in milliseconds. For each optimized configuration, we report the number of workers used.

Individual machine computation time. For worker machines with multiple devices, the gradient estimates from each device were summed locally on the CPU before being sent to the parameter servers. We modeled the total computation time per machine as a semi-parametric model. The parametric part returned the maximum computation time of the machine's devices. The non-parametric part was responsible for modeling this aggregation time. We fit one individual machine model per type of machine (e.g. EC2 instance type).

Communication time. We modeled the communication time as another semi-parametric model. Our parametric model learns a *connection_speed* parameter per type of machine. It predicts the total communication time as

$$\max_{m \in machines} \frac{transfer(m)}{connection\_speed_m}$$

where $transfer(m)$ is the amount of data that must be transfered each iteration by machine m. It is a function of whether m is a worker, the number of other workers, and the size of the parameters m holds as a parameter server if any. We fit a single communication time model for the entire cluster. Finally, we predict the total time of an SGD iteration as the sum of the maximum predicted individual machine time and the communication time.

Since our probabilistic model simulates individual device and machine computation times, it benefits from real measurements of these properties. We therefore also measure in each iteration the time needed by all devices and machines to perform their assigned workload.

6.2.4 Experiment results

Experimental Setup. We evaluated our optimizer on Amazon EC2 using TensorFlow version *v0.8*. There are three inputs to our tuning procedure. The machines available, the neural network being trained and the batch size. We constructed three machine settings, described in Table 1, designed to recreate heterogeneous environments. Each contains 10 machines of varying computational power. Settings B and C contain one and two GPU instances respectively. While neural networks perform most efficiently on GPUs, we tried to design realistic settings where a variety of CPUs and GPUs are available. We evaluated each of the three hardware setting with the three neural networks referenced in Table 2 using four batch sizes for a total of thirty-six experiments. The four batch sizes for each network were selected to explore the tradeoff with processing speed. Recall that batch size is an algorithmic parameter equaling inputs per iteration, and that lower batch sizes tend to improve final result accuracy at the cost of less parallelism.

Comparison with simple configurations. To show the importance of tuning, we compared our optimized configurations with two simple configurations 1) *Uniform Devices*: a load balanced equally among all devices, and 2) *Uniform GPUs*: a load balanced equally among GPUs (in Settings B and C). In both cases, we set worker machines to also be parameter servers which tends to yield good results. Figure 7 shows the outcome of each experiment. Our optimized configurations significantly outperform these simple configurations on most experiments.

Inspecting the optimized configurations and their associated models delivers a number of insights.

- **Communication cost.** Large networks, like AlexNet, are often scheduled on a subset of the machines due to their expensive communication cost. This is lessened with larger batch sizes where there is more computation to perform per iteration and hence the cost of using more workers is amortized. On the other hand, the smaller GoogleNet was always scheduled on all available machines.
- **Parameter servers.** Another key setting that must be optimized, which we do not report here due to lack of space, is the set of parameter server machines. Param-

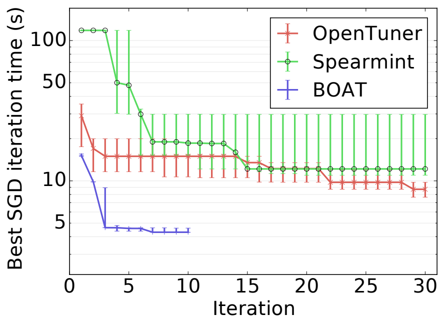


Figure 8: Convergence of the frameworks on Setting C using SpeechNet with a 2^{16} batch size.

eter servers need to perform large amounts of communication and hence benefit from being placed on machines with high network speed. In the optimized configurations, most of the parameters were usually placed on `c4.8xlarge` instances, which indeed have the largest bandwidth. Inspecting the learned communication models shows that they had correctly inferred that the *connection_speed* parameter of `c4.8xlarge` was higher than those of other instances.

- **Load balance.** Comparing "Uniform Devices" configurations with our optimized configurations on GoogleNet shows the importance of load balancing, both used all available devices but the optimized configurations are significantly faster. Interestingly, the correct load balance depends on the neural network architecture. AlexNet's optimized configurations had a higher proportion of work assigned to GPUs than those of SpeechNet.
- **Batch size tradeoff.** Recall that batch size is an algorithmic parameter and lower batch sizes tend to produce better accuracy of the final neural network at the cost of less parallelism. Each sub-graph of Figure 7 shows the tradeoff between processing rate and batch size. With our auto-tuner, users can find optimized configurations for different batch sizes and easily explore this Pareto-front, hiding the details of the configuration used.

These observations confirm the intuition that "one size fits all" approaches are not appropriate, as optimized configurations are influenced by hardware, workload and batch size.

Comparison with traditional Bayesian optimization. We now consider whether the benefits of auto-tuning could have been achieved with an off-the-shelf optimization tool. Figure 8 compares the performance of our bespoke auto-tuner with OpenTuner [4] and Spearmint [26], which were each ran for thirty iterations. Each optimization was run three times, we report for every iteration the median, min and max performance of the best configuration found so far. Our bespoke auto-tuner significantly outperforms generic auto-tuners. The median best configuration achieved by OpenTuner is 8.71s per SGD iteration, more than twice slower than our median time (4.31s), and not much faster than the Uniform GPUs configuration (9.82s). The reason this tuning task is difficult is because the space of efficient configurations is extremely narrow, assigning one of the workers too much work creates a bottleneck, yielding poor performance.

All of our experiments finished the ten iterations within two hours. As neural networks training typically lasts over a week, the performance gains largely outweigh the tuning overhead, making our auto-tuner practical in realistic settings. Our largest experiments involved 32 dimensions, we expect our auto-tuner would scale well to larger settings as there would be a proportional increase in the number of measurements.

7. RELATED WORK

System performance modeling. Predicting the performance of workloads has received significant interest. Most of the time, this is used online as part of a scheduler to best execute an incoming workload. Ernest [29] uses parametric probabilistic models to predict the performance of distributed analytics jobs by first running them on small samples of the data. Quasar[15], Paragon [14] and ProteusTM [16] profile the early stages of workloads to find similar previously-scheduled workloads. They use this to suggest appropriate configurations. These frameworks use generic probabilistic models to predict performance. They work well when few parameters are being tuned but have difficulty scaling to large configuration spaces due to the curse of dimensionality [21]. BOAT tackles this by using bespoke models which are engineered to resemble the system's behavior and can leverage high rates of measurements.

Many fined-grained performance predictors have been built for MapReduce workloads [18]. In comparison, probabilistic programming allows to build simple models, requiring little programmer effort, and leverage empirical data to accurately predict performance. Databases use performance models to predict the behavior of queries [7]. These perform an optimization over a complex configuration space, but rely on the constrained scope of queries, which only use relational operators, to make accurate predictions.

System auto-tuning. Auto-tuning is often used to adapt numerical libraries to their underlying hardware, ATLAS [30] does this in the context of BLAS libraries. OpenTuner [4] is a generic auto-tuner which combines multiple optimization algorithms. In these works, evaluating a configuration can be done quickly, and hence the objective function can be evaluated many times. BOAT makes auto-tuning applicable to new domains where this isn't the case. Spearmint [26] and Yelp's MOE [10] are optimizers implementing traditional Bayesian optimization. BOAT builds upon them by allowing the use of bespoke models.

8. CONCLUSION

In this paper we presented BOAT, a framework to build bespoke auto-tuners in environments where black box optimizers fail. We introduced Structured Bayesian Optimization, which enables developers to inject domain knowledge about the structure of their systems into the optimization procedure. Evaluation results show how optimizers built with BOAT can significantly outperform traditional auto-tuners in complex tuning problems. BOAT is open source and available at `https://github.com/VDalibard/BOAT`.

Acknowledgements

We thank Felix Gessert, Ionel Gog, Tim Harris, Wenjun Hu, Paul Kelly, Rafal Mantiuk, Hugo Paquet, Amitabha Roy, Malte Schwarzkopf and Wolfram Wingerath for offering feedback on early drafts of this paper. This research was partly funded by the EPSRC (EP/P004024, EP/M508007/1 and GCRF) and a Computer Laboratory Premium Scholarship (Sansom scholarship).

9. REFERENCES

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Martín Abadi 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.

[3] Jason Ansel et al. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[4] Jason Ansel et al. Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.

[5] Eric Brochu, Vlad M Cora, and Nando de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Technical Report UBC TR-2009-023, University of British Columbia, 2009.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14:70–93, 2016.

[7] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[8] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016.

[9] Soumith Chintala. Deepmark benchmark. `https://github.com/DeepMark/deepmark`.

[10] Scott Clark, Eric Liu, Peter Frazier, JiaLei Wang, Deniz Oktay, and Norases Vesdapunt. MOE: A global, black box optimization engine for real world metric optimization. `https://github.com/Yelp/MOE`, 2014.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[12] Valentin Dalibard. *A framework to build bespoke auto-tuners with structured Bayesian optimisation.* PhD thesis, University of Cambridge (UCAM-CL-TR-900), January 2017.

[13] Jeffrey Dean et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012*, pages 1232–1240, 2012.

[14] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[15] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014.

[16] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. ProteusTM: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–771. ACM, 2016.

[17] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE, FOSE track)*, 2014.

[18] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[19] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[20] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[21] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* The MIT Press, 2012.

[22] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[23] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[24] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH*, pages 1058–1062, 2014.

[25] 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, 2016.

[26] Jasper Snoek, Hugo Larochelle, and Ryan Prescott Adams. Practical bayesian optimization of machine learning algorithms. In *Neural Information Processing Systems*, 2012.

[27] Christian Szegedy et al. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[28] The Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org.

[29] 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.

[30] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.