TERA: Optimizing Stochastic Regression Tests in Machine Learning Projects

Saikat Dutta
UIUC
Urbana, USA
saikatd2@illinois.edu

Jeeva Selvam
UIUC
Urbana, USA
jselvam2@illinois.edu

Aryaman Jain
UIUC
Urbana, USA
aryaman4@illinois.edu

Sasa Misailovic
UIUC
Urbana, USA
misailo@illinois.edu

ABSTRACT

The stochastic nature of many Machine Learning (ML) algorithms makes testing of ML tools and libraries challenging. ML algorithms allow a developer to control their accuracy and run-time through a set of hyper-parameters, which are typically manually selected in tests. This choice is often too conservative and leads to slow test executions, thereby increasing the cost of regression testing.

We propose TERA, the first automated technique for reducing the cost of regression testing in Machine Learning tools and libraries (jointly referred to as projects) without making the tests more flaky. TERA solves the problem of exploring the trade-off space between execution time of the test and its flakiness as an instance of Stochastic Optimization over the space of algorithm hyper-parameters. TERA presents how to leverage statistical convergence-testing techniques to estimate the level of flakiness of the test for a specific choice of hyper-parameters during optimization.

We evaluate TERA on a corpus of 160 tests selected from 15 popular machine learning projects. Overall, TERA obtains a geomean speedup of 2.23x over the original tests, for the minimum passing probability threshold of 99%. We also show that the new tests did not reduce fault detection ability through a mutation study and a study on a set of 12 historical build failures in studied projects.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Software Testing, Machine Learning, Bayesian Optimization, Test Optimization

ACM Reference Format:
Saikat Dutta, Jeeva Selvam, Aryaman Jain, and Sasa Misailovic. 2021. TERA: Optimizing Stochastic Regression Tests in Machine Learning Projects. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464844

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464844

1 INTRODUCTION

The growing popularity of Machine Learning (ML) has led to rapid development of general-purpose libraries and specialized tools that build on top of these libraries. These tools perform various tasks in applications like computer vision, natural language processing, and medical diagnosis by implementing algorithms such as Deep Learning [44], Reinforcement Learning [61], or Probabilistic Programming [45, 48]. However, bugs in the implementations of such tools can make the ML-based applications vulnerable to failures and lead to loss of lives and property [47, 103].

Testing of ML libraries and tools is currently not well-understood, which causes the developers to apply ad-hoc techniques when writing tests. An important trait of many ML algorithms – e.g., Reinforcement Learning [100], Bayesian modelling [31], Seq-to-seq learning [99] – is *inherent randomness*, meaning that each execution of the algorithm may produce a slightly different result. Hence, developers often opt to execute such algorithms for long cycles (more than actually necessary) to ensure their results are *highly likely to be close to expected values in tests*, thereby unnecessarily increasing the cost of testing.

An optimized testing procedure for ML algorithms needs to make careful choices. ML algorithms allow a developer to control their accuracy and run-time through a set of *hyper-parameters* – numerical values that guide model selection or define training strategies. Common examples of hyper-parameters for learning algorithms include the number of training iterations, learning rate, and the number of elements sampled from output distribution.

Listing 1 shows a common pattern of tests that check for correctness of an ML algorithm. Such a test typically involves: (1) setup code, e.g., for downloading sample data, initializing test environment on Line 2, (2) initializing a stochastic ML algorithm with one or more hyper-parameters $P_1, \ldots, P_k$ on Line 3, (3) executing the algorithm and computing accuracy metrics on Lines 4-5, and (4) assertions checking if computed metrics are near to or exceed expected values on Lines 6-7. When the developers do not choose the hyper-parameters carefully, these tests can be slow to execute. We observe that such tests are typically more time-consuming than other tests in the test-suite and consume a significant portion of test time, sometimes even exceeding 80% (Section 5).

```
1 def testAlgo():
2     [[setup code]]
3     trainer = MLAlgo( P_1 = v_1, P_2 = v_2, ..., P_k = v_k )
4     trainer.train()
5     metrics = trainer.compute_metrics()
6     for i in range(len(metrics)):
7         assert metrics[i] >= expected[i]
```

Listing 1: Example test pattern

Choosing optimal hyper-parameters is often non-intuitive and difficult for a developer to get right. In this process, developers also need to ensure that their tests are not too *flaky* – pass and fail non-deterministically for the same version of code – due to randomness of the ML algorithm. For instance, if a developer chooses hyper-parameters too conservatively (e.g., selects a large number of training iterations), the test becomes *less flaky* but is too expensive to run. On the other hand, if the developer chooses hyper-parameters too liberally, the test runs faster but can become *more flaky*. Dutta et al. [33] showed that algorithmic randomness is the major cause of flakiness in the ML domain, further signifying the importance of accounting for randomness in tests.

At present, developers have to make ad-hoc decisions and manually select sub-optimal hyper-parameter values. Naturally, they are more inclined to be conservative since they are focused on eliminating flakiness. An important and intriguing challenge then is to find a way to significantly reduce the running time of such tests without making them more flaky.

Our Work. We propose TERA – the first automated technique for reducing the cost of regression testing in ML projects[1] without making the tests more flaky. TERA rests on a (seemingly counter-intuitive) insight that modestly relaxing the desired passing probability of some tests can result in both faster and highly reliable execution of the test suite. To find the optimized version of the tests, TERA systematically navigates the trade-off space between execution time of the test and its passing probability by tuning the algorithm hyper-parameters.

To determine the degree of flakiness of a test, we define a metric *Test Passing Probability* (**TPP**) – probability that a given test passes for the same code version. For its exploration, TERA exposes TPP to the developer as *a tunable knob* $\alpha \in [0, 1]$. For instance, if the developer specifies $\alpha = 0.99$, TERA will try to find a set of optimal hyper-parameter values which minimize the running time of the test without dropping the probability of passing below 99% (or such that $TPP \geq \alpha$). If successful, the optimization will reduce the regular testing time and the build time (which involves running tests across multiple environments) of the project in the continuous integration systems like Travis-CI [22] and CircleCI [3].

TERA formulates the problem of exploration of the trade-off space between the execution time of the test and its passing probability as an instance of *Stochastic Optimization* over the space of algorithm hyper-parameters. Stochastic Optimization [94] encompasses a family of algorithms for optimizing objective functions when randomness is present. The main benefits of Stochastic Optimization are that (1) the optimization can reduce the running time of the test in a black-box fashion (i.e., it does not need to look into the test body); and (2) it automates the selection of algorithm hyper-parameters in a systematic manner. In this work, we use Bayesian Optimization, as an instance of Stochastic Optimization method, to solve the optimization problem.

For a given test, TERA constructs an objective function that executes the test (using a given set of hyper-parameter values) several times and returns the average execution time (the optimization objective) and the test passing probability. However, to construct this objective function, we must address two challenges: (1) How many times to execute the test? and (2) How to estimate the test passing probability? Existing literature on optimization algorithms does not provide mechanisms to automatically develop such stochastic objective functions.

We apply two key techniques to address the challenges above. First, we monitor the values in the assertions of the test while executing the test several times. We then use the samples of the actual values in the assertion (e.g., `metrics[i]` in Listing 1, Line 7) to check whether it converges to the target distribution. We use the convergence property to dynamically determine how many times to execute the test. Second, we approximate the passing probability of the test by computing the passing probability of its assertions. We compute this probability by first estimating the distribution using the samples of the actual values and then computing how likely it is to exceed the expected values (e.g., `expected[i]` in Listing 1, Line 7). We present more details in Section 4.6.

Results. We evaluate TERA on a corpus of 160 tests selected from 15 projects, chosen from four popular ML libraries – PyTorch, TensorFlow, Pyro, PyMC3 – and tools that build on top of them. These tools provide application specific functionalities and have a wide user base, making them an important part of the ML domain. TERA found the optimized configurations for 133 tests. TERA's optimized tests are 2.23x (geo-mean) faster than the original tests, for the passing probability threshold (α) of 0.99 (or 99%). Developers already accepted our optimizations for 24 tests at the time of writing this paper. We performed two studies on the ability of optimized tests to detect faults:

- On a set of mutated programs, we observed that the mutation scores increase slightly on average. We further inspect some mutants and find two key trends, which we discuss in Section 7.1: 1) an optimized test catches a bug missed by original test when faults introduce small variations in computations which are absorbed in longer cycles (original) but detected in tighter executions (optimized) – increasing true positive rate, and 2) optimized test misses a bug when the error accumulation exceeds a certain threshold (more likely in longer cycles) – increasing false positives. TERA's approach of changing hyper-parameters only affects executions which exhibit such small deviations - which are very rare as demonstrated by our mutation study. Hence, the optimized tests retain most of the fault detection ability of original tests. We also discuss a composite test-execution strategy to mitigate the false positives in Section 7.2.
- On a set of 12 historical failures in the project builds, we confirmed that our optimized tests were able to detect the faults in all cases.

These results jointly show that our approach can improve the performance of testing, while retaining the fault detection ability of the optimized tests. We anticipate developers can apply TERA, in addition to existing tests, when a new test of ML algorithm gets added and when such a test fails due to regression errors.

Contributions. This paper makes the following contributions:

- We frame the problem of reducing the running time of tests in Machine Learning projects as an optimization problem over the space of hyper-parameters used in learning algorithms.

[1] By "ML projects" we denote ML-related libraries or tools, in which the outcomes are affected by some stochastic component (e.g., in the algorithm or the data-set that the test generates). Hereon, we will use "projects" to refer to such libraries/tools.

- We propose TERA, an automated approach for optimizing expensive tests by combining Bayesian optimization and statistical testing techniques.
- We evaluate TERA on 160 tests from 15 projects. We show that the optimized versions of the tests run 2.23x times faster while still retaining similar fault detection ability.

We provide a complete replication package containing the source code of TERA and the instructions for reproducing our results at https://github.com/uiuc-arc/tera.

2 EXAMPLE

Listing 2 shows an example test (simplified) for a reinforcement learning algorithm in **ML-Agents** project [10]. **ML-Agents** provides implementations of several training algorithms (like deep reinforcement learning) for training agents in games and simulation environments. We describe the test next.

Lines 2-12 initialize a simple simulation environment (*SimpleEnvironment*) and the training algorithm (**SAC**). Line 13 performs the training step. Lines 14-18 compute the score (rewards) of the trained agent for the given environment and checks if the scores are above the expected value (0.8).

```
1  def test_2d_sac():
2      env = SimpleEnvironment(...)
3      config = TrainerSettings(
4          trainer_type=TrainerType.SAC,
5          hyperparameters=SACSettings(
6              learning_rate=5.0e-3,
7              batch_size=16,
8              ...
9          ),
10         max_steps=10000
11     )
12     trainer = create_trainer(env, config, ...)
13     trainer.start_learning()
14     processed_rewards = [
15         reward_processor(rewards) for rewards in env.final_rewards.values()
16     ]
17     for reward in processed_rewards:
18         assert reward > 0.8
```

Listing 2: Example test from ml-agents

```
1  def sample_mini_batch(batch_size, sequence_length):
2      num_seq_to_sample = batch_size // sequence_length
3      mini_batch = AgentBuffer()
4      ...
5      num_sequences_in_buffer = buff_len // sequence_length
6      start_idxes = (
7              np.random.randint(num_sequences_in_buffer, size=num_seq_to_sample)
8              * sequence_length
9          )# Sample random sequence starts
10     for key in self:
11         mb_list = [self[key][i : i + sequence_length] for i in start_idxes]
12         mini_batch[key].set(list(itertools.chain.from_iterable(mb_list)))
13     return mini_batch
```

Listing 3: Source of Randomness (Batching)

We select this test for optimization with TERA because it uses a machine learning algorithm (**SAC**), which is a reinforcement learning algorithm that makes random choices, requires selecting several hyper-parameters, and contains an approximate assertion (Line 18). These hyper-parameters (like `max_steps`) determine the running time of the test. Developers typically choose these hyper-parameters in an ad-hoc manner, based on their intuition on what seems to be *good enough*. As a result, this test runs longer than what is required (as we show later) to achieve the desired reward (Line 18). The original test takes 90 seconds to run.

```
1  def sample_action(dist):
2      ...
3      continuous_action = dist.sample()
4      return AgentAction(continuous_action)
```

Listing 4: Source of Randomness (Sampling Action)

The Soft-Actor Critic (SAC) Algorithm [52] is a deep Reinforcement Learning algorithm. Reinforcement learning algorithms aim to maximize the expected reward of an agent solving a given task (such as playing a game).

Sources of Randomness. The SAC algorithm involves several sources of randomness. Listing 3, shows the simplified code snippet for a function `sample_mini_batch`, which is used by SAC algorithm [11]. In each iteration, the algorithm uses this function to compute gradients using batches randomly sub-sampled (Lines 6-12) from the agent buffer (which contains traces of the previous steps). Listing 4 shows another function `sample_action`, which is used by SAC [12] to sample the next action (which can either be a discrete choice or a continuous value) using a specified distribution (Line 3) for the agent. Due to these random choices of mini-batches and actions at each step, every execution of the test can yield slightly different results (rewards). Hence, if the developers do not choose optimal hyper-parameters[2], the test can sometimes fail and become unacceptably *flaky*.

Optimizing the test. A naive approach to reduce the running time of the test might be to do binary search on the `max_steps` parameter and choose a value for which the test passes. However, this approach is problematic. First, reducing `max_steps` alone, or any other hyper-parameter may not help us find the optimal runtime. We need to simultaneously adjust other hyper-parameters like `batch_size` and `learning_rate`. Second, running the test once is not enough. We need to ensure the test passes with high probability i.e., is not too *flaky*. These problems make manual test optimization hard for developers.

To optimize this test, we select three hyper-parameters: `learning_rate`, `batch_size`, and `max_steps`. To use TERA, we need to first define a valid range of values for each parameter. This, in turn defines the search space for optimization. For `learning_rate`, we select a continuous range $[1e^{-5}, 1.0]$, which is a few orders below and above the original value ($5e^{-3}$). For `batch_size`, we allow a set of discrete choices $\{2, 4, 6, 8, 16, 32, 64, 128, 256\}$, which are typical batch sizes used in machine learning. For `max_steps` we select a discrete interval: $[100, 10000]$ with increments of 100. We select the

[2]For tests like this, one could argue that a simple way to deal with randomness during testing is to set the seed in the random number generators, which will make the execution more deterministic. The developers can then just execute the test for a much smaller number of steps and reduce the run-time. However, setting the seeds may not always be the right choice: they can be brittle in presence of program changes and can hide bugs [33].

original value (10000) as the upper bound for this interval since we want to reduce the number of steps and consequently the run-time.

Using the parameter specifications above, TERA automatically adds instrumentation blocks to the code, which perform two steps: (1) replace original parameter values with placeholders: e.g. `max_steps = 10000 => max_steps = "<max_steps>"`, and (2) add code to log the actual (`reward`) and expected values (`0.8`) in the assertion. The values in the assertion help TERA reason about how likely is the test to pass over multiple runs.

Next, TERA's Optimizer module constructs a scoring function, that encodes the optimization problem. The scoring function runs the instrumented version of the test with the given parameter values several times and monitors the execution of the test. Then, it inspects the execution trace (the logged assertion values) and determines the probability of passing of the test. This scoring function is called by the Bayesian optimization algorithm, as it systematically explores the search space. For this experiment, we allow the minimum passing probability threshold $\alpha = 0.99$.

TERA reduces the running time of this test to less than 15 seconds. This is over 6x faster than the original test. The optimal configuration TERA found has the following hyper-parameter values: `max_steps` : 2300, `batch_size` : 4, and `learning_rate` : 0.023. To estimate the impact of this optimization on the fault detection capability, we can compute the mutation score post-hoc. In the case of the tests in the **ML-Agents** project, the mutation score after the optimization is slightly above the one of the original tests (62.44% vs 61.16%), indicating that the fault detection capability is not reduced.

3 BACKGROUND

In this section, we introduce necessary background related to Bayesian optimization, which is a Stochastic Optimization method, and convergence testing.

3.1 Bayesian Optimization

Bayesian Optimization [72, 81] is a popular technique used for global optimization of black-box functions. Given a randomized objective function f, we want to find an input $x \in \mathbb{R}^d$ ($d \in \mathbb{N}$) which minimizes the output of f, subject to a set of constraints on the input space, encoded by the functions $g_1, \ldots, g_K$. Formally:

$$\min_{x \in \mathbb{R}^d} f(x), \text{ s.t. } g_i(x) \geq 0, \text{ for } i = 1, \ldots, K$$

Bayesian Optimization algorithms do not make any assumptions about the nature of the objective function and use a prior distribution to model the behavior of the objective function. The user only needs to define the input space of the objective function. The algorithm then evaluates the function using different inputs and updates the prior to form the posterior distribution using the outputs obtained from function evaluations. The posterior distribution is then used to create an acquisition function. The acquisition function is used to select the inputs for the next round such that it maximizes the chances of finding the optimal parameters.Two common choices of prior/posterior distributions include Gaussian Processes, used in Gaussian Process Regression [88], and Kernel Density Estimators (Non-Parametric), used in Tree-Parzen Estimators [25] (which we use in this work). Examples of acquisition functions include the probability of improvement, expected improvement (used in this work), and knowledge gradient. Bayesian Optimization is advantageous over other methods like random/grid/genetic search [81] when the objective function is computationally expensive.

Researchers have previously employed Bayesian Optimization for problems such as compiler auto-tuning [29], compiler testing [66], finding optimal configurations for software systems [74], and program analysis [56, 77].

3.2 Automating Convergence Testing

Given a test which performs stochastic computations, we want to determine how flaky it is. A naive way would be to run the test a large number of times and then check how often it fails. However, this approach is expensive, especially when the test run-time is high or when the test fails rarely.

Convergence testing. Suppose we have an assertion Φ in a test function T. We can determine the probability of passing of the test by computing the probability of passing for this assertion. To compute this probability, we need to reason about the entire distribution of values that the expression in the assertion can evaluate to. Without loss of generality, let us assume we have an assertion of the form: **assert** $x < \gamma$, where x is a variable in the test and γ is a fixed threshold.

We want to estimate the distribution for x so that we can compute the probability of x exceeding γ. We frame this problem as estimating the distribution of an unknown function $\mathcal{F}$, where $\mathcal{F}$ evaluates T, capturing and returning the value of x. We use a sampling-based approach for this problem such that we execute $\mathcal{F}$ several times, obtain a number of samples of x, estimate the distribution from the samples, and compute the probability of passing: $\Pr(\mathcal{F} \leq \gamma)$. This approach involves two main challenges. We need to decide (1) *how many samples* to collect at minimum and (2) whether *we have seen enough samples*.

Several convergence metrics exist in literature [40, 41, 89]. We use the Geweke Diagnostic [41] (similar to [33]) as a heuristic to measure convergence of a set of samples, to solve the second challenge outlined above. Intuitively, the Geweke diagnostic checks whether the mean of, say, the first 10% of samples is not significantly different from, say, the last 50%. If true, then we can say that the distribution has converged. To measure the difference between the two sub-sets of samples, the Geweke diagnostic computes the Z-score (can be essentially considered as standard deviation), which is computed as the difference between the two sample means divided by the standard errors. Equation 1 presents the formula for the Z-score computation for Geweke diagnostic, where a is the early set of samples, b is the later set of samples, $\hat{\lambda}$ is the mean of each set and Var is the variance of each set of samples.

$$z = \frac{\hat{\lambda}_a - \hat{\lambda}_b}{\sqrt{\text{Var}(\lambda_a) + \text{Var}(\lambda_b)}} \tag{1}$$

To use the Geweke Diagnostic as the convergence test, the user needs to specify the minimum desired threshold (which we call the *convergence threshold*). The convergence testing procedure keeps collecting samples (from test runs) until the Geweke Diagnostic drops below the user-specified threshold. Naturally, a lower threshold needs more samples for convergence. Dutta et al. [33] showed

that using a threshold of 1.0 works well for detecting flakiness in ML projects. We use the same threshold 1.0 in our work.

Choosing the minimum number of samples is non-trivial. Too few samples can lead us to incorrect conclusions whereas too many samples can be too expensive to compute (especially when running the test is expensive). The user can choose appropriate number of samples. For our evaluation we use a minimum of 30 samples (guided by existing studies [91] that recommend this number of samples for statistical significance).

4 TERA

4.1 Problem Formulation

We formalize the optimization problem TERA aims to solve as follows. Given a test $T : \emptyset \mapsto \{0, 1\}$, TERA transforms the test to an equivalent variant $T' : \boldsymbol{\theta} \mapsto \{0, 1\}$, which is parameterized by an ordered tuple of hyper-parameters $\boldsymbol{\theta}$. Here, $\boldsymbol{\theta} = (P_1, \ldots, P_k)$, where each P_i ($i \in \{1, \ldots, k\}$) is a tunable hyper-parameter we can optimize and k is the number of hyper-parameters identified in T.

Each parameter is either a discrete integer (e.g., number of iterations) or a continuous value (e.g., learning rate). Therefore, $P_i \in U$ $\forall i \in \{1, \ldots, k\}$, where $U = \mathbb{Z}$ or $\mathbb{R}$ and $\boldsymbol{\theta} \in U_1 \times U_2 \ldots \times U_k$.

We define a function $TPP : (T', \boldsymbol{\theta}) \mapsto [0, 1]$, which takes a transformed test T' and a tuple of hyper-parameters $\boldsymbol{\theta}$ and returns the probability of *passing* of T' when executed using the selected hyper-parameters $\boldsymbol{\theta}$. Additionally, we also define function $Time : (T', \boldsymbol{\theta}) \mapsto \mathbb{R}^+$, which returns the execution time of the test using the selected parameters.

TERA searches for a hyper-parameter tuple $\boldsymbol{\theta}^* \in U_1 \times U_2 \ldots \times U_k$, which when provided as an input to T' minimizes the execution time of test: *Time* (our objective function), given the constraint that the test passes with at-least probability $TPP(T', \boldsymbol{\theta}^*) \geq \alpha$. Formally:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in U_1 \times \ldots \times U_k}{\text{argmin}} \; Time(T', \boldsymbol{\theta}) \quad \text{s.t.} \;\; TPP(T', \boldsymbol{\theta}) \geq \alpha$$

We must address several challenges to solve this optimization problem. First, the nature of the objective function (*Time*) is unknown since we may not have sufficient information about the exact functional form of the given test or the code under test. Hence, we need black-box optimization methods for this problem (Section 4.5). Second, the optimization space of hyper-parameters is typically large, which makes any analytical or enumerative approaches infeasible. Further, evaluating each configuration can be expensive since the test execution can take several minutes and even multiple executions. Hence, we resort to sampling based approaches to find optimal hyper-parameters. However, instead of randomly sampling from the search space, we use Bayesian techniques to sample more efficiently (Section 4.5). Third, since the test execution involves randomness (as shown in Section 2), we need to determine how likely the test is to pass with a given tuple of hyper-parameters $\boldsymbol{\theta}$. A naive strategy is to run the test N times and report how often it passes. However, this can lead to imprecise results. We show how we can apply statistical techniques to both determine *how many times to run the test* and precisely *compute the probability of passing* using the assertions in the test (Listing 1, Lines 6-7). We filter out hyper-parameters which drop the probability of passing below user-specified threshold α. We provide more details in Section 4.6.

Algorithm 1 TERA Algorithm

Input: Test T, Parameters $\boldsymbol{\theta}$, Min. Passing Probability α
Output: Optimized Test T^*, Parameters $\boldsymbol{\theta}^*$

1: **procedure** TERA(T, $\boldsymbol{\theta}$)
2: *Search_Space* ← *Initialize_Search_Space*(P)
3: T' ← *TestInstrumentor*(T, $\boldsymbol{\theta}$)
4: *Optimizer* ← *BayesOpt*(*Search_Space*, *Scorer*, T', MAX_EVALS, TIMEOUT, α)
5: T^*, $\boldsymbol{\theta}^*$ ← *Optimizer.minimize()*
6: **return** T^*, $\boldsymbol{\theta}^*$
7: **end procedure**

4.2 System Overview

We describe how we implement the solution for the optimization problem discussed above in TERA. Algorithm 1 describes the main algorithm for TERA and how it uses the main components. It takes a test T, a tuple of hyper-parameters $\boldsymbol{\theta}$ used in the test, and the minimum test passing probability α as its inputs. TERA consists of four main components:

- The *Test Identifier* finds tests which run inference algorithms or training algorithms and contain one or more tunable parameters. For each parameter in $\boldsymbol{\theta}$, we define the valid range of values for the parameter and how to sample the values (Line 2).
- The *Test Instrumentor* modifies the given test by creating placeholders for hyper-parameters that TERA needs to optimize and adding instrumentation code for logging the actual and expected values in the test assertion (Line 3).
- The *Optimizer* executes the Bayesian Optimization algorithm. It runs a test with different parameter configurations several times and finds an optimal parameter configuration which minimizes the running time of the test. We initialize the *Optimizer* (Line 4) using the defined search space, the scoring function (Scorer), the instrumented test T', and a few hyper-parameters like maximum number of evaluation of test (`MAX_EVALS`) and time limit for optimization (`TIMEOUT`).
- The *Scorer* implements the entire optimization problem. It takes the instrumented test T', a tuple of hyper-parameters $\boldsymbol{\theta}$, and the minimum passing probability α as input. Then it runs the test (with the parameter configuration) several times, records the actual and expected values in the assertion, and computes the probability of the assertion passing. If the probability of passing is greater or equal to the user specified threshold (α), then the Scorer returns the average run time of the tests as output. Otherwise, the Scorer returns infinity (∞). The *Scorer* is passed as the scoring function by the optimization algorithm.

4.3 Test Identifier

To run TERA, we first need to identify parameters in the test which directly affect the running time of the test and the accuracy of the result. For instance, to run an inference algorithm like Stochastic Variational Inference (SVI) in *Pyro*, the developer needs to set the number of iterations to run and the learning rate of the Adam Optimizer (which is a variant of Stochastic Gradient Descent).

Similarly, to run a reinforcement learning algorithm in *ML-Agents*, the developer needs to choose hyper-parameters like the number of iterations, batch size, and learning rate. We identify such parameters manually in the test and use TERA to tune them. For instance, we look for parameters that match the following patterns: *samples*, *iterations*, *epochs*, *batch size*, *learning rate*, *num passes*, and *chains*.

We also need to identify assertions in the test which TERA can use to determine the reliability of the test results. In particular, we look for assertions which perform approximate comparisons between expected and actual values. This notion of approximate assertions is similar to the ones used in previous works [33, 75]. One example of such an assertion is the Python assert statement of the form: `assert` $a <| > | <= | >= b$. Other examples include numpy APIs like `assert_allclose` and `assert_almost_equal`, and unittest APIs like `assertLess` and `assertGreater`. One difference with previous works is that we also consider assertions that check for *exact equality*. However, we limit it to cases where the assertions check some property of a trained model or inference.

Overall, we typically spend 1-2 hours per project on average to identify tests with suitable hyper-parameters and assertions as described above. We anticipate that the developers who have familiarity with their projects will identify such tests much faster.

4.4 Test Instrumentor

For the given test T in a project, and hyper-parameters $\boldsymbol{\theta}$, the Test Instrumentor performs two tasks. First, it replaces the original values of each parameter with a placeholder, which will be used by TERA to set new parameter values and run the test. Second, it adds statements to record the actual and expected values in the test assertions (specified by the user). This step ensures TERA can later reproduce the executions, by simply reading the values from the logs and reason about the distribution of values. The Test Instrumentor can handle logging any scalar, vector, or tensor objects. We use Python's *AST* library [87] to implement the Test Instrumentor.

4.5 Optimizer

TERA uses Bayesian Optimization to optimize the running time of the tests. In this work, we use *Tree Parzen Estimators* (TPE) Algorithm [25], which is a variant of Bayesian Optimization [72]. To use this algorithm, we need to provide:

- **Legal Parameter Values:** First, we need to define the space of legal parameter values that the optimization algorithm uses to sample the parameter values. To use the TPE algorithm, we need to specify a distribution for each parameter that will be used for sampling. We use three kind of parameters spaces in this work: (1) Continuous bounded interval, e.g. $x \in [1e^{-5}, 1.0]$, (2) Discrete bounded interval, e.g. $x \in [100, 1000]$, (3) Discrete choices, e.g. $x \in \{1, 2, 3, 4, 5\}$. For the parameters with continuous bounded interval (e.g. learning rate), we use a *log-uniform* distribution so that it samples values of different orders. For parameters with discrete bounded intervals (e.g. iterations) or discrete choices (e.g. batch size), we use a *uniform* distribution for sampling.
  We manually define the bounds of the distribution based on the kind of parameter. For instance, for parameters like *iterations* and *number of samples*, we choose the upper bound to be the default value of the parameter and lower bound to 100 (or 1 if default is less than 100). For parameters like *learning rate*, we choose the lower and upper bounds as $1e^{-5}$ and 1.0 respectively.
- **Objective Function:** Second, we need to define an objective function, which takes as input a set of new parameter values proposed in an iteration by the algorithm and returns a score which intuitively evaluates the goodness of a given set of parameter values. Since we are concerned with reducing the run time of the tests, we could just return the execution time of the test as the score. However, it is insufficient to run the test once. We must also ensure that the test passes with *high probability*. Otherwise, we might obtain a faster, but highly flaky test.

4.6 Scorer

The Scorer module encodes the optimization problem. Algorithm 2 describes the Scorer algorithm. First, it replaces the parameter placeholders in the instrumented test T' with the actual values in $\boldsymbol{\theta}$ and creates a concrete version of the test T_C (Line 2). Next, it initializes the set of samples S to an empty set (Line 4). Then it iteratively runs the test, collects samples, and computes the score (Lines 5–17). We next describe how the Scorer decides *how many times to run the test* and *how to compute the probability of passing*.

Collecting samples from the distribution. We need to determine whether a version of the test is too flaky. Machine Learning algorithms do not come with formal specifications of accuracy which makes it hard to determine the correctness of any implementation. Hence, we use the assertions in the test as specifications of correctness. The Scorer collects the actual and expected values of the assertion from each run of the test (Lines 7–11). Then it applies the convergence test (Lines 12–15) to determine whether we have enough samples of actual values to reason correctly about the distribution i.e. whether the distribution has converged. If the convergence test fails, we continue running the test more times until the distribution converges. We compute the probability of passing of the test (Line 18). If this computed probability is above or equal to α, then we compute and return the average running time of the test executions (Line 20), otherwise we return ∞ (Line 22).

Computing the probability of passing. Given a set of samples, we need to determine the probability that the assertion passes. To compute the probability of passing, we perform the following steps. First, we fit a distribution to the set of samples. Since, we may not know the exact shape of the distribution, we try and fit a number of distributions and choose the one with the best fit (maximum likelihood). In our experiments we used the following distributions: *normal*, *exponential*, *gamma*, *pareto*, *student-t*, *lognorm*, *log uniform*, *log normal*, and *truncated normal*. In contrast, Dutta et al. [33] used empirical distribution to fit the samples; however, empirical distributions are not suited for computing the tails of a distribution. Next, we compute the probability that a sample from the fitted distribution is within the assertion threshold. For instance, for an assertion of the form: `assert` $x < y$, we obtain a distribution $\mathcal{D}$ fitted on the samples of x. Then we compute the cumulative distribution frequency: $\texttt{CDF}(\mathcal{D}, y)$, which is also the probability of passing of the test : $\Pr(x < y)$. For equality assertions, we only compute the percentage of times the actual and expected values match exactly to derive the probability of passing of the test.

Algorithm 2 Scorer Algorithm

Input: Instrumented test T', Parameters $\boldsymbol{\theta}$, Min. Passing Probability α
Output: Score C

```
1:  procedure SCORER(T', θ, α)
2:      T_C ← setParameters(T', θ)
3:      i ← 0
4:      S ← ∅
5:      while i < MAX_ITERS do
6:          b ← 0
7:          while b < BATCH_SIZE do
8:              sample ← ExecuteTest(T_C)
9:              S ← S ∪ {sample}
10:             b ← b + 1
11:         end while
12:         score ← ConvergenceScore(S)
13:         if score < CONV_THRESHOLD then
14:             break
15:         end if
16:         i ← i + BATCH_SIZE
17:     end while
18:     TPP ← ComputeProbPass(S)
19:     if TPP ≥ α then
20:         return AvgRunTime(S)
21:     end if
22: return ∞
23: end procedure
```

Table 1: Project Details

Project	Description	#Tests	%Time
autokeras [1]	ML architecture tuning	2	2.89%
bambi [2]	Bayesian Modelling	2	13.50%
cleverhans [4]	Adverserial Attacks for ML models	5	75.72%
fairseq [5]	Seq-to-Seq Modelling	2	0.58%
gensim [6]	Topic Modelling Library	10	25.88%
gpytorch [7]	Gaussian Process Modelling	9	44.99%
imbalanced-learn (im.-learn) [9]	Learning over Imbalanced Datasets	2	1.89%
ml-agents [10]	Training ML agents	14	68.04%
numpyro [13]	Probabilistic Programming	13	24.04%
parlai [14]	Dialog AI modelling	29	5.53%
pyGPGO [15]	Bayesian Optimization	3	85.00%
pymc3 [17]	Probabilistic Programming	18	14.85%
pymc-learn [16]	Probabilistic Machine Learning	8	5.82%
pyro [18]	Probabilistic Programming	22	26.36%
sbi [21]	Simulation Based Inference	21	84.28%
Total/Avg		160	31.96%

5 METHODOLOGY

Selection of projects. For this work, we focus on two probabilistic programming systems: Pyro [26, 85] and PyMC3 [84, 93], and two machine learning frameworks: PyTorch [80] and TensorFlow [102]. We look for tests in these projects as well as their dependent projects using GitHub's API. Among the dependent projects [3], we select projects with at least 10 stars and manually inspect them to search for tests. Since TensorFlow and PyTorch have a very large set of dependents, we only inspect the top 30 dependent projects (based on stars) for each. For PyMC3 and Pyro, we inspect 8 and 5 dependent projects respectively. Overall, we end up with 71 unique projects. Out of these, we exclude 14 projects which are not maintained, require special build systems (e.g. bazel) to build/run tests, or need special hardware (e.g. Raspberry Pi).

For each remaining project, we install the project locally and run its test-suite using *pytest* [86] to obtain the test run-times. We then sort the tests based on run-time in decreasing order and inspect them to check if they fit our criteria (Section 4.3). We filter out tests which run for less than five seconds since they are already inexpensive. We also exclude tests which run for more than 15 minutes. These involve tests which are typically run on GPUs on Continuous Integration servers and run considerably slower when run on CPUs (which we use for our evaluation). We exclude tests if their parameters have a low value (e.g. 1-2 iterations). We exclude a project if it has no such expensive tests. We excluded 15 projects based on this criteria. Finally, among the remaining projects, we find several suitable tests in Pyro and PyMC3. Among the dependent projects, we found tests in 3 PyMC3 dependents, 3 Pyro dependents, 4 PyTorch dependents, and 3 TensorFlow dependents. Overall, we find 160 tests in these projects. In these tests, we find 17 unique parameters. The top five parameters (and their occurrences) are *learning-rate* (74), *batch_size* (49), *num_samples* (46), *num_epochs* (31), and *num_steps* (28).

Table 1 shows the details for these projects. Column **Project** presents the base name of the project. Column **Description** presents the main utility of the project. Column **#Tests** presents the number of tests we find in each project. Column **%Time** shows the portion of the total test-suite run-time consumed by the selected tests. *We observe that these tests consume more than 31% of the run-time of the whole test-suite. Hence, optimizing these tests can significantly reduce the run-time of the test-suites.*

Mutation Testing. Mutation Testing [58, 79] is an approach for evaluating the effectiveness of a test suite using artificial injected faults. Mutation testing approaches apply simple mutation operators on source code, e.g. changing arithmetic operators, mutating constants, mutating expressions, etc. We use mutation testing analysis to compare the effectiveness of the original and the optimized versions of the tests.

For each project, we select the subset of tests that we are able to optimize using TERA. We compute the line coverage of this subset of tests (original version) and generate mutant versions of code by applying mutation operators only on the covered lines. We run both the original and optimized test suite on these mutants and compute the mutation score:

$$Mutation\ score = \frac{Mutants\ Killed}{Total\ No.\ of\ Mutants}\%$$

To account for the randomness in the analysis (some mutants may be killed/survive by chance), we run the analysis on each project 20 times, and then report the average and standard deviation of the mutation scores.

Extracting historically failed tests. For a given project, we obtain the set of most recent 200 failed builds on Travis CI. For this step we use the GitHub Actions API [43] to fetch the builds. We use the Travis API [22] when the former is not available for a project. Since TERA's optimization mainly targets algorithm parameters, we focus only on builds which contain one or more assertion errors and filter out builds failing due to configuration errors or syntactic errors (like type error or out of bound errors). For this step, we developed a simple Python script to parse the build logs and search for assertion errors. Next, among the builds with assertion errors, we manually look for tests that failed and contain one or more tunable parameters for ML algorithms. For each such test, we find the failing version of the code from the build information and try to reproduce the failure locally. If this step works, we also find the

[3] We use the dependent "packages" as reported by the GitHub API, which are projects that can compile into reusable libraries. Packages are more likely to be actively maintained by developers and have reasonable test suites.

most recent version of code before the build where the test passes. Finally, we run TERA to optimize these tests using the passing version of code and check if the optimized test reproduces the failure in the failing version of code. Overall, we find 12 such tests.

Experimental Setup. For all our experiments, we used 32 core machines with 3.7 GHz Intel processors and 64 GB memory on Azure. For PyMC3, we used machines with larger memory (144 GB) since its tests are more memory-intensive. For the main algorithm (Section 4.2), we set the maximum evaluation at 5000 and timeout for the search process at 100 minutes every round. We choose 99% as the minimum probability of passing (Section 4.6) in all cases. For the convergence test (Section 3.2), we choose a threshold of 1.0 for the Geweke Diagnostic metric, maximum iterations as 500, initial batch size of 30, and update batch size of 30. We implemented TERA entirely using Python. We used the HyperOpt python package [8, 24] for Bayesian Optimization.

6 EVALUATION

We answer the following research questions:

- **RQ1** How much does TERA reduce the run-time of the tests?
- **RQ2** What is the impact of TERA on the fault detection capability of the tests?
- **RQ3** How does TERA's optimization impact the reproduction of historically failed builds?
- **RQ4** What is the run-time of TERA?

RQ1: Run-time Reduction Obtained by TERA

We apply TERA on 15 projects selected using the methodology from Section 5. For each project, we find tests that have one or more tunable parameters. We identified 160 such tests (most run an inference algorithm or a training algorithm).

Table 2 presents the results for the amount of run-time reduction TERA obtains for the selected tests. Column **Project** presents the project name. Column **#Tests** presents the number of tests we considered. Column **Mean Speedup** presents the geometric mean speedup TERA obtained. Column **Max Speedup** presents the maximum speedup TERA obtained for any test. Column **Original Run-time** presents the total running time of the original version of the tests. Column **Optimized Run-time** presents the total running time of the optimized version of the tests. The last row presents the total number of tests, overall geometric mean speedup, overall maximum speedup, average running time of the original tests, and average run time of the optimized tests.

From Table 2, we observe that TERA significantly reduces the run-time of the tests in a majority of cases. Overall, TERA obtains an average reduction of 2.23x across all projects. For Pyro, TERA obtains the highest average reduction (9.94x), with a maximum of 93.65x (reducing from 322s to 3s) for one test. Out of 160 tests, TERA was able to optimize 133 tests, with more than 10% speedup in 119 cases and more than 50% speedup in 79 cases. The results show TERA can significantly reduce the running time of the tests while still ensuring that the tests pass with high probability.

Tests that TERA optimized. Among the tests that were optimized by TERA, parameters like number of sampling iterations in inference algorithms (like MCMC) and the maximum number of training iterations were mostly reduced. It is commonly known that these parameters directly influence how long the algorithms (and consequently the tests) will run. However, reducing these parameters alone is not sufficient.

Table 2: Run-time improvements of tests obtained by TERA

Project	#Tests	Mean Speedup	Max Speedup	Original Run-time	Optimized Run-time
autokeras	2	1.08x	1.16x	33.40s	30.66s
bambi	2	1.39x	1.95x	56.64s	44.27s
cleverhans	5	1.30x	1.40x	26.74s	20.10s
fairseq	2	1.22x	1.23x	3.97s	3.24s
gensim	10	1.35x	4.52x	162.89s	132.81s
gpytorch	9	1.97x	3.38x	38.45s	17.25s
im.-learn	2	1.43x	1.99x	10.22s	5.93s
ml-agents	14	2.21x	6.17x	811.60s	354.58s
numpyro	13	1.41x	6.82x	279.49s	178.85s
parlai	29	1.10x	2.42x	269.19s	212.71s
pyGPGO	3	3.23x	5.19x	262.87s	54.37s
pymc-learn	8	1.98x	5.08x	494.56s	254.25s
pymc3	18	2.13x	12.78x	469.89s	224.14s
pyro	22	9.94x	93.65x	3039.84s	495.94s
sbi	21	3.22x	7.50x	2221.73s	769.90s
Total/Avg	160	2.23x	93.65x	545.43s	186.60s

We observe that in most cases, TERA finds the configuration with the maximum speedup if it adjusts one or more associated parameters as well. One such parameter is learning rate of optimizers (e.g. Adam [62], Adagrad [105]). The learning rate controls how fast the inference/training updates the weights based on computed gradients in each round. A higher learning rate can often overshoot the optimal point whereas a lower learning rate can make convergence very slow. Another example parameter is batch size. Smaller batch sizes enable faster training through parallelization but can return non-optimal solutions. Large batch sizes can lead to optimal solutions at the cost of slow convergence. These trade-offs hence also influence how long the test needs to run to match expected results. TERA enables the developers to effectively navigate this trade-off space while still ensuring the test passes with high probability.

In our evaluation we find multiple cases where developers significantly over-estimate the number of iterations/samples required for obtaining the desired results in the tests. For instance, in a test for variational inference [19] in Pyro, developers run variational inference on a simple model using a simple loss function: Radial Basis Function (RBF) [20] of size 1. However, the developers initially specified 25000 iterations (learning rate: 2e-4) for inference (which takes 322s to run), which is much more than what is needed for inference to converge. TERA finds that running 100 iterations with learning rate 0.09 is enough for the model to converge and pass the desired accuracy in the test and takes only about 3s. We observe similar patterns in other projects as well, indicating that developers are often too conservative.

Tests that TERA did not optimize. Among the tests for which the speedup was less than 10% (mostly in projects like *parlai*, *autokeras*, *bambi*, and *imbalanced-learn*) we observed that in some cases, TERA did optimize the parameters to some extent, but that alone did not reduce the running time of the tests by much. There can be several reasons behind this. For instance, in some cases, other parts of the test like initialization and setup contribute to the majority of the test run-time. In a few other cases, there are other parameters which affect the running time more, but were not exposed in

Table 3: Mutation testing scores

Project	#Mutants	Original	Optimized
autokeras	274	50.00% (±0.00)	50.00% (±0.00)
bambi	770	60.14% (±3.25)	62.55% (±5.36)
cleverhans	185	62.19% (±0.12)	64.16% (±0.69)
fairseq	3374	16.38% (±2.13)	16.39% (±2.13)
gensim	1075	27.88% (±5.72)	26.53% (±5.53)
gpytorch	555	61.25% (±0.32)	63.32% (±0.68)
im.-learn	457	34.57% (±0.00)	35.23% (±0.00)
ml-agents	724	61.16% (±0.11)	62.44% (±0.03)
numpyro	566	60.60% (±0.00)	61.11% (±1.73)
parlai	335	59.51% (±0.39)	58.81% (±0.00)
pyGPGO	102	67.65% (±0.00)	68.63% (±0.00)
pymc-learn	91	68.68% (±2.49)	74.18% (±2.44)
pymc3	750	48.40% (±0.00)	58.01% (±0.03)
pyro	443	47.40% (±0.00)	48.31% (±0.00)
sbi	346	66.46% (±1.43)	67.65% (±1.46)
Average		52.82%	54.49%

the test itself. In some tests, the parameters were already at their optimum value (like *parlai*), hence making too many adjustments causes the tests to fail more often than the allowable threshold.

Developer Responses. Due to limited time, we randomly sampled projects which had high speedups and spread the pull requests among them. We intended to have over 20% of the tests sampled from the test population. Overall, we selected 37 tests across 7 projects and sent Pull Requests to their developers. So far, 24 tests have been accepted and merged into the projects, 9 rejected, and 4 are still pending developer responses. For the cases that were rejected, the developers thought that the gains (in testing time) were not significant enough for them to accept our changes. Developer responses reflect that they are often open to accepting changes in hyper-parameters in the tests, if they can provide significant gains.

RQ2: Fault Detection Ability of Optimized Tests

Modifying the tests written by developers can impact the capability of the tests in catching regressions in code. In this research question, we study the impact of TERA's optimizations on the fault detection ability of the tests. We describe our approach and the results next.

For each project, we generate several buggy versions (mutants) of the code using the methodology outlined in Section 5. We use the Mutmut tool [73] for mutation testing of our projects. To control the cost of mutation testing, we apply mutations only on code related to main inference or training algorithms. We leave out code for utility functions since they are usually almost equally shared across most tests. For projects with longer running times (*sbi*, *pyro*, and *pymc-learn*) we choose the top 50% of the most optimized tests.

Table 3 shows the average (and standard deviation) of mutation scores across 20 runs for each project. It also shows the number of mutants generated per project (Column **#Mutants**). We observe that the average mutation scores remain the same or improve slightly in 13 out of 15 projects. We also perform Student's t-test [27] to check the hypothesis that mutation score of optimized test-suite is smaller than original. Interestingly, it rejects the hypothesis in 14 cases, including gensim which has high variance. The improvement in mutation scores reflects that optimizing the tests can make them tighter and help them detect more regression bugs, which would otherwise be hidden when running for longer cycles. **PyMC3** is an extreme case, in which the mutation score improves by almost 10%. Our investigation found that **PyMC3** developers often set a very high number of sampling iterations (>5000) in the tests. Thus, small variations in computations (caused by faults in the system) can often remain hidden during long cycles of test execution. However, the end-user will experience regressions in performance when using the tool to solve real-world tasks.

The mutation scores regress by about 1-2% in 2 projects (gensim and parlai). This is not unexpected, since we allow the tests to have a minimum probability of passing of 99% during optimization. As a result, the tests might run for fewer cycles than necessary to catch subtle regression bugs. For example, some faults only surface up when the error propagation exceeds the expected threshold leading to a failure. The developer however can opt for a higher passing threshold if required. We discuss more about such examples from our mutation study in Section 7.1.

Overall, we observe that the mutation scores are roughly around 52-54%, which indicates many mutants survive (i.e. not killed). This behavior can potentially be attributed to the probabilistic nature of the ML algorithms. This means that some mutations can generate valid approximations of the software which still meet the desired accuracy specifications (i.e. tests in our case), as observed previously by Hariri et al. [53] in general approximate software.

Table 4: Reproducing historical failures

Project	Passing SHA	Failing SHA	#Tests	#Reproduced
gensim	0027fb5	3db9406	3	3
ml-agents	82ea74f	3f4b2b5	1	1
numpyro	71532cc	b5d548b5	1	1
pyro	f9dee1e2	7f84f19	2	2
pyGPGO	1c718d	c21120	1	1
sbi	86d9b07	c8aec2f	1	1
sbi	1534cff	fa705c0	3	3
Total			12	12

RQ3: Reproducing Historical Failures

In this research question, we evaluate if we apply TERA to the historical versions of tests in these projects, do they still fail when the original versions failed in historical builds.

We obtain 12 failing tests across 6 projects using the methodology outlined in Section 5. For each test, we run TERA to optimize the test (using the passing version). Finally, we report how many of the optimized tests reproduce the failure in the failing code version.

Table 4 shows the result for this experiment. Column **Passing SHA** shows the commit hash of the version of the code where the test passes. Column **Failing SHA** shows the commit hash of the version of code where the original test fails. Column **#Tests** shows the number of tests which failed in the failing version and we optimized using TERA. Column **#Reproduced** shows the number of tests which were optimized and reproduced the failure in the failing version of code. We observe that in all cases we are able to optimize the original version of the test using TERA. The optimized tests also reproduce the failure in the failing version of code in all cases. This demonstrates TERA's optimized tests can reproduce real failures.

RQ4: Efficiency of TERA

We analyze the amount of time TERA's optimization algorithm takes to find optimal parameters.

Table 5: Running times for optimization

Project	#Tests	Avg. Time	Med. Time	Avg. #Iters	Avg. #Params	Avg. Runs	Avg. Test Run-time
autokeras	2	2m4s	2m4s	6	1.0	30.00	16.70s
bambi	2	2h53m40s	2h53m40s	39	3.0	30.00	28.32s
cleverhans	5	43s	48s	3	1.0	91.50	5.35s
fairseq	2	25s	25s	12	1.0	30.00	1.98s
gensim	10	1m17s	57s	5	1.0	30.00	16.29s
gpytorch	9	5m43s	3m27s	60	2.0	30.00	4.27s
im.-learn	2	3m38s	3m38s	47	1.5	30.00	5.11s
ml-agents	14	44m36s	36m21s	103	2.9	30.73	57.97s
numpyro	13	1h0m34s	51m29s	99	1.7	30.00	21.50s
parlai	29	29m4s	15m49s	94	3.0	34.03	9.28s
pyGPGO	3	8m26s	3m42s	2	1.0	30.00	87.62s
pymc-learn	8	2h36m1s	3h30m38s	25	1.6	34.29	61.82s
pymc3	18	3h16m18s	3h28m10s	46	3.1	30.00	26.10s
pyro	22	17m2s	5m25s	21	2.0	45.63	138.17s
sbi	21	1h10m53s	1h3m7s	84	1.9	51.69	105.80s

Table 5 presents the measurements. Column **Avg. Time** shows the average time taken by TERA for a complete run of the optimization algorithm (Section 4.2) – i.e., until it either exhausts evaluating all configurations in the search space or reaches a terminating condition such as exceeding maximum function evaluations (`MAX_EVALS`) or exceeding the allotted time limit (`TIMEOUT`). Column **Med. Time** shows the median optimization time. Column **Avg. #Iters.** shows the average number of iterations taken by the optimization algorithm. Column **Avg. #Params** shows the average number of parameters per test. Column **Avg. Runs** shows the average number of test runs in a single optimization round. Column **Avg. Test Run-time** shows the average run-time of the original test.

For 10 projects, the average optimization time is less than an hour. For 5 projects (sbi, bambi, pymc-learn, numpyro, and pymc3), the average optimization time is more than an hour. **PyMC3** and **Bambi** have a higher average number of parameters, and we chose a wide range of legal values for each parameter. A developer with more domain experience could select a smaller range of values. **Numpyro** also requires high number of iterations due to its large search space. **Sbi**'s tests have a high running time (>100 seconds) and the flakiness of the tests increases TERA's iterations. Similarly, **pymc-learn** also has tests with high run-time (>60 seconds).

7 DISCUSSION

7.1 Fault Detection Ability of Optimized Tests

Modifying the tests written by developers can have an adverse impact on the fault detection ability of tests [96]. We can characterize the fault detecting effectiveness of a test using the following metrics: True Positive (TP) – failing on real faults, False Positive (FP) – failing when no faults, True Negative (TN) – not failing on no faults, and False Negative (FN) – not failing on real faults. TERA improves or retains the TP rate of the test in most cases, as shown by our mutation study from RQ2, and can increase the FN rate in some cases (where the mutation score drops). Since we use a minimum passing threshold of 99%, the FP rate may increase slightly (similarly TN rate would reduce by a small amount). Even though the optimized tests may regress in some of these factors, the gap in effectiveness is very small in practice and would only miss faults which require very rare executions to manifest as test failures. These observations along with our study on reproducing historical bugs from RQ3 show that TERA's optimized tests are highly reliable. We discuss a strategy to alleviate some of these adverse effects in Section 7.2.

We manually inspect some mutants from our mutation study for cases: 1) when our optimized test catches a bug missed by original test and 2) when our optimized test misses a bug caught by original test. We identified two trends: for the first case, we observe that small variations in computations (due to faults in the system) can remain undetected during long cycles of execution in the original test, whereas they are detected by the optimized version which has a tighter execution since it runs for fewer cycles. For the second case, we observe that some faults only manifest as failures when the error accumulated exceeds a certain threshold – these are detected by the original test which runs for sufficient cycles but are sometimes missed by the optimized test.

7.2 Composite Test Running Strategies

We can mitigate some of the adverse affects of optimization by using composite running strategies. For instance, we can *re-run on failure*: first run the optimized test (which is correct with probability α), and if it fails, run the original test, which succeeds with typically higher probability ($> \alpha$). This composite execution is, on average, faster than executing the original test, and can still retain the effectiveness of the original test. In this case, the expected run-time of the test will be: $\alpha \cdot T_{opt} + (1-\alpha) \cdot (T_{opt} + T_{orig})$, where T_{opt} and T_{orig} are the run-times of the optimized and original versions of the test respectively. Since we typically restrict $1 - \alpha$ to a small value (e.g., less than 1%), the second term does not increase the run-time significantly. For instance, for Pyro this would only increase the total run-time of optimized tests from 495.94s to only 526.34s. Overall, using this composite strategy increases the running time of our optimized tests across all projects by only 3% on average. This strategy can help reduce flaky failures (false positives) and increases the chances of detecting genuine faults (true positives).

7.3 Comparison of Different Search Methods

In this work, we use Bayesian optimization for efficiently searching for optimal hyper-parameters (Section 4.2). Researchers also

Table 6: Comparison of different search methods

Project	BayesOpt		Random		Binary	
	$\mathbf{Spd}_{\text{test}}$	$\mathbf{T}_{\text{TERA}}$	$\mathbf{Spd}_{\text{test}}$	$\mathbf{T}_{\text{TERA}}$	$\mathbf{Spd}_{\text{test}}$	$\mathbf{T}_{\text{TERA}}$
autokeras	1.08x	2m4s	1.07x	2m18s	1.00x	19s
bambi	1.39x	2h53m40s	1.32x	3h34m16s	1.19x	1h11m9s
cleverhans	1.30x	43s	1.17x	1m20s	1.25x	1m23s
fairseq	1.22x	25s	1.10x	4m22s	1.01x	43s
gensim	1.35x	1m17s	1.26x	2m55s	1.27x	53s
gpytorch	1.97x	5m43s	1.64x	9m5s	1.60x	1m8s
im.-learn	1.43x	3m38s	1.29x	7m5s	1.07x	1m0s
ml-agents	2.21x	44m36s	2.11x	1h10m4s	2.09x	8m23s
numpyro	1.41x	1h0m34s	1.40x	37m20s	1.38x	10m32s
parlai	1.10x	29m4s	1.06x	33m8s	1.05x	36s
pyGPGO	3.23x	8m26s	3.20x	24m38s	3.17x	10m52s
pymc-learn	1.98x	2h36m1s	1.50x	3h10m21s	1.46x	1h55m15s
pymc3	2.13x	3h16m18s	2.06x	3h38m9s	1.92x	1h53m27s
pyro	9.94x	17m2s	7.70x	1h27m16s	2.71x	29m4s
sbi	3.22x	1h10m53s	1.72x	53m49s	1.60x	11m21s
Avg	2.23x	58m27s	1.89x	1h12m20s	1.59x	26m59s

Here, $\mathbf{Spd}_{\text{test}}$ is the Avg. Speedup (Geo-mean) of the optimized tests and $\mathbf{T}_{\text{TERA}}$ is the Avg. Time (Arithmetic Mean) that TERA takes per project and per search method.

commonly use other search methods such as random search or binary search. We compare our main optimization results against a version of TERA which uses these two alternatives instead of Bayesian optimization.

Random search is a method which uniformly samples from the search space of hyper-parameters to find optimal results. For random search method, we use the same configuration for TERA as the main evaluation (see Section 5). Binary search method evaluates the middle element in the value interval for a given parameter (such as iterations) and proceeds with either half of the interval depending on whether the objective function evaluates to true (choose lower half) or false (choose upper half). The search continues until the interval is reduced to a single element. Since binary search cannot optimize multiple parameters simultaneously, our implementation optimizes one parameter at a time (keeping others fixed), then uses the optimal value found when optimizing the next parameter. We only choose parameters with a bounded discrete interval such as *iterations* and *number of samples* for optimization using binary search. We ignore parameters such as *learning rate*, since optimizing such parameters in isolation has no direct effect on test's run time.

Table 6 presents the results for this experiment. First column shows the name of the project. For each search method, its two sub-columns show the average speedup (geometric mean) of the optimized test and average time (arithmetic mean) for running the optimization algorithm, respectively.

We observe that Bayesian optimization outperforms both random search and binary search methods. Although random search reduces the execution time of tests, it finds a less optimal parameter setting than Bayesian optimization for all projects. This is not surprising since random search, unlike Bayesian optimization, does not learn from results obtained in earlier rounds to adapt the search process. Compared to Bayesian optimization, random search takes more time to finish in 13 projects and less time in 2 projects. We observe that binary search is less effective overall and provides lower speedups in all projects than Bayesian optimization and in 13 projects than random search. It is faster than other methods since it evaluates fewer parameter values or combinations of parameter values. We conclude that binary search may only be suitable for tests with few parameters with discrete bounded intervals.

Table 7: Savings on build/test time per day using TERA

Project	Builds/day	Savings/day
autokeras	1	2s
bambi	1	12s
cleverhans	1	6s
fairseq	2	1s
gensim	1	30s
gpytorch	1	21s
im.-learn	1	4s
ml-agents	11	1h23m47s
numpyro	5	8m23s
parlai	18	16m56s
pyGPGO	1	3m28s
pymc-learn	1	4m0s
pymc3	1	4m5s
pyro	1	42m23s
sbi	1	24m11s

7.4 Gains of Optimization

We anticipate that the cost of running TERA can be easily amortized through the daily savings developers will get in build/test time on CI servers. Table 7 shows the number of builds developers currently trigger per day (**Builds/day**) and the savings developers would get if they use TERA's optimized tests instead of original tests for their builds (**Savings/day**). We compute **Savings/day** as: $\textit{Builds/day} \times (\textit{Original Run-Time} - \textit{Optimized Run-Time})$. The *Original* and *Optimized* run-times can be obtained from Table 2.

We observe that TERA can provide large savings for the developers in many projects – more than 80 mins/day for *ml-agents* and more than 40 mins/day for *pyro*. These gains are further enhanced with increasing builds per day. Finally, we expect that developers will run TERA offline (e.g. outside of normal working hours) without impacting their time.

7.5 Threats to Validity

In this work, we study only a subset of projects in the ML domain, so our results and observations may not generalize to all projects. To mitigate this threat, we focus on four widely used machine learning frameworks, and their top starred dependent projects, which indicates they have a large user-base and are popular.

We may have missed some tests, in the studied projects, which use ML algorithms and have tunable parameters. To account for this risk, multiple student co-authors independently studied these projects and their test-suites to find tests which fit our criteria. As a result, we obtain a substantial number of such tests.

Optimization is a hard problem which makes it difficult to find the best solution for a given problem. As such, it is possible to further enhance the reduction in run-time.

Changing the developer-set parameters in a test can make the test less reliable (or more *flaky*). We mitigate this risk in two ways. First, we set the minimum passing probability to 99% during optimization, which ensures test quality does not regress too much. Second, we perform mutation testing of the test suites and show the optimized suite is commonly as good as the original suite. Further, since mutations may not represent real errors, we also show that optimized tests can reproduce real historical failures.

8 RELATED WORK

Test Reduction. There is significant research on reducing the test size in terms of the lines of code while preserving the test coverage [49, 59, 101] and reproducing the same bugs [90, 109]. Most of these approaches assume that the test and the program-under-test are deterministic (either natively, or with fixed seeds) and the lines of code are the proxy for the execution time. In contrast, TERA reduces the execution time of tests for machine learning algorithms. TERA finds the optimal parameters (such as learning rates or the numbers of iterations) that lead to reduced execution times of the tests with minor impact on the test's fault-detection ability. To the best of our knowledge, the only existing approach for reducing the parameters of machine learning algorithms along with the program code for the purpose of testing is Storm [35] (for probabilistic programming languages). However, Storm's reduction of parameters uses a simple binary search, and does not consider the scenario of optimizing test run-times.

AutoML Methods. Automated Machine Learning (or AutoML) is a novel approach for automated construction of an end-to-end ML pipeline, using limited computational budget [55, 108]. AutoML methods deal with data preparation, feature engineering, model generation, and model evaluation. The model generation step involves selecting from a set of suitable ML architectures (Architecture Optimization) and choosing optimal model specific hyper-parameters (Hyper-Parameter Optimization) [37, 113]. In these steps, AutoML methods typically aim to optimize the accuracy of the model on a data-set. Unlike AutoML, TERA targets the dual problem of reducing the running time of a test executing a *fixed* ML architecture while *also* preserving the desired passing probability of the test.

Hyper-parameter Tuning For Machine Learning. Bergstra et al. [25] explored various strategies to optimize hyper-parameters for neural networks. They showed that Gaussian Process based Bayesian Optimization methods and the newly proposed Tree Parzen Estimator Algorithm perform better than manual or random search based methods on several difficult data-sets. Snoek et al. [98] proposed a new algorithm based on a Gaussian process based surrogate model for Bayesian Optimization for Machine Learning Algorithms. Their approach also accounts for the cost for each configuration of the learning algorithm during optimization by considering the expected improvement per second in the acquisition function. Maclaurin et al. [69] introduced a gradient based hyper-parameter optimization technique. Unlike those use-cases, the objective of TERA is to improve performance of software testing, which it accomplishes by maintaining a desired level of reliability (i.e. minimum probability of passing) using statistical machinery while reducing the running time of the test.

Flaky Tests. Flaky tests have emerged as an important problem in software testing – several studies characterized and classified such tests in real-world projects [50, 54, 63, 68, 78, 92, 110], and are considered an important class of bugs in industry [54, 63]. Researchers also developed automated tools to detect [23, 33, 38, 65, 95, 106], and fix flaky tests caused due to test-order dependency [97] and under-determined specifications [111].

Prior work has studied the causes and fixes for flaky tests in open-source software [68, 78]. They studied flaky tests in traditional software, finding that common causes for flaky tests include async wait, concurrency, and test-order dependencies. Romano et al. [92] studied flaky UI tests in web and Android projects- their causes, manifestations, and fixes. In this work, we optimize tests in ML projects while modestly relaxing the desired passing probability of the test. While reducing parameters like number of iterations could potentially make the tests more flaky, TERA ensures that the probability of passing does not degrade beyond an acceptable threshold chosen by the developer.

Lam et al. [64] proposed a technique to handle flakiness in tests due to asynchronous calls. They show that the running time of such tests can be reduced significantly while still retaining similar test failure rate (or flakiness). They run each test only a fixed number of times to determine how often it fails and use simple binary search for finding optimal timeout times. TERA, on the other hand, handles tests which are flaky due to algorithmic randomness. TERA uses convergence tests to determine how many times to run the tests and Bayesian optimization to search for optimal hyper-parameters.

Dutta et al. [34] proposed a method to fix flaky tests in Machine Learning projects by only updating assertion bounds in tests. In contrast, in this work we look at the dual problem of speeding up the test by tuning the hyper-parameters whist preserving a high passing probability. While both hyper-parameters and assertion thresholds can affect the flakiness of a test, only hyper-parameters influence the execution time of the test.

Testing of Systems Dealing with Randomness. Robust machine learning frameworks like TensorFlow [102] and PyTorch [80] have paved the way for rapid development of machine learning based solutions. In recent times, there has also been a surge in interest in probabilistic programming in both academic and industrial research communities. This has led to the development of numerous probabilistic programming languages over the years [28, 30, 39, 42, 45, 46, 70, 71, 76, 82, 85, 104, 107]. Researchers proposed techniques for testing and verifying probabilistic systems [32, 67], machine learning frameworks [36, 51, 57, 83, 112], and randomized algorithms [60] to complement manual test writing. These techniques complement manual test development, but the advances in efficient automated test generation for these systems is yet to catch up with the speed of application development, while capturing the inherent nondeterminism and overcoming the lack of reliable oracles in this domain.

9 CONCLUSION

We presented TERA, an approach to help developers optimize the running time of the tests which involve stochastic computations. TERA combines techniques from Bayesian Optimization and statistical convergence testing to effectively reduce the running time of the tests while guarding their reliability. Using TERA we obtained more than 2.23x average speedup in 160 tests across 15 projects in Machine Learning domain. We anticipate developers will use TERA for the following main tasks: (1) optimize existing expensive tests, (2) optimize parameters of newly added tests, (3) update hyper-parameters after test modification.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their feedback. This work was supported in part by NSF, Grants No. CCF-1703637 and CCF-1846354, Microsoft Azure, and Facebook PhD Fellowship.

REFERENCES

[1] 2021. Autokeras. https://github.com/keras-team/autokeras.
[2] 2021. Bambi. https://github.com/bambinos/bambi.
[3] 2021. CircleCI. https://circleci.com.
[4] 2021. Cleverhans. https://github.com/tensorflow/cleverhans.
[5] 2021. Fairseq. https://github.com/pytorch/fairseq.
[6] 2021. Gensim. https://github.com/RaRe-Technologies/gensim.
[7] 2021. Gpytorch. https://github.com/cornellius-gp/gpytorch.
[8] 2021. HyperOpt: Hyperparameter Optimization. https://github.com/hyperopt/hyperopt.
[9] 2021. imbalanced-learn. https://github.com/scikit-learn-contrib/imbalanced-learn.
[10] 2021. ML-Agents. https://github.com/Unity-Technologies/ml-agents.
[11] 2021. ML-Agents minibatch. https://github.com/Unity-Technologies/ml-agents/blob/0e573a1865d0800ad5cd6649b9bdec99327028a1/ml-agents/mlagents/trainers/sac/trainer.py#L251.
[12] 2021. ML-Agents sampleaction. https://github.com/Unity-Technologies/ml-agents/blob/master/ml-agents/mlagents/trainers/torch/action_model.py#L70.
[13] 2021. numpyro. https://github.com/pyro-ppl/numpyro.
[14] 2021. ParlAI. https://github.com/facebookresearch/ParlAI.
[15] 2021. PyGPGO. https://github.com/josejimenezluna/pyGPGO.
[16] 2021. PyMC-Learn. https://github.com/pymc-learn/pymc-learn.
[17] 2021. PyMC3. https://github.com/pymc-devs/pymc3.
[18] 2021. Pyro. https://github.com/pyro-ppl/pyro.
[19] 2021. Pyro Test for Variational Inference. https://github.com/pyro-ppl/pyro/blob/25368f56c984506a46e412a9017c0d8fa43fd0c6/tests/infer/test_inference.py#L288.
[20] 2021. Pyro Test using RBD Kernel. https://github.com/pyro-ppl/pyro/blob/25368f56c984506a46e412a9017c0d8fa43fd0c6/tests/infer/test_inference.py#L292.
[21] 2021. Sbi. https://github.com/mackelab/sbi.
[22] 2021. Travis-CI. https://travis-ci.org.
[23] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*.
[24] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML*.
[25] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *NeurIPS*.
[26] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* (2019).
[27] C Alan Boneau. 1960. The effects of violations of assumptions underlying the t test. *Psychological bulletin* (1960).
[28] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, et al. 2016. Stan: A probabilistic programming language. *JSTATSOFT* 20, 2 (2016).
[29] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *ICSE*.
[30] Guillaume Claret, Sriram Rajamani, Aditya Nori, Andrew Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *FSE*.
[31] Peter Congdon. 2014. *Applied bayesian modelling*. John Wiley & Sons.
[32] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *FSE*.
[33] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *ISSTA*.
[34] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds. In *FSE*.
[35] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *FSE*.
[36] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*.
[37] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research* (2019).
[38] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *ICST*.
[39] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*.
[40] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis*. Chapman and Hall/CRC.
[41] John Geweke. 1991. *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments*. Federal Reserve Bank of Minneapolis, Research Department Minneapolis (Staff Report 148), MN.
[42] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* (1994).
[43] Github Actions 2020. https://github.com/features/actions.
[44] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. MIT Press Cambridge.
[45] Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: a language for generative models. In *UAI*.
[46] Noah D Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages.
[47] 2016. A Google self-driving car caused a crash for the first time. *The Verge* (2016). https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report.
[48] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *FoSE*.
[49] Alex Groce, Josie Holmes, and Kevin Kellar. 2017. One test to rule them all. In *ISSTA*.
[50] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An empirical study of flaky tests in python. In *ICST*.
[51] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *ASE*.
[52] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*.
[53] Farah Hariri, August Shi, Owolabi Legunsen, Milos Gligoric, Sarfraz Khurshid, and Sasa Misailovic. 2018. Approximate Transformations as Mutation Operators. In *ICST*.
[54] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *SCAM*.
[55] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A Survey of the State-of-the-Art. *Knowledge-Based Systems* (2021).
[56] Kihong Heo, Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2018. Adaptive Static Analysis via Learning with Bayesian Optimization. *TOPLAS* (2018).
[57] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: A mutation testing framework for deep learning systems. In *ASE*.
[58] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *TSE* (2010).
[59] James A Jones and Mary Jean Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *TSE* (2003).
[60] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical algorithmic profiling for randomized approximate programs. In *ICSE*.
[61] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* (1996).
[62] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[63] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*.
[64] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *ICSE*.
[65] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *ICST*.
[66] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *OOPSLA* (2015).
[67] Yamilet R Serrano Llerena, Marcel Böhme, Marc Brünink, Guoxin Su, and David S Rosenblum. 2018. Verifying the long-run behavior of probabilistic system models in the presence of uncertainty. In *FSE*.
[68] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.
[69] Dougal Maclaurin, David Duvenaud, and Ryan Adams. 2015. Gradient-based hyperparameter optimization through reversible learning. In *ICML*.
[70] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint 1404.0099* (2014).
[71] T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2013. Infer.NET 2.5. Microsoft Research Cambridge. http://research.microsoft.com/infernet.
[72] Jonas Mockus. 2012. *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media.
[73] Mutmut: Python mutation tester 2020. Mutmut: Python mutation tester. https://github.com/boxed/mutmut.
[74] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding faster configurations using flash. *TSE* (2018).
[75] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *ASE*.

[76] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*.

[77] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. *OOPSLA* (2015).

[78] Fabio Palomba and Andy Zaidman. 2017. Does Refactoring of Test Smells Induce Fixing Flaky Tests?. In *ICSME*.

[79] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*.

[80] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*.

[81] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. 1999. BOA: The Bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*.

[82] Avi Pfeffer. 2001. IBAL: a probabilistic rational programming language. In *IJCAI*.

[83] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE*.

[84] Pymc3WebPage 2020. Pymc3. https://github.com/pymc-devs/pymc3.

[85] PyroWebPage 2020. Pyro. http://pyro.ai.

[86] Pytest 2020. https://docs.pytest.org/en/stable.

[87] Python AST package 2021. https://docs.python.org/3/library/ast.html.

[88] Joaquin Quiñonero-Candela and Carl Edward Rasmussen. 2005. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research* (2005).

[89] Adrian E Raftery and Steven M Lewis. 1995. The number of iterations, convergence diagnostics and generic Metropolis algorithms. *Practical Markov Chain Monte Carlo* (1995).

[90] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*.

[91] John A Rice. 2006. *Mathematical statistics and data analysis*. Cengage Learning.

[92] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-based Flaky Tests. In *ICSE*.

[93] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* (2016).

[94] Johannes Schneider and Scott Kirkpatrick. 2007. *Stochastic optimization*. Springer Science & Business Media.

[95] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *ICST*.

[96] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating test-suite reduction in real software evolution. In *ISSTA*.

[97] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *FSE*.

[98] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *NeurIPS*.

[99] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215* (2014).

[100] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[101] Sriraman Tallam and Neelam Gupta. 2005. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes* (2005).

[102] TensorFlowWebPage 2020. TensorFlow. https://www.tensorflow.org.

[103] 2016. Understanding the fatal Tesla accident on Autopilot and the NHTSA probe. *electrek* (2016). https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe.

[104] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv* (2016).

[105] Rachel Ward, Xiaoxia Wu, and Leon Bottou. 2019. Adagrad stepsizes: sharp convergence over nonconvex landscapes. In *ICML*.

[106] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2021. Probabilistic and Systematic Coverage of Consecutive Test-Method Pairs for Detecting Order-Dependent Flaky Tests. In *TACAS*.

[107] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *AISTATS*.

[108] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Hu Yi-Qi, Li Yu-Feng, Tu Wei-Wei, Yang Qiang, and Yu Yang. 2018. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306* (2018).

[109] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *TSE* (2002).

[110] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *TSE* (2020).

[111] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-Specific Fixes for Flaky Tests with Wrong Assumptions on Underdetermined Specifications. In *ICSE*.

[112] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *FSE*.

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