TrimTuner: Efficient Optimization of Machine Learning Jobs in the Cloud via Sub-Sampling

Pedro Mendes[1], Maria Casimiro[1,2], Paolo Romano[1], David Garlan[2]
[1]*INESC-ID and Instituto Superior Técnico, Universidade de Lisboa*
[2]*Institute for Software Research, Carnegie Mellon University*

***Abstract*—This work introduces TrimTuner, the first system for optimizing machine learning jobs on the cloud that leverages sub-sampling techniques to reduce the cost of the optimization process while keeping into account user-specified constraints. TrimTuner jointly optimizes the cloud and application-specific parameters and, unlike state of the art works for cloud optimization, eschews the need to train the model with the full training set every time a new configuration is sampled. Indeed, by leveraging sub-sampling techniques and data-sets that are up to 100× smaller than the original one, TrimTuner reduces the cost of the optimization process by up to 50×.**

Further, TrimTuner speeds-up the process of recommending the next configuration to sample by 65×. The reasons for this improvement are twofold: i) a novel domain specific heuristic that reduces the number of configurations for which the acquisition function has to be evaluated; ii) the adoption of an ensemble of decision trees which enables boosting the speed of the recommendation process by one additional order of magnitude.

***Index Terms*—Machine Learning, Cloud optimization, Sub-sampling, Bayesian Optimization**

I. Introduction

Training machine learning (ML) jobs on the cloud represents the *de facto* standard approach today for training large ML models. Existing ML models are in fact increasingly complex [1], and their training procedure sometimes involves an enormous amount of computational resources, which the cloud can provision in an elastic and on-demand fashion, sparing users and enterprises from colossal capital investments. Further, by taking advantage of economies of scale, cloud providers can drastically reduce their internal operational costs, which ultimately leads to cost savings for the end users.

Nonetheless, a key problem that users face is that modern cloud providers offer a large spectrum of heterogeneous virtual machine (VM) types, optimized for different types of resources and with different costs. For instance, at the time of writing, Amazon Web Services (AWS) EC2 offers approximately 300 different VM types [2]. The problem is further exacerbated by the fact that the (distributed) training process of ML jobs exposes several hyper-parameters — such as the batch size considered in each training iteration or the frequency of synchronization among workers. The optimal configuration for these parameters can be substantially affected by the choice of the type and number of provisioned cloud resources [3].

As such, end users who wish to train their ML models in cloud infrastructures are faced with a complex constrained optimization problem: determining which type/amount of cloud resources and model hyper-parameters to use in order to maximize the model's accuracy while enforcing constraints on the maximum cost and/or time of the training process.

Given the complexity of modelling the dynamics of modern ML and cloud platforms via white-box methods, a common approach in the literature is to rely on black-box modelling and Bayesian-optimization techniques [3]–[5]. These techniques have the key advantage of requiring no prior knowledge of the target ML model to be optimized, and as such require the target ML model to be deployed and trained in several (cloud/hyper-parameter) configurations. The corresponding measurements of accuracy and execution cost/time are then used to build black-box models, e.g., Gaussian Processes (GPs), that guide the optimization process by recommending which configurations to test next by balancing — via different model-driven heuristics, a.k.a., acquisition functions [6] — exploration (of unknown regions of the configuration space) and exploitation (of the models' current knowledge).

Unfortunately, these systems suffer from a severe limitation: each time a configuration is tested, the target ML model needs to be trained on its entire data-set. As such, with these methods, the optimization process can become prohibitively expensive (and slow) if the target model is meant to be trained on massive data-sets, as is increasingly the case in practice [1].

In this work, we tackle this problem by presenting TrimTuner, the first system for optimizing the training of ML jobs on the cloud that exploits data sub-sampling techniques to enhance the efficiency of the optimization process.

TrimTuner considers the problem of identifying the cloud and hyper-parameter configuration that maximizes the accuracy of a ML model, while ensuring that user-defined constraints on the efficiency (e.g., cost or execution time) of the training process are complied with.

TrimTuner deploys the target job using data-sets that are up to 100× smaller than the original one and constructs predictive models that keep into account how shifts of the data-set size affect both the quality (i.e., accuracy) and training efficiency (i.e., cost or execution time) of the target model. These models are then used within a novel acquisition function that aims at estimating the advantage of testing a new configuration $\mathbf{x}$ using a data-set that is $s\times$ smaller than the original one ($s \in [0,1]$) by weighing in two main factors: (i) the expected information gain [7] that testing $\langle \mathbf{x}, s \rangle$ will yield on the configuration that maximizes accuracy when using the full data-set (s=1) and (ii) the likelihood that the latter configuration $\langle \mathbf{x}, s = 1 \rangle$ meets the user-specified constraints.

Overall, this paper makes contributions of both a methodological and practical nature. From a methodological perspective, TrimTuner builds on recent systems for hyper-parameter tuning of ML models [8]–[10], which had first investigated the use of sub-sampling techniques, and extends them in a number of ways: (i) by supporting the enforcement of additional user-specified constraints; (ii) by jointly optimizing the model's hyper-parameters and the selection of the cloud configuration (number and type of virtual machines); (iii) by introducing a novel domain-specific heuristic, named "Constrained Expected Accuracy", that accelerates the recommendation process by a factor of up to 2× when compared to state of the art approaches [11], [12]; (iv) by proposing the adoption of an ensemble [13] of decision trees [14] as a light-weight alternative to Gaussian Processes (the *de facto* standard modelling approach for BO), which enables boosting the speed of the recommendation process by one additional order of magnitude.

From a practical perspective, when compared to state of the art BO-based systems for the (constrained) optimization of ML jobs, such as Lynceus [3] or Cherrypick [4], TrimTuner reduces the cost of the exploration process by up to 50× thanks to the use of sub-sampling techniques. Another practical contribution of this work is making available to the community the data-sets obtained for the evaluation of TrimTuner, which consider the training of three neural networks via TensorFlow on AWS EC2 over a search space (encompassing both model hyper-parameters and cloud-related parameters) composed of approximately 1400 configurations, whose collection incurred a cost of approximately $1200 and took about two months.

The remainder of this paper is structured as follows: in § II we describe related work; § III presents TrimTuner; § IV evaluates our contributions and finally § V concludes the paper.

II. Related Work

This section discusses related works in the areas of cloud optimization and Bayesian optimization.

Cloud Optimization Approaches. Recent works to find the optimal configuration to deploy jobs in the cloud, such as CherryPick [4], Ernest [15], Paris [16] and Arrow [5] focus solely on optimizing cloud related parameters and disregard the possibility of learning from related tasks. Lynceus [3] was the first work to highlight the relevance of jointly optimizing the cloud configuration and the hyper-parameters affecting the distributed training process of ML models.

The key difference between TrimTuner and these systems is its reliance on sub-sampling techniques to reduce the cost of testing configurations during the optimization process. As we discuss in § IV, this allows TrimTuner to reduce the cost and duration of the optimization process by up to a factor of 50 and 65, respectively.

Another key aspect of TrimTuner, which differentiates it from systems like Paris [16], Quasar [17], or Arrow [5] is that it does not rely on *a priori* knowledge of similar types of jobs, whose representativeness constitutes a key assumption on which the accuracy of the optimization process hinges. Conversely, TrimTuner (analogously, e.g., to CherryPick and Lynceus) operates in a fully on-line fashion.

Bayesian Optimization. TrimTuner, similarly to other recent systems for the optimization of cloud jobs [3], [4], relies on a generic optimization methodology, known as (model based) Bayesian Optimization (BO), which has been adopted in a wide range of application domains including self-tuning of transactional memory systems [18], [19], databases [20] and of hyper-parameters of ML models [8], [10], [21].

BO aims to identify the optimum $\mathbf{x}^*$ of an unknown function $f : \mathbb{X} \rightarrow \mathbb{R}$ and operates as follows [22]: (i) f is evaluated (i.e., tested or sampled) over N initial configurations, $\mathbf{x}_i$, selected at random so as to build an initial training set $\mathcal{S}$ composed of pairs $\langle \mathbf{x}_i, f(\mathbf{x}_i) \rangle$; (ii) $\mathcal{S}$ is used to train a black-box model (typically a Gaussian Process [23]) that serves as a predictor/estimator of the unknown function f; (iii) an *acquisition function*, noted α, is used to exploit the model's knowledge (and related uncertainty) to determine which configuration to evaluate next by balancing exploitation of model's knowledge — recommending configurations that the model deems to be optimal — and exploratory behaviours — recommending configurations whose knowledge can reduce the model's uncertainty and enhance its accuracy; (iv) the process is iteratively repeated until a stopping condition is met, e.g., after a fixed budget is consumed or if the gains from further sampling are predicted to be marginal by the model or below a fixed threshold.

The definition of the acquisition function is arguably one of the most crucial components of BO methods and in the literature there exist a number of proposals. Expected Improvement (EI) is probably the most well-known acquisition function. As the name suggests, EI (Eq. (1)) uses the probability distribution of observing a value $f(\mathbf{x})$ at $\mathbf{x}$, predicted by the model trained on data-set $\mathcal{S}$, in order to measure the expected amount by which evaluating f at $\mathbf{x}$ can improve over the current best value or *incumbent* η:

$$\alpha_{EI}(\mathbf{x}) = \int max(0, f(\mathbf{x}) - \eta) p(f(\mathbf{x})|\mathcal{S}) df(\mathbf{x}) \qquad (1)$$

Entropy Search (ES) [24] is an alternative acquisition function that chooses which configurations to evaluate by predicting the corresponding information gain on the optimum, rather than aiming to evaluate near the optimum (as in EI). ES (Eq. (2)) is based on the probability distribution $p_{opt}(\mathbf{x}|\mathcal{S})$, namely the likelihood that a configuration $\mathbf{x}$ belongs to the set of optimal configurations for f, given the current observations in $\mathcal{S}$. The information gain deriving from testing $\mathbf{x}$ is computed using the expected Kullback-Leibler divergence (relative entropy) between $p_{opt}(\cdot|\mathcal{S} \cup \{\mathbf{x}, y\})$ and the uniform distribution $u(\mathbf{x})$, with expectations taken over the model-predicted probability of obtaining measurement y at $\mathbf{x}$:

$$\alpha_{ES}(\mathbf{x}) = \mathbb{E}_{p(y|\mathbf{x},\mathcal{S})} \left[\int p_{opt}(\mathbf{x'}|\mathcal{S} \cup \{\mathbf{x}, y\}) \cdot \log \frac{p_{opt}(\mathbf{x'}|\mathcal{S} \cup \{\mathbf{x}, y\})}{u(\mathbf{x'})} d\mathbf{x'} \right] \qquad (2)$$

Despite being numerically much more complex to compute than EI, ES allows for quantifying to what extent testing a configuration $\mathbf{x}$ will give the model knowledge about the optimum $\mathbf{x}^*$, where generally $\mathbf{x} \neq \mathbf{x}^*$. In the context of hyper-parameter tuning of ML models, this property of ES has been exploited by MTBO [21] and FABOLAS [8] to trade off the information gain and the cost (i.e., execution time) of training a ML model in a configuration $\langle \mathbf{x}, s \rangle$, where $\mathbf{x}$ denotes a hyper-parameter's configuration and $s \in [0, 1]$ the sub-sampling rate applied to the original/full model's training data-set.

More precisely, in FABOLAS, the acquisition function (Eq. (3)) for $\langle \mathbf{x}, s \rangle$ is defined as the ratio between the information gain on the configuration that maximizes accuracy for the full data-set (s=1) and the (predicted) cost of training the model with a sub-sampling rate s and hyper-parameters $\mathbf{x}$:

$$\alpha_F(\mathbf{x}, s) = \mathbb{E}_{p(y|\mathbf{x},s,\mathcal{S})} \left[\int p_{opt}^{s=1}(\mathbf{x'}|\mathcal{S} \cup \{\mathbf{x}, s, y\}) \log \frac{p_{opt}^{s=1}(\mathbf{x'}|\mathcal{S} \cup \{\mathbf{x}, s, y\})}{u(\mathbf{x'})} d\mathbf{x'} \right] \cdot \frac{1}{C(\mathbf{x}, s)} \quad (3)$$

TrimTuner builds on these approaches and extends them in several ways. First, TrimTuner supports the definition of additional (independent) constraints, e.g., on cloud cost and/or execution times of training/querying the model. This is achieved by extending the acquisition function to factor in the probability that the new incumbent will comply with the constraints, which will be discovered after updating the model with the observation of a (possibly sub-sampled) configuration $\langle \mathbf{x}, s \rangle$. Unlike existing constrained versions of ES-based acquisition functions, such as Predictive Entropy Search with Constraints (PESC) [25] and constrained Max-value Entropy Search (cMES) [26], the proposed acquisition function does not make use of Bochner's theorem for a spectral approximation, which allows TrimTuner to use GPs with non-stationary kernels (as in FABOLAS) or lightweight (ensembles of) decision trees. Further, TrimTuner jointly optimizes the hyper-parameters of the training process and the cloud configuration — which, as already mentioned, is crucial to maximize the cost efficiency of the recommended configuration [3].

Finally, the numerical computation of the ES (and of any acquisition function based on ES, like FABOLAS' and TrimTuner's) is onerous. TrimTuner introduces two mechanisms to accelerate the recommendation process: (i) a novel domain-specific heuristic, that is used to estimate which configurations are most likely to yield the highest values of the acquisition function — thus restricting the number of configurations for which the acquisition function is evaluated; (ii) differently from FABOLAS and MBTO, which rely on GPs to estimate the probability distribution of the accuracy and cost of unknown configurations, TrimTuner relies on an ensemble of decision trees that, as we will show in § IV, achieves comparable accuracy while enabling speed-ups of up to 14×.

III. TrimTuner

TrimTuner is a Bayesian Optimization (BO) based approach that jointly optimizes the configuration of the cloud platform over which a ML model is trained as well as the hyper-parameters of the training process. More formally, TrimTuner considers the following constrained optimization problem:

$$\begin{aligned} \underset{\mathbf{x} \in \mathbb{X}}{\text{maximize}} \quad & A(\mathbf{x}, s = 1) \\ \text{subject to} \quad & q_1(\mathbf{x}, s = 1) \geq 0, \ldots, q_m(\mathbf{x}, s = 1) \geq 0 \end{aligned} \quad (4)$$

where $\mathbf{x} \in \mathbb{X}$ denotes the vector encoding both the cloud's and hyper-parameters' configuration, $s \in [0, 1]$ is the sub-sampling rate (relative to the full data-set), A is the accuracy and $q_1 \ldots q_m$ is a set of Quality of Service (QoS) constraints on the target model (e.g., on the maximum duration/cost of its training process or on the latency for querying the resulting ML model). We assume that the accuracy and constraint functions are unknown, independent, and can only be observed via point-wise and noisy evaluations by training the ML model in the corresponding configuration.

Note that both the objective function (i.e., model's accuracy) and the constraints are expressed for configurations using the full data-set (s=1). However, in order to enhance the efficiency of the optimization process, TrimTuner tests configurations using sub-sampled data-sets ($s < 1$) and leverages the knowledge gained via these cheaper evaluations to recommend configurations that use the full data-set.

Algorithm 1 provides the pseudo-code of TrimTuner. TrimTuner receives as input: (i) the ML model (M) whose training process has to be optimized along with its (full) training set; (ii) the set $\mathbb{X}$ of possible cloud and hyper-parameter configurations; (iii) a set of QoS constraints ($\mathbf{Q}$).

Initialization phase. As in typical BO approaches, since no *a priori* knowledge is assumed on the target job, TrimTuner

Algorithm 1 TrimTuner's pseudo-code

1: **function** TrimTuner(M, $\mathbb{X}$, $\mathbf{Q}$)
▷ *M: Model to be trained and full data-set*
▷ *$\mathbb{X}$: Set of possible cloud and hyper-parameters configurations*
▷ *$\mathbf{Q}$: Set of QoS constraints*
▷ *Initialization phase*
2: $\mathcal{T} = \{\langle \mathbf{x}, s_i \rangle : \mathbf{x} \in \mathbb{X}, i \in [1, k]\}$ ▷ *Set of untested configs.*
3: Select at random a configuration $\mathbf{x} \in \mathbb{X}$
4: **for** $i = 1, \ldots, k$ **do** ▷ *Test $\boldsymbol{x}$ with k sub-sampling rates*
5: $\langle a, c, \mathbf{q} \rangle \leftarrow$ Train M in configuration $\langle \mathbf{x}, s_i \rangle$
6: $\mathcal{S}^A \leftarrow \mathcal{S}^A \cup \{\mathbf{x}, s_i, a\}$ ▷ *Add accuracy of $\langle \boldsymbol{x}, s_i \rangle$ to $\mathcal{S}^A$*
7: $\mathcal{S}^C \leftarrow \mathcal{S}^C \cup \{\mathbf{x}, s_i, c\}$ ▷ *Add cost of $\langle \boldsymbol{x}, s_i \rangle$ to $\mathcal{S}^C$*
8: $\mathcal{S}^{\mathbf{Q}} \leftarrow \mathcal{S}^{\mathbf{Q}} \cup \{\mathbf{x}, s_i, \mathbf{q}\}$ ▷ *Add QoS constr. of $\langle \boldsymbol{x}, s_i \rangle$ to $\mathcal{S}^{\boldsymbol{Q}}$*
9: $\mathcal{T} \leftarrow \mathcal{T} \setminus \langle \mathbf{x}, s_i \rangle$ ▷ *Remove $\langle \boldsymbol{x}, s_i \rangle$ from untested configs*
10: Fit models $A(\mathbf{x}, s)$,$C(\mathbf{x}, s)$,$\mathbf{Q}(\mathbf{x}, s)$ using $\mathcal{S}^A$,$\mathcal{S}^C$,$\mathcal{S}^{\mathbf{Q}}$, resp.
▷ *Main optimization loop*
11: **for** $(i = 1, \ldots, MaxIterations)$ **do**
▷ *Select the most promising configurations using CEA*
12: $\mathcal{D} \leftarrow$ { β configs $\langle x, s \rangle \in \mathcal{T}$ with max. CEA }
▷ *Acq. fun. evaluated only on the configs selected by CEA*
13: $\langle \mathbf{x}', s' \rangle \leftarrow argmax_{\langle \mathbf{x}, s \rangle \in \mathcal{D}} \alpha_T(\mathbf{x}, s)$ ▷ *Eq. 5*
14: $\langle a, c, \mathbf{q} \rangle \leftarrow$ Train M in configuration $\langle \mathbf{x}', s' \rangle$
15: $\mathcal{S}^A \leftarrow \mathcal{S}^A \cup \{\mathbf{x}', s', a\}$ ▷ *Add accuracy of $\langle \boldsymbol{x}', s' \rangle$ to $\mathcal{S}^A$*
16: $\mathcal{S}^C \leftarrow \mathcal{S}^C \cup \{\mathbf{x}', s', c\}$ ▷ *Add cost of $\langle \boldsymbol{x}', s' \rangle$ to $\mathcal{S}^C$*
17: $\mathcal{S}^{\mathbf{Q}} \leftarrow \mathcal{S}^{\mathbf{Q}} \cup \{\mathbf{x}', s', \mathbf{q}\}$ ▷ *Add QoS of $\langle \boldsymbol{x}', s' \rangle$ to $\mathcal{S}^{\boldsymbol{Q}}$*
18: $\mathcal{T} \leftarrow \mathcal{T} \setminus \langle \mathbf{x}', s' \rangle$ ▷ *Remove $\langle \boldsymbol{x}', s' \rangle$ from untested configs*
19: Fit $A(\mathbf{x}, s)$,$C(\mathbf{x}, s)$,$\mathbf{Q}(\mathbf{x}, s)$ using $\mathcal{S}^A$,$\mathcal{S}^C$,$\mathcal{S}^{\mathbf{Q}}$, resp.
20: New incumbent $\mathbf{x}^*$: feasible config. with max accuracy for $s = 1$, as predicted by the models $A(\cdot)$ and $\mathbf{Q}(\cdot)$.

bootstraps its knowledge base via a random sampling strategy. More precisely, TrimTuner randomly selects a configuration[1] $\mathbf{x} \in \mathbb{X}$ and tests it (i.e., trains the model) using different sub-sampling levels $s_1, \ldots, s_k$.

We choose these sub-sampling levels so that they are biased towards configurations with small data-sets (thus reducing the cost/duration of the bootstrapping phase), while still gathering representative information on how variations of the sub-sampling rate s affect the objective and the constraint functions. Specifically, in the MNIST data-set (which will be evaluated in § IV), we consider $s \in \{1/60, 1/10, 1/4, 1/2\}$. Note that since we test the same cloud/hyper-parameter configuration and only vary the sampling rate, we can test all the configurations $\langle \mathbf{x}, s_i \rangle$ ($i \in [1, k]$) via a single training instance by taking a snapshot of the accuracy and QoS constraints whenever the sub-sampling rate s_i is achieved. This ensures that the model bootstrapping has a cost equivalent to testing a single configuration using only 50% of the model's data-set.

Whenever a configuration $\langle \mathbf{x}, s \rangle$ is tested, the model is retrained and the corresponding accuracy, training cost (due to the use of cloud resources) and QoS constraint values are stored in the $\mathcal{S}^A$, $\mathcal{S}^C$ and $\mathcal{S}^{\mathbf{Q}}$ data-sets, respectively.

The initialization phase ends by building a distinct black-box model (see § III-A) that predicts, for an (untested) configuration $\langle \mathbf{x}, s_i \rangle$: (i) the accuracy of the target ML model, noted $A(\mathbf{x}, s_i)$, using S^A (ii) the (cloud) cost of its training process, noted $C(\mathbf{x}, s_i)$, using S^C and (iii) the value of each QoS constraint $q_1 \ldots q_m \in \mathbf{Q}$, noted $\mathbf{Q}(\mathbf{x}, s_i)$, using $S^{\mathbf{Q}}$.

Main optimization loop. The optimization loop of TrimTuner relies on a novel acquisition function (Eq. (5)) to determine which configuration to test next:

$$\alpha_T(\mathbf{x}, s) = \mathbb{E}_{p(\mathbf{q},a|\mathbf{x},s,\mathcal{S})}\left[\prod^{q_i \in \mathbf{Q}} p(q_i(\mathbf{x}^*, s{=}1) \geq 0 | \mathcal{S} \cup \{\mathbf{x}, s, \mathbf{q}, a\}\right] \frac{1}{C(\mathbf{x}, s)} \cdot \mathbb{E}_{p(a|\mathbf{x},s,\mathcal{S})}\left[\int p^{s=1}_{opt}(\mathbf{x'}|\mathcal{S}^A \cup \{\mathbf{x}, s, a\}) \log \frac{p^{s=1}_{opt}(\mathbf{x'}|\mathcal{S}^A \cup \{\mathbf{x}, s, a\})}{u(\mathbf{x'})} d\mathbf{x'}\right] \tag{5}$$

Conceptually, this acquisition function extends the one of FABOLAS (Eq. (3)) by additionally accounting for the probability that the new incumbent, $\mathbf{x}^*$, predicted by the models after having acquired knowledge on configuration $\langle \mathbf{x}, s \rangle$ will meet the QoS constraints. The key challenge here is that the new incumbent that the models will predict after testing $\langle \mathbf{x}, s \rangle$ is unknown at this point, since we are still reasoning on whether to test $\langle \mathbf{x}, s \rangle$ or not.

We tackle this problem via a simulation approach that exploits the current model's knowledge. Intuitively, when computing the acquisition function for $\langle \mathbf{x}, s \rangle$, the following steps are executed for *all* possible values of accuracy and QoS constraints, $\langle a, \mathbf{q} \rangle$:

1) Extend the accuracy and constraints data-sets ($\mathcal{S}^A$ and $\mathcal{S}^{\mathbf{Q}}$) with $\{\mathbf{x}, s, a\}$ and $\{\mathbf{x}, s, \mathbf{q}\}$, respectively.
2) Train the models with the extended data-sets.
3) Identify the new incumbent $\mathbf{x}^*$ predicted using the updated models. This is the configuration $\langle \mathbf{x}, s = 1 \rangle$ that, based on the predictions of the updated accuracy and constraint models, achieves the largest accuracy among the ones that comply with all constraints.
4) Compute the probability that all constraints are met by the incumbent determined in the previous step, as the product of the probabilities (predicted by the updated models) that each constraint $q_i \in \mathbf{Q}$ is respected (recall that constraints are assumed independent).

Finally, the expectation over all possible values of accuracy and QoS constraints $\langle a, \mathbf{q} \rangle$ has to be computed. To this end, the models (prior to being updated) can be used to predict the probability of configuration $\langle \mathbf{x}, s \rangle$ yielding $\langle a, \mathbf{q} \rangle$ for its accuracy and constraints, respectively. The above expectation can be numerically approximated, e.g., using the Gauss-Hermite quadrature [28], which, roughly speaking, approximates the unbounded integral associated with the expectation (that should be computed over all possible values $\langle a, \mathbf{q} \rangle$) by discretizing the $\langle \mathbf{x}, s \rangle$ space over a small number of predetermined "root" points. To limit the computational complexity of computing α_T, in TrimTuner we use a single coarser, but cheaper, approximation, which conceptually coincides with using a singe root in the GH quadrature: we simulate the testing of $\langle \mathbf{x}, s \rangle$ by computing the above steps considering only for $\langle a, \mathbf{q} \rangle$ the accuracy and QoS constraints values predicted by the corresponding models in $\langle \mathbf{x}, s \rangle$.

Note that α_T (see Eq. (5)) extends the acquisition function of FABOLAS (see Eq. (3)) in a modular way, in the sense that the second and third lines of Eq. (5), which correspond to FABOLAS' acquistion function, can be numerically computed as discussed in FABOLAS' paper.

Note also that, for efficiency reasons, TrimTuner (see Alg. 1 line 12) does not evaluate the acquisition function on every possible untested configuration $\langle \mathbf{x}, s \rangle$, but only on a smaller sub-set (denoted $\mathcal{D}$ in the pseudo-code) that is determined via a novel filtering heuristic, which we called CEA (Constrained Expected Accuracy) and that we describe in § III-B.

Once having determined which configuration $\langle \mathbf{x}', s' \rangle \in \mathcal{D}$ maximizes the acquisition function α_T, that configuration is tested (by training M) and the observed accuracy, cost and constraints values are stored in the corresponding data-sets. Next, the models are updated to incorporate the knowledge obtained by testing $\langle \mathbf{x}', s' \rangle$ and the new incumbent is recommended. As already mentioned, the incumbent is selected using the accuracy and constraint models as the configuration with full data-set (s=1) that is (predicted to be) feasible and that achieves maximum accuracy.

TrimTuner adopts a simple stop condition that terminates the optimization after a fixed number of cycles. However, it would be relatively straightforward to incorporate more sophisticated, adaptive stop-conditions [18], [22] that, e.g.,

[1]Testing more than a configuration provided no benefit in all the tests we performed, but TrimTuner supports testing a larger number of initial configurations, selecting them using Latin Hyper-Cube Sampling [27].

interrupt the optimization if the new predicted incumbent does not improve significantly over the best known optimum.

A. Models

In order to estimate the probability distribution of accuracy (A), cost (C) and constraints ($\mathbf{Q}$) for unexplored configurations $\langle \mathbf{x}, s \rangle$, TrimTuner relies on two black-box modelling techniques, namely Gaussian Processes (GP) and ensembles of Decision Trees (DT).

GPs. GPs [23] represent the *de facto* standard modelling approach in BO, due to their analytical tractability and flexibility [8], [22]. Key to the tractability of GPs is that the prediction that they generate follows, by construction, a Gaussian Distribution with known parameters. It is the possibility to define specialized kernels that provides flexibility to GPs, by allowing to imbue the model with domain-specific knowledge.

TrimTuner, analogously to FABOLAS, relies on kernels designed to capture the expected impact on cost and accuracy deriving from the use of sub-sampling. Specifically, we use a kernel obtained by the inner product of a "general purpose" *Matérn* 5/2 kernel and two custom kernels that encode, respectively, the expectation that accuracy and cost of a ML model grow normally with larger data-set sizes (see [8] for details).

Training GPs is notoriously an expensive process, since estimating their hyper-parameters requires complex/time-consuming marginalizations using Markov Chain Monte Carlo Methods (MCMC). Since evaluating the acquisition function of Eq. (5) (as well as the one of Eq. (3)), requires re-training the models several times, as we shall see in § IV, the recommendation process can be extremely slow if a computationally expensive modelling technique, such as GPs, is used. This led us to explore an alternative approach based on DTs (more precisely Extra Trees [29]).

Ensemble of DTs. DTs are known for their high efficiency, but they can not be directly used to replace GPs since, unlike GPs, DTs do not provide a measure of uncertainty of their prediction. We circumvent this problem by using an ensemble of DTs and injecting diversity among the various learners by generating their data-sets drawing with replacement from the same data-set. We then estimate the probability distribution for a prediction as a Gaussian with mean and standard deviation derived from the predictions of the ensemble.

B. CEA

As mentioned, the numerical computation of TrimTuner's acquisition function (and in general of ES-based acquisition functions) is very expensive, especially if GP models are used. The problem is further exacerbated since TrimTuner, differently from BO-based optimizers like FABOLAS, considers a configuration space that encompasses not only the model's hyper-parameters, but also the cloud configuration. This results in an exponential growth of the search space and, as such, of the set of untested configurations for which the acquisition function should be evaluated.

The common approach, in the BO literature, to cope with this computational challenge, is to rely on generic search heuristics. These heuristics range from simple random sampling to sophisticated black-box optimizers [11], [12] and aim at reducing the number of configurations for which the acquisition function is evaluated. TrimTuner instead introduces a novel, domain-specific heuristic called Constrained Expected Accuracy (CEA) (Eq. (6)):

$$CEA(\mathbf{x}, s) = A(\mathbf{x}, s) \cdot \prod^{q_i \in \mathbf{Q}} p(q_i(\mathbf{x}, s) \geq 0 | \mathcal{S}) \qquad (6)$$

Defined as the product of the predicted accuracy for a (possibly sub-sampled) configuration $\langle \mathbf{x}, s \rangle$ by the probability that $\langle \mathbf{x}, s \rangle$ satisfies the constraints, CEA can be seen as a rough, but cheap, approximation of α_T. In fact, while CEA directly estimates the quality of a (possibly sub-sampled) configuration $\langle \mathbf{x}, s \rangle$, α_T predicts how much information the test of $\langle \mathbf{x}, s \rangle$ will disclose about $\langle \mathbf{x}^*, s = 1 \rangle$, namely the optimal feasible configuration using the full data-set.

Due to its simplicity, CEA can be efficiently evaluated for every untested configuration, and TrimTuner uses it to rank and filter configurations: only the $\beta \in [0, 1]$ configurations with largest CEA are evaluated using α_T. We evaluate the sensitivity of TrimTuner to the β parameter in § IV-B.

TABLE I: TensorFlow (top) and cloud (bottom) parameters.

Parameter	Values
Learning rate	$\{10^{-3}, 10^{-4}, 10^{-5}\}$
Batch size	$\{16, 256\}$
Training mode	{sync, async}
Data-set size [%]	$\{1.67, 10, 25, 50, 100\}$

VM type	VM characteristics	#VMs
t2.small	{1 VCPU, 2 GB RAM}	{8, 16, 32, 48, 64, 80}
t2.medium	{2 VCPU, 4 GB RAM}	{4, 8, 16, 24, 32, 40}
t2.xlarge	{4 VCPU, 16 GB RAM}	{2, 4, 8, 12, 16, 20}
t2.2xlarge	{8 VCPU, 32 GB RAM}	{1, 2, 4, 6, 8, 10}

TABLE II: Number of feasible configurations with an accuracy within 5% of the accuracy of the feasible configuration with highest accuracy for the full data-set.

Neural Network	Feasible Configurations	Feasible Configurations with high accuracy
RNN	178 (61.8%)	28 (9.72%)
MLP	161 (55.8%)	29 (10.07%)
CNN	111 (38.5%)	39 (13.54%)

IV. Evaluation

This section aims at addressing two main questions:

- Which cost and time savings does TrimTuner achieve vs existing BO-based approaches (§ IV-A)?
- How expensive is the recommendation process in TrimTuner and how effective is the CEA heuristic in accelerating it (§ IV-B)?

Data-sets. The data-sets used to evaluate TrimTuner were obtained by training three different neural networks (NN)

(Convolutional Neural Network (CNN), Multilayer Perceptron (MLP), and Recurrent Neural Network (RNN)) in the AWS cloud. The networks were implemented using the Tensorflow framework [30] and trained on the MNIST database [31].

Each configuration (see Table I) is composed of cloud resources (number, type, and size of the virtual machines), application-specific parameters (batch size, learning rate, and training mode), and the size of the data-set used to train the NN. This results in a search space of 1440 configurations (288 configurations for each data-set size). To reduce noise in the measurements, we executed the training of the NN in each configuration three times. Whenever we train a model in a configuration $\langle \mathbf{x}, s \rangle$ we measure the achieved accuracy, training time and cloud cost. Collecting these data-sets took more than 2 months and costed a total of approx. $1200. We have made these data-sets publicly available[2], along with the implementations of TrimTuner and of the considered baselines.

Based on these data-sets, we define a QoS constraint that limits the maximum training cost to be $0.02, $0.06, and $0.1 for RNN, MLP, and CNN, respectively. Table II reports the number (percentage) of configurations that use the full data-set (s=1) and comply with the cost constraints, and the number of those whose accuracy is no more than 5% lower than the accuracy of the feasible configuration with highest accuracy. Only around 10% of the full data-set configurations are close to the optimum, which illustrates the non-triviality of the considered optimization problem.

Baselines. We compare TrimTuner using GPs and DTs against EIc and EIc/USD, two popular acquisition functions for constrained optimization problems that were used by two recent cloud optimizers (CherryPick [4] and Lynceus [3], resp.).

EIc extends EI (Eq. (1)) by factoring in the probability that the configuration being evaluated will meet the constraints. EIc/USD, in its turn, extends EIc by considering the trade off between the benefits stemming from an exploration (computed using EIc) and the exploration cost (estimated via a dedicated model). None of these techniques use sub-sampling. We use GPs as base models for both EIc and EIc/USD that were implemented resorting to George library [32] in Python. We include in the comparison also Fabolas [8], which uses sub-sampling but does not consider constraints, and a simple random approach. We used the publicly available standard implementation of FABOLAS.

Experimental setup and evaluation. The reported results are the average of 10 independent runs. For all the compared systems we bootstrap the models using 4 initial samples. For TrimTuner and FABOLAS, that use sub-sampling strategies, we select a cloud and hyper-parameter configuration uniformly at random, and test it over the considered 4 data-set sizes (Table I). For EIc and EIc/USD, which do not use sub-sampling, we sample 4 full data-set configurations using Latin Hypercube Sampling (LHS).

We set the maximum number of iterations to 44 for all optimizers. Unless otherwise stated, for both TrimTuner variants we use the CEA heuristic, setting the filtering rate β to 10%.

All the systems were implemented in Python3.6 and the simulations were deployed in a VM running Ubuntu 18.04 LTS with 32 cores and 8GB of memory, hosted in machines with a Intel Xeon Gold 6138 CPU and 64GB of memory.

Evaluation Metrics. To evaluate the systems, we use a metric which we named *Constrained Accuracy* ($Accuracy_C$), that penalizes recommended configurations that do not respect the cost constraint.

$$Accuracy_C = \begin{cases} A(x,s), & \text{if } (x,s) \text{ is feasible} \\ A(x,s) \cdot \frac{C_{max}}{C(x,s)}, & \text{otherwise} \end{cases} \quad (7)$$

It is easy to see that this metric imposes larger penalizations to configurations that violate the constraint by a larger extent.

A. Comparison with state of the art optimizers

Figure 1 evaluates the cost efficiency of the compared solutions by reporting the $Accuracy_C$ of the recommended incumbent as a function of the cost of the optimization process for the various networks. The plots clearly show that both TrimTuner variants achieve higher $Accuracy_C$ levels at a fraction of the cost of the other solutions. The reason underlying TrimTuner's gains with respect to EIc and EIc/USD is the use of sub-sampling, whose benefits are clear both in the initialization stage (shown using dashed lines) and once the models are in use. The average cost of each exploration step with TrimTuner is approx. 10× and 2.4× smaller than with EIc and EIc/USD, respectively, which is explicable considering that TrimTuner uses an average sub-sampling rate of approx. 1/4 whereas EIc and EIc/USD test using full data-sets.

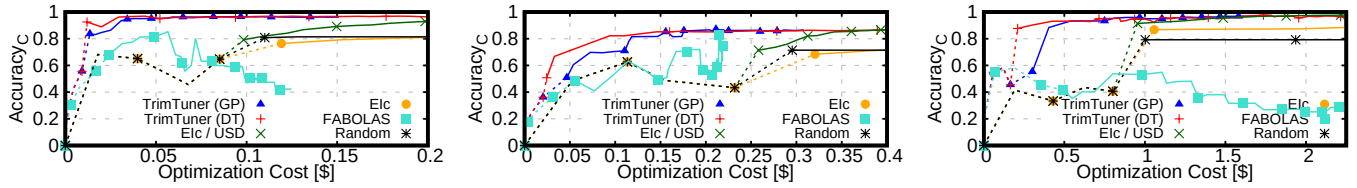
As expected, FABOLAS is the worst performing solution in this constrained optimization problem (for which it is not designed): since FABOLAS does not keep into account constraints, although the configurations it recommended achieve high accuracy, they frequently violate the cost constraint.

As for the cost efficiency of the optimization process for the two variants of TrimTuner, the plots do not highlight significant differences. DTs appear to be slightly more accurate than GPs in the considered data-sets, confirming that they can represent a solid (and as we will see shortly way more lightweight) modelling alternative to GPs.

Figure 2 provides another perspective to assess the gains achieved by TrimTuner w.r.t. the considered baselines by reporting the time (Figure 2a) and cost (Figure 2b) savings that TrimTuner (DT variant) achieves to identify a configuration whose accuracy is 90% (or more) of the optimal (feasible) solution. We omit FABOLAS and Random from the plots, which perform poorly, to enhance visualization. Using EIc and EIc/USD, the optimization process lasts up to 65×/15× (resp.) and costs up to 50×/10× more.

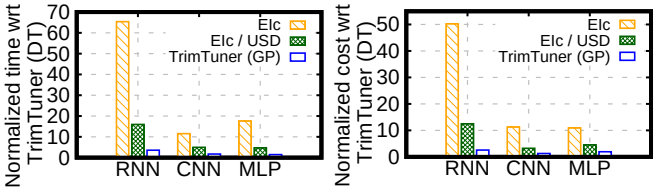
Table III reports the average time to recommend the next configuration to test, allowing us to compare the computational complexity of the considered solutions. As expected, EIc and EIc/USD, which use the simplest acquisition functions, although with GPs, have the lowest computational cost.

[2]https://github.com/pedrogbmendes/TrimTuner.git



(a) RNN (b) MLP (c) CNN

Fig. 1: Accuracy$_C$ for RNN, MLP, and CNN as a function of the optimization cost. The dashed lines correspond to the initialization phase, during which a random sampling policy is used by all solutions.



(a) Time speed ups. (b) Cost savings.

Fig. 2: Time (2a) and cost (2b) savings achieved by TrimTuner when using an ensemble of decision trees.

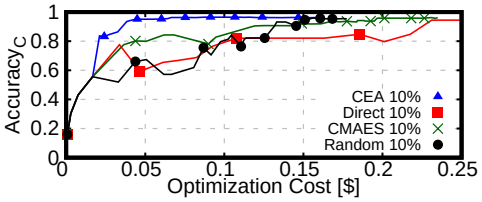


Fig. 3: Comparison of the optimization cost for RNN using TrimTuner (GP variant) using different filtering heuristics.

TABLE III: Average time to recommend a configuration (average of the three data-sets).

Optimizer	Avg. time to recommend a configuration [min]	Standard Deviation
TrimTuner (GPs)	18.65	2.31
TrimTuner (DTs)	1.36	0.28
Fabolas	13.96	1.88
EIc (or EIc/USD)	1.17	0.07

FABOLAS, which also relies on GP-based models, takes approximately one order of magnitude longer to output a recommendation (≈14 minutes) and the GP-based variant of TrimTuner takes approximately 30% longer. It is interesting to see that the DT-based implementation of TrimTuner achieves a 13× speed up compared to the GP variant, attaining a performance almost on par with EIc and EIc/USD.

To sum up, TrimTuner can recommend configurations that achieve high accuracy and meet the cost constraint, while obtaining significant cost and time reductions w.r.t. the considered baselines through the use of sub-sampling.

B. Sensitivity to Filtering Heuristic

The data reported in Table III was obtained by using in both TrimTuner variants the CEA heuristic configured to select 10% (β) of the untested configurations. This section investigates the efficiency of this heuristic and the sensitivity of TrimTuner to the setting of the β parameter.

We start by comparing, in Figure 3, the $Accuracy_C$ that is achieved in RNN when using TrimTuner with GPs and the following heuristics: CEA, two state of the art black-box optimizers (used, e.g., in FABOLAS), namely Direct [12] and CMAES [11] and a simple random policy. For all the heuristics we set the filtering level (β) to 10% and treat the optimization cost as the independent variable.

The plot confirms the cost-efficiency of the proposed heuristic: considering the cost spent to recommend a feasible configuration whose accuracy is 90% (or more) of the optimum, CEA achieves a 3.62× and a 7× savings when compared to CMAES and Direct, respectively.

Table IV allows to evaluate the computational efficiency of CEA by comparing the average time to recommend the next configuration using TrimTuner (both variants) with different heuristics and filtering levels (including no filter). We start by observing that when considering a 10% filtering level, recommending configurations with CEA takes roughly as long as with Random. Note that with a random policy, the time taken to recommend a configuration is dominated by the evaluation of the acquisition function (for the configurations selected by the filtering heuristic). This data confirms that CEA is, indeed, a lightweight filtering heuristic, that is actually more computationally efficient than Direct and CMAES (by up to approximately 2×).

Finally, in Figure 4, we report the results of a sensitivity study on the impact of tuning the filtering level (β) with CEA. As expected, the best results are achieved when the no filtering heuristic is employed. However, this comes at a very high computational cost (see Table IV): if no filtering heuristic is used, it takes on average about two hours to recommend a configuration for the GP variant of TrimTuner; with DTs, that time reduces to approximately 4 minutes, but still remains quite large. Overall, these experimental data confirm the relevance of developing effective filtering heuristics.

As for the sensitivity to the tuning of β, clearly the smaller the number of configurations that the heuristic can select, the worse the performance. Yet, we do not observe a significant degradation for values of β as low as 10%, which motivates the setting employed in the study presented in Section IV-A.

TABLE IV: Average time to recommend the next configuration with different heuristics and filtering levels (RNN, TrimTuner).

Filtering Heuristic	Recommendation time TrimTuner (GPs) [min]	Recommendation time TrimTuner (DTs) [min]
No filter	125.76	3.69
CEA (1%)	5.94	1.07
CEA (10%)	16.85	1.72
CEA (20%)	28.65	2.05
Direct (10%)	36.18	2.63
CMAES (10%)	30.87	2.26
Random (10%)	16.53	1.62

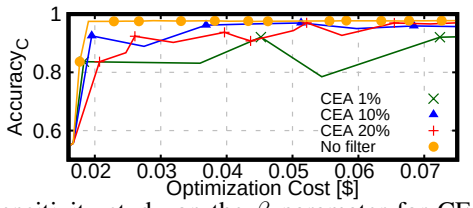


Fig. 4: Sensitivity study on the β paramater for CEA (RNN, DT). The initialization phase does not use CEA and is omitted.

V. Conclusion

We presented TrimTuner, a system for optimizing the training of ML jobs on the cloud that exploits data sub-sampling techniques to enhance the efficiency of the optimization process. TrimTuner builds upon recent systems for hyper-parameter tuning, which it extends by supporting user-defined constraints, and jointly optimizing cloud and model's hyper-parameters. TrimTuner relies on two methods to accelerate the recommendation process: a new heuristic, called Constrained Expected Accuracy (CEA); and the adoption of an ensemble of decision trees to model the cost and accuracy.

Thanks to sub-sampling, TrimTuner reduces the cost and latency of the exploration process by up to 50× and 65×, respectively, whereas the joint use of CEA and decision trees accelerates the recommendation process by up to 117×.

In future work, we plan to extend TrimTuner to cope with alternative optimization problems, e.g., multi-objective optimization of both cost and accuracy, and to evaluate it in problems with multiple constraints.

References

[1] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *ACL*, 2019.

[2] Amazon Web Services, "Amazon EC2 Instance Types," https://aws.amazon.com/ec2/instance-types/, accessed 30 Jun. 2020.

[3] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwanepoel, and D. Garlan, "Lynceus: Cost-efficient tuning and provisioning of data analytic jobs," in *ICDCS*, 2020.

[4] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017.

[5] C. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Arrow: Low-level augmented bayesian optimization for finding the best cloud VM," in *ICDCS*, 2018.

[6] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *NIPS*, vol. 2, 2012, pp. 2951–2959.

[7] J. T. Kent, "Information gain and a general measure of correlation," *Biometrika*, vol. 70, no. 1, pp. 163–173, 1983.

[8] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparamaters on large datasets," in *AISTATS*, vol. 54, 2017, pp. 528–536.

[9] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, pp. 1–52, 2018.

[10] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *ICML*, vol. 80, 2018, pp. 1437–1446.

[11] N. Hansen, *The CMA Evolution Strategy: A Comparing Review*. Springer Berlin Heidelberg, 2006, pp. 75–102.

[12] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian optimization without the lipschitz constant," *J. Optim. Theory Appl.*, vol. 79, no. 1, p. 157–181, 1993.

[13] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[14] ——, *Classification and regression trees*. Routledge, 1984.

[15] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *NSDI*, 2016.

[16] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *SoCC*, 2017, pp. 452–465.

[17] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *ASPLOS*, 2014.

[18] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, "Proteustm: Abstraction meets performance in transactional memory," in *ASPLOS*, 2016.

[19] J. Zeng, P. Romano, J. Barreto, L. Rodrigues, and S. Haridi, "Online tuning of parallelism degree in parallel nesting transactional memory," in *IPDPS*, 2018.

[20] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *VLDB*, vol. 2, no. 1, 2009.

[21] K. Swersky, J. Snoek, and R. P. Adams, "Multi-task bayesian optimization," in *NIPS*, vol. 2, 2013, pp. 2004–2012.

[22] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," Tech. Rep. arXiv:1012.2599, 2010.

[23] M. A. Osborne, R. Garnett, and S. J. Roberts, "Gaussian processes for global optimization," in *LION*, 2009.

[24] P. Henning and C. J. Schuler, "Entropy search for information-efficient global optimization," in *Journal of Machine Learning Research*, vol. 13, no. Jan. JMLR, inc, 2012, pp. 1809–1837.

[25] J. Hernández-Lobato, M. Gelbart, M. Hoffman, R. Adams, and Z. Ghahramani, "Predictive entropy search for bayesian optimization with unknown constraints," in *ICML*, 2015, pp. 1699–1707.

[26] V. Perrone, I. Shcherbatyi, R. Jenatton, C. Archambeau, and M. Seeger, "Constrained bayesian optimization with max-value entropy search," *ArXiv*, vol. abs/1910.07003, 2019.

[27] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," in *Technometrics*, vol. 21, no. 2. Taylor & Francis, Ltd., American Statistical Association, American Society for Quality, 1979, pp. 239–245.

[28] Q. Liu and D. A. Pierce, "A note on gauss-hermite quadrature," in *Biometrika*, vol. 81, no. 3. Oxford University Press, Biometrika Trust, 1994, pp. 624–629.

[29] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," in *Journal of Machine Learning*, vol. 63, 2006, pp. 3–42.

[30] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[31] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," in *IEEE Signal Processing Magazine*, vol. 29, no. 6. IEEE, 2012.

[32] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil, "Fast Direct Methods for Gaussian Processes," *ArXiv*, vol. abs/1403.6015, 2014.