Open Source Vizier: Distributed Infrastructure and API for Reliable and Flexible Blackbox Optimization

Xingyou Song, Sagi Perel, Chansoo Lee, Greg Kochanski, Daniel Golovin

Google Research, Brain Team

Abstract Vizier is the de-facto blackbox and hyperparameter optimization service across Google, having optimized some of Google's largest products and research efforts. To operate at the scale of tuning thousands of users' critical systems, Google Vizier solved key design challenges in providing multiple different features, while remaining fully fault-tolerant. In this paper, we introduce Open Source (OSS) Vizier, a standalone Python-based interface for blackbox optimization and research, based on the Google-internal Vizier infrastructure and framework. OSS Vizier provides an API capable of defining and solving a wide variety of optimization problems, including multi-metric, early stopping, transfer learning, and conditional search. Furthermore, it is designed to be a distributed system that assures reliability, and allows multiple parallel evaluations of the user's objective function. The flexible RPC-based infrastructure allows users to access OSS Vizier from binaries written in any language. OSS Vizier also provides a back-end ("Pythia") API that gives algorithm authors a way to interface new algorithms with the core OSS Vizier system. OSS Vizier is available at https://github.com/google/vizier.

1 Introduction

Blackbox optimization is the task of optimizing an objective function f where the output $f(x)$ is the only available information about the objective. Due to its generality, blackbox optimization has been applied to an extremely broad range of applications, including but not limited to hyperparameter optimization (He et al., 2021), drug discovery (Shields et al., 2021), reinforcement learning (Parker-Holder et al., 2022), and industrial engineering (Zhang et al., 2020).

Google Vizier (Golovin et al., 2017) is the first hyperparameter tuning service designed to scale, and has thousands of monthly users both on the research[1] and production side at Google. Since its inception, Google Vizier has run millions of blackbox optimization tasks and saved a significant amount of computing and human resources to Google and its customers.

Figure 1: Vizier: An advisor.

This paper describes Open Source (OSS) Vizier, a standalone Python implementation of Google Vizier's APIs. It consists of a *user API*, which allows users to configure and optimize their objective function, and a *developer API*, which defines abstractions and utilities for implementing new optimization algorithms. Both APIs consist of Remote Procedure Call (RPC) protocols (Section 3) to allow the setup of a scalable, fault-tolerant and customizable blackbox optimization system, and Python libraries (Sections 4.3 and 6) to abstract away the corresponding RPC protocols.

Compared to (Golovin et al., 2017), OSS Vizier features an evolved backend design for algorithm implementations, as well as new functionalities such as conditional search and multi-objective optimization. OSS Vizier's RPC API is based on Vertex Vizier[2], making OSS Vizier compatible with any framework which integrates with Vertex Vizier, such as XManager[3].

[1] A list of research works that have used Google Vizier can be found in Appendix C.

[2] https://cloud.google.com/vertex-ai/docs/vizier/overview. [3] https://github.com/deepmind/xmanager.

Due to the existence of 3 different versions (Google, Vertex/Cloud, OSS) of Vizier, to prevent confusion, we explicitly refer to the version (e.g. "Google Vizier") whenever Vizier is mentioned. We summarize the distinct functionalities of each version of Vizier below:

- Google Vizier: C++ based service hosted on Google's internal servers and integrated deeply with Google's internal infrastructure. The service is available only for Google software engineers and researchers to tune their own objectives with a default algorithm.
- Vertex/Cloud Vizier: C++ based service hosted on Google Cloud servers, available for external customers + businesses to tune their own objectives with a default algorithm.
- OSS Vizier: Fully standalone and customizable code that allows researchers to host a Python-based service on their own servers, for any downstream users to tune their own objectives.

2 Problem and Our Contributions

Blackbox optimization has a broad range of applications. Inside Google, these applications include: optimizing existing systems written in a wide variety of programming languages; tuning the hyperparameters of a large ML model using distributed parallel processes (Verbraeken et al., 2020); optimizing a non-computational objective, which can be e.g. physical, chemical, biological, mechanical, or even human-evaluated (Kochanski et al., 2017). Generally, such objectives $f(x)$ we are interested in optimizing possess a moderate number (e.g. several hundred) of parameters for the input x, may produce noisy evaluation measurements, and may not be smooth or continuous.

Furthermore, the blackbox optimization workflow greatly varies depending on the application. The evaluation latency can be anywhere between seconds and weeks, while the budget for the number of evaluations, or `Trials`, varies from tens to millions. Evaluations can be done asynchronously (e.g. ML model tuning) or in synchronous batches (e.g. wet lab experiments). Furthermore, evaluations may fail due to transient errors and should be retried, or may fail due to persistent errors (e.g. $f(x)$ cannot be measured) and should not be retried. One may also wish to stop the evaluation process early after observing intermediate measurements (e.g. from a ML model's learning curve) in order to save resources.

To handle all of these scenarios, OSS Vizier is developed as a **service**. The service architecture does not make assumptions on how `Trials` are evaluated, but rather simply specifies a stable API for obtaining suggestions $x_1, x_2, \ldots$ to evaluate and report results as `Trials`. Users have the freedom to determine when to request trials, how to evaluate trials, and when to report back results.

Another advantage of the service architecture is that it can collect data and metrics over time. Google Vizier runs as a central service, and we track usage patterns to inform our research agenda, and our extensive database of runs serves as a valuable dataset for research into meta-learning and multitask transfer learning. This allows users to transparently benefit from the resulting improvements we make to the system.

2.1 Comparisons to Related Work

Table 1 contains a non-comprehensive list of open-source packages for blackbox optimization, focusing on hyperparameter tuning. Overall, OSS Vizier API is compatible with many of the features present in other hyperparameter tuning open-source packages. We did not include commercial services for hyperparameter tuning such as Microsoft Azure, Amazon SageMaker, SigOpt and Vertex Vizier. For a comprehensive review of hyperparameter tuning tools, see (He et al., 2021). There are many other blackbox optimization tools not mentioned in Table 1, including iterated racing (López-Ibáñez et al., 2016; Vieira, 2021), as well as heuristics and automation of algorithm designs (Bezerra et al., 2016; Hoos and Stützle, 2018); see more comparisons and usages in (Lindauer et al., 2022; Feurer et al., 2015).

We divide the open-source packages into three categories:

Name	Type	Client Languages	Parallel Trials	Features*
OSS Vizier	Service	Any	Yes	Multi-Objective, Early Stopping, Transfer Learning, Conditional Search
SMAC	Framework	Python	Yes	Multi-Objective, Multi-fidelity, Early Stopping, Conditional Search, Parameter Constraints
Advisor	Service	Any	Yes	Early Stopping
OpenBox	Service	Any	Yes	Multi-Objective, Early Stopping, Transfer Learning, Parameter Constraints
HpBandSter	Framework	Python	Yes	Early Stopping, Conditional Search, Parameter Constraints
Ax + BoTorch	Framework	Python	Yes	Multi-Objective, Multi-fidelity, Early Stopping, Transfer Learning, Parameter and Outcome Constraints
HyperOpt	Library	Python	No	Conditional Search
Emukit	Library	Python	No	Multi-Objective, Multi-fidelity, Outcome Constraints

Table 1: Open Source Optimization Packages. *OSS Vizier supports the API only.

- **Services** host algorithms in a server. OSS Vizier, Advisor (Chen, 2017) and OpenBox (Li et al., 2021), which are modeled after Google Vizier (Golovin et al., 2017), belong to this category. Services are more flexible and scalable than frameworks, at the cost of engineering complexities.

- **Frameworks** execute the entire optimization, including both the suggestion algorithm and user evaluation code. Ax (Facebook, 2021) and HpBandSter (ML4AAD, 2018) belong to this category. While frameworks are convenient, they often require knowledge on the system being optimized, such as how to manage resources and perform proper initialization and shutdown.

- **Libraries** implement blackbox optimization algorithms. HyperOpt (Bergstra et al., 2013), Emukit (Paleyes et al., 2019), and BoTorch (Balandat et al., 2020) belong to this category. Libraries offer the most freedom but lack scalability features such as error recovery and distributed/asynchronous trial evaluations. Instead, libaries are often used as algorithm implementations for frameworks or services (e.g. BoTorch in Ax).

One major architectural difference between OSS Vizier and other services is that OSS Vizier's algorithms may run in a separate service and communicate via RPCs with the API server, which performs database operations. With a distributed backend setup, OSS Vizier can serve algorithms written in different languages, scale up to thousands of concurrent users, and continuously process user requests without interruptions during a server maintenance or update.

Furthermore, there are other minor differences between the services. While OSS Vizier and OpenBox support distinguishing workers via the workers' logical IDs (Section 5), Advisor does not. In addition, OSS Vizier's Python clients possess more sophisticated functionalities than Advisor's, while OpenBox lacks a client implementation and requires users to implement client code using framework-provided worker wrappers. OSS Vizier also emphasizes algorithm development, by providing a developer API called *Pythia* (Section 6) and utility libraries for state recovery. Other features of OSS Vizier include:

- OSS Vizier is one of the first open-source AutoML systems simultaneously compatible with a large-scale industry production service, Vertex Vizier, via our PyVizier library (Section 4.3).

- The backend of OSS Vizier is based on the standard Google Protocol Buffer library, one of the most widely used RPC formats, which allows extensive customizability. In particular, the client (i.e. blackbox function to be tuned) can be written in any language and is not restricted to machine learning models in Python.

- OSS Vizier is extensively integrated with numerous other Google packages, such as Deepmind XManager for experiment management (Section 7).

3 Infrastructure

We briefly conceptually define a *Study* as all relevant data pertaining to an entire optimization loop, a *Suggestion* as a suggested x, and a *Trial* containing both x and the objective $f(x)$. Note that in the code, we use `Trial` as a container to store both x and $f(x)$ and thus, a `Trial` without $f(x)$ is also considered a suggestion. We define these core primitives more programatically in Section 4.

3.1 Protocol Buffers

OSS Vizier's APIs are RPC interfaces that carry protocol buffers, or *protobufs/protos*[4], to allow simple and efficient inter-machine communication. The protos are language- and platform- independent objects for serializing structured data, which make building external software layers and wrappers onto the system straightforward. In particular, the user can provide their own:

- **Visualization Tools**: Since OSS Vizier securely stores all study data in its database, the data can then be loaded and visualized, with e.g. standard Python tools (Colab, Numpy, Scipy, Matplotlib) and other statistical packages such as R via RProtoBuf (Eddelbuettel et al., 2014). Front-end languages such as Angular/Javascript may also be used for visualizing studies.
- **Persistent Datastore**: The database in OSS Vizier can changed based on the user's needs. For instance, a SQL-based datastore with full query functionality may be used to store study data.
- **Clients**: Protobufs allow binaries written in Python, C++, and other languages to be tuned and/or used for evaluating the objective function. This allows OSS Vizier to easily tune existing systems.

We explain the interactions between these components in a distributed backend below.

3.2 Distributed Backend

In order to serve multiple users while remaining fault-tolerant, OSS Vizier runs in a distributed fashion, with a *server* performing the algorithmic proposal work, while users or *clients* communicate with the server via RPCs using the Client API, built upon gRPC [5]. A packet of RPC communication is formatted in terms of standard Google protobufs.

To start an optimization loop, a client will send a `CreateStudy` RPC request to the server, and the server will create a new Study in its datastore and return the ID to the client. The main tuning workflow in OSS Vizier will then involve the following repeated cycle of events:

1. The client sends a `SuggestTrials` RPC request to the server.
2. The server creates a `Operation` in its datastore, and starts a thread to launch a Pythia policy (i.e. blackbox optimization algorithm) to compute the next suggested `Trials`. The server returns an `Operation` protobuf to the client to denote the computation taking place.
3. The client will repeatedly poll the server via `GetOperation` RPCs to check the status of the `Operation` until the `Operation` is done.
4. When the Pythia policy produces its suggestions, the server will store these suggestions into the `Operation` and mark the `Operation` done, which will be collected by the client's `GetOperation` ping.
5. The client retrieves the suggestions $x_i, ..., x_{i+n}$ stored inside the `Operation`, and returns objective function measurements $f(x_i), ..., f(x_{i+n})$ to the server via calls to the `CompleteTrial` RPC.

Note that the server may be launched in the same local process as the client, in cases where distributed computing is not needed and functio evaluation is cheap (e.g. benchmarking algorithms

[4] https://github.com/protocolbuffers/protobuf [5] https://grpc.io/

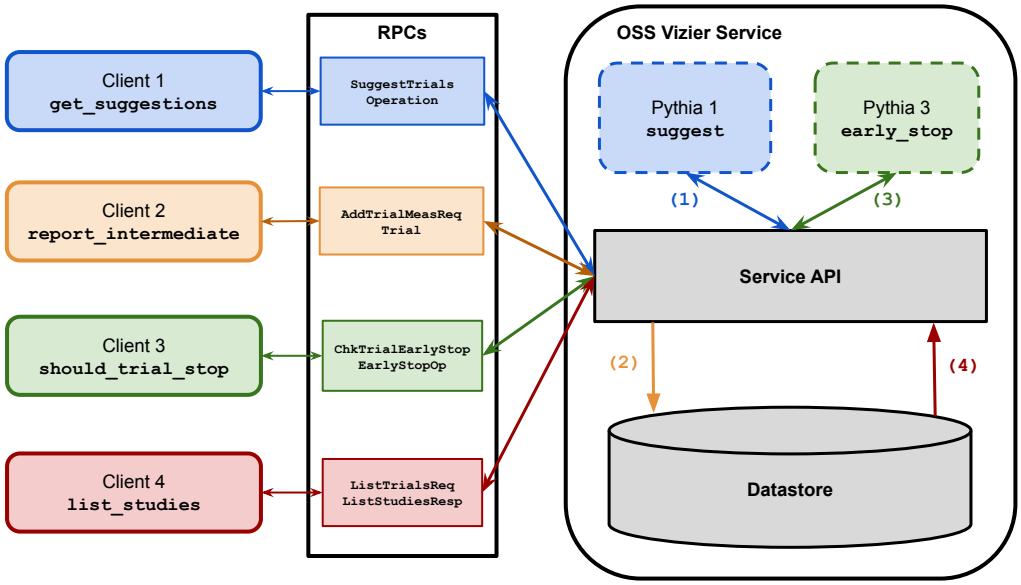

Figure 2: Pictorial representation of the distributed pipeline. The OSS Vizier server services multiple clients, each with their own types of requests. Such requests can involve running Pythia Policies, saving measurement data, or retrieving previous studies. Note that Pythia may run as a separate service from the API service.

on synthetic functions). However, if the user wishes to use the distributed setting, the following are core advantages of OSS Vizier's system:

Server-side Fault Tolerance. The `Operations` are stored in the database and contain sufficient information to restart the computation after a server crash, reboot, or update.

Automated/Early Stopping. A similar sequence of events takes place when the client sends a `CheckTrialEarlyStoppingStateRequest` RPC, in which the policy determines if a trial's evaluation should be stopped, and returns this signal as a boolean via the `EarlyStoppingOperation` RPC.

Batched/Parallel Evaluations. Note that *multiple clients may work on the same study, and the same* `Trial`. This is important for compute-heavy experiments (e.g. neural architecture search) which need to parallelize workload by using multiple machines, with each machine j evaluating the objective $f(x_j)$ after being given suggestion x_j from the server.

Client-side Fault Tolerance. When one of the parallel workers fails and then reboots, the service will assign the worker the same suggestion as before. The worker can choose to load a model from the checkpoint to warm-start the evaluation.

4 Core Primitives

In Figure 3, we provide a pictorial example representation of how OSS Vizier's primitives are structured; below we provide definitions.

4.1 Definitions

A `Study` is a single optimization run over a feasible space. Each study contains a name, its description, its state (e.g. `ACTIVE`, `INACTIVE`, or `COMPLETED`), a `StudySpec`, and a list of suggestions and evaluations (`Trials`).

A `StudySpec` contains the configuration details for the `Study`, namely the search space $\mathcal{X}$ (constructed by `ParameterSpecs`; see §4.2), the algorithm to be used, automated stopping type (see

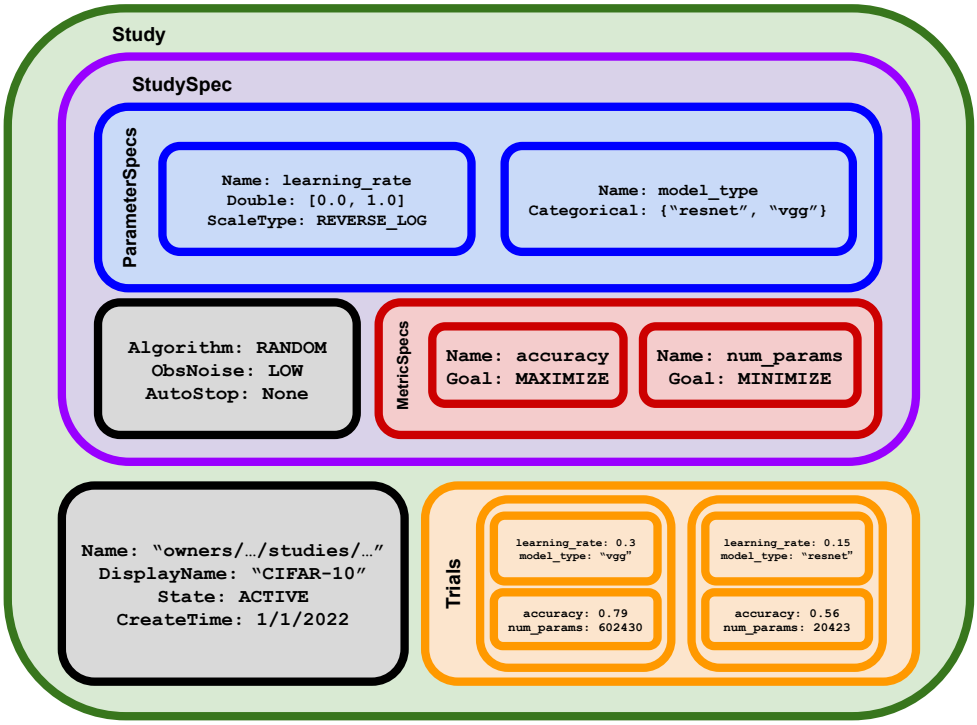

Figure 3: Example of a study that tunes a deep learning task, featuring relevant data types.

Appendix B.1), the type of `ObservationNoise` (see Appendix B.2), and at least one `MetricSpec`, containing information about the metric f to optimize, including the metric name and the goal (i.e. whether to minimize or maximize f). Multiple `MetricSpecs` will be used for cases involving multiobjective optimization, where the goal is to find Pareto frontiers over multiple objectives $f_1, ..., f_k$.

A `Trial` is a container for the input $x \in \mathcal{X}$, as well as potentially the scalar value $f(x)$ or multiobjective values $f_1(x), ..., f_k(x)$. Each `Trial` possesses a `State`, which indicates what stage of the optimization process the `Trial` is in, with the two primary states being `ACTIVE` (meaning that x has been suggested but not yet evaluated) and `COMPLETED` (meaning that evaluation is finished, and typically that the objectives $(f_1(x), ..., f_k(x))$ have been calculated).

Both the `StudySpec` and the `Trials` can contain `Metadata`. Metadata is not interpreted by OSS Vizier, but rather a convenient method for developers to store algorithm state, by users to store small amounts of arbitrary data, or as an extra communication medium between user code and algorithms.

4.2 Search Space

Search spaces can be built by combining the the following primitives, or `ParameterSpecs`:

- `Double`: Specifies a continuous range of possible values in the closed interval $[a, b]$ for some real values $a \leq b$.
- `Integer`: Specifies an integer range of possible values in $[a, b] \in \mathbb{Z}$ for some integers $a \leq b$.
- `Discrete`: Specifies a finite, ordered set of values from $\mathbb{R}$.
- `Categorical`: Specifies an unordered list of strings.

Furthermore, each of the numerical parameters {Double, Integer, Discrete} has a *scaling type*, which toggles whether the underlying algorithm is performing optimization in a transformed space. The scale type allows the user to conveniently inform the optimizer about the shape of the function, and can sometimes drastically accelerate the optimization. For instance, a user may use logarithmic scaling, which expresses the intent that a parameter ranging over $[0.001, 10]$ should roughly receive the same amount of attention in the subrange $[0.001, 0.01]$ as $[1, 10]$, which would otherwise not be the case when using uniform scaling.

Each parameter also can potentially contain a list of child parameters, each of which will be active only if the parent's value matches the correct value(s). This allows the notion of *conditional search*, which is helpful when dealing with search spaces involving incompatible parameters or parameters which only exist in specific scenarios. For example, this can be useful when competitively tuning several machine learning algorithms along with each algorithm's parameters. E.g. one could tune the following for the `model` parameter: `{"linear", "DNN", "random_forest"}`, each with its own set of parameters. Conditional parameters help keep the user's code organized, and also describe certain invariances to OSS Vizier, namely that when `model="DNN"`, $f(x)$ will be independent of the `"random_forest"` and `"linear"` model parameters.

These parameter primitives can be used flexibly to build highly complex search spaces, of which we provide examples in Appendix A.

4.3 PyVizier

All the above objects are implemented as protos to allow RPC exchanges through the service, as mentioned in Section 3. However, for ease-of-access, each object is also represented by an equivalent *PyVizier* class to provide a more Pythonic interface, validation, and convenient construction (further details and examples are found in Appendix D.3). Translations to and from protos are provided by the `to_proto()` and `from_proto()` methods in PyVizier classes. **PyVizier provides a common interface across all Vizier variants (i.e. Google Vizier, Vertex Vizier, and OSS Vizier)**[6]. The two intended primary use cases for PyVizier are:

- Tuning user binaries. For such cases, the core PyVizier primitive is the `VizierClient` class that allows communication with the service.
- Developing algorithms for researchers. In this case, the core PyVizier primitives are the Pythia `Policy` and `PolicySupporter` classes.

Both cases typically use the `StudyConfig` and `SearchSpace` classes to define the optimization, and the `Trial`, and `Measurement` classes to support the evaluation. We describe the two cases in detail below.

5 User API: Parallel Distributed Tuning with OSS Vizier Client

The OSS Vizier service must be set up first (see pseudocode in Appendix D.2), preferably on a multithreaded machine capable of processing multiple RPCs concurrently. Then, replicas of Code Block 1 can be launched in parallel, each with a unique command-line argument to be used as the client id in Line 11. The first replica to be launched creates a new `Study` from the `StudyConfig`, which defines the search space, relevant metrics to be evaluated, and the algorithm for providing suggestions. The other replicas then load the same study to be worked on. There are a few important aspects worth noting in this setting:

- The service does not make any assumptions about how `Trials` are evaluated. Users may complete `Trials` at any latency, and may do so with a custom client written in any language. Algorithms

[6]For compatibility reasons, protos have slightly different names than PyVizier equivalents; e.g. `StudySpec` protos are equivalent to `StudyConfig` PyVizier objects. We describe conversions further in Appendix D.3

```
from vizier import StudyConfig, VizierClient

config = StudyConfig() # Search space, metrics, and algorithm.
root = config.search_space.select_root() # "Root" params must exist in every trial.
root.add_float('learning_rate', min=1e-4, max=1e-2, scale='LOG')
root.add_int('num_layers', min=1, max=5)
config.metrics.add('accuracy', goal='MAXIMIZE', min=0.0, max=1.0)
config.algorithm = 'RANDOM_SEARCH'

client = VizierClient.load_or_create_study(
    'cifar10', config, client_id=sys.argv[1]) # Each client should use a unique id.

while suggestions := client.get_suggestions(count=1)
  # Evaluate the suggestion(s) and report the results to Vizier.
  for trial in suggestions:
    metrics = _evaluate_trial(trial.parameters)
    client.complete_trial(metrics, trial_id=trial.id)
```

Code Block 1: Pseudocode for tuning a blackbox function using the included Python client. To save space, we did not use longer official argument names from the actual code.

may however, set a time limit and reassign `Trials` to other clients to prevent stalling (e.g. due to a slow client).

- Each `Trial` is assigned a `client_id` and only suggested to clients created with the same `client_id`. This design makes it easy for users to recover from failures during `Trial` evaluations; if one of the tuning binaries is accidentally shut down, users can simply restart the binary with the same client id. The tuning binary creates a new client attached to the same study and OSS Vizier suggests the same `Trial`.

- Multiple binaries can share the same `client_id` and collaborate on evaluating the same `Trial`. This feature is useful in tuning a large distributed model with multiple workers and evaluators.

- The client may optionally turn on automated stopping for objectives that can provide intermediate measurements (e.g. learning curves in deep learning applications). Further details and an example code snippet can be found in Appendix B.1 and Appendix 3 respectively.

6 Developer API: Implementing a New Algorithm Using Pythia Policy

6.1 Overview

As we have explained in Section 3, OSS Vizier runs its algorithms in a binary called the *Pythia service* (which can be the same binary as the API service). When the client asks for suggestions or early stopping decisions, the API service creates operations and sends requests to the Pythia service. This section describes the default python implementation of the Pythia service included in the open-source package.

The Pythia service creates a `Policy` object that executes the algorithm and returns the response. `Policy` is designed to be a minimal and general-purposed interface built on top of PyVizier, to allow researchers to quickly incorporate their own blackbox optimization algorithms. `Policy` is usually given a `PolicySupporter`, which is a mini-client specialized in reading and filtering `Trials`. As shown in Code Block 2, a typical `Policy` loads `Trials` via `PolicySupporter` and processes the request at hand.

```python
from vizier.pythia import Policy, PolicySupporter, SuggestRequest, SuggestDecisions

class MyPolicy(Policy):
  def __init__(self, policy_supporter: PolicySupporter):
    self.policy_supporter = policy_supporter  # Used to obtain old trials.

  def suggest(self, request: SuggestRequest) -> SuggestDecisions:
    """Suggests trials to be evaluated."""
    Xs, y = _trials_to_np_arrays(self.policy_supporter.GetTrials(
        status='COMPLETED')) # Use COMPLETED trials only.
    model = _train_gp(Xs, y)
    return _optimize_ei(model, request.study_config.search_space)
```

Code Block 2: Pseudocode for implementing a Gaussian Process Bandit.

6.2 PolicySupporter

The `PolicySupporter` allows the `Policy` to actively decide what `Trials` from what `Studies` are needed to generate the next batch of `Suggestions`. Policies can meta-learn from potentially any `Study` in the database by calling the `GetStudyConfig` and `GetTrials` methods. Beyond that, the Policy can request only the `Trials` it needs; e.g. for algorithms that only need to look at newly evaluated `Trials`, this can reduce the database work by orders of magnitude relative to loading all the `Trials`.

6.3 State Saving via Metadata

The primary application of Google Vizier (Golovin et al., 2017) was optimizing a blackbox function that is expensive to evaluate. Over time, as Google Vizier became widely adopted, there was an increasing number of applications where users wished to evaluate cheap functions over a very large number of `Trials`. Popular methods for these applications include evolutionary methods and local search methods, such as NSGA-II (Deb et al., 2002), Firefly (Yang, 2010), and Harmony Search (Lee and Geem, 2005) to name a few (For a survey on meta-heuristics, see Beheshti and Shamsuddin (2013)).

A typical algorithm in this category iteratively updates its population pool and generates mutations to be suggested, both of which take constant time with respect to the number of previous trials, as opposed to e.g. cubic time when using Gaussian Processes in a Bayesian Optimization loop. Since the lifespan of a `Policy` object is equivalent to that of one suggestion or early stopping operation, the algorithm would need to fetch all `Trials` in the `Study` and reconstruct its state in O(number of previous trials) time. This leads to slow and difficult-to-maintain implementations.

`PolicySupporter` provides an easy-to-use API for developers to send algorithm states into the database as `Metadata`. `Metadata` is a key-value mapping with namespaces that help prevent key collisions. There are two tables for metadata in the database: one attached to the `StudySpec` and another to each `Trial`. A `Policy` can restore its last saved state from metadata, reflect the recently added `Trials`, and process the operation at hand. We provide example code for this functionality in Appendix D.4

7 Integrations

OSS Vizier is also compatible with multiple other interfaces developed at Google as well. These include:

[7] https://cloud.google.com/vertex-ai/docs/reference/rest/v1beta1/StudySpec.

- Vertex Vizier whose Protocol Buffer definitions are exactly the same[7] as OSS Vizier's. This consistency also allows a wide variety of other packages (discussed below) pre-integrated with Vertex Vizier to be used with minimal changes.
- Deepmind XManager experiments currently can be tuned by Vertex Vizier[8] through `VizierWorker`. This worker can also be directly connected to an OSS Vizier server to allow custom policies to manage experiments.
- OSS Vizier will also be the core backend for PyGlove (Peng et al., 2020)[9], which is a symbolic programming language for AutoML, in particular facilitating combinational and evolutionary optimization which are common in neural architecture search applications.

8 Conclusion, Limitations and Broader Impact Statement

Conclusion. We discussed the motivations and benefits behind providing OSS Vizier as a service in comparison to other blackbox optimization libraries, and described how our gRPC-based distributed back-end infrastructure may be deployed as a fault-tolerant yet flexible system that is capable of supporting multiple clients and diverse use cases. We further outlined our client-server API for tuning, our algorithm development Pythia API, and integrations with other Google libraries.

Limitations. Due to proprietary and legal concerns, we are unable to open-source the default algorithms used in Google Vizier and Cloud Vizier. Furthermore, this paper intentionally does not discuss algorithms or benchmarks, as the emphasis is on the systems aspect of AutoML. Algorithms may easily be added as policies to OSS Vizier's collection over time from contributors.

OSS Vizier also may not be suitable for all problems within the very broad scope of blackbox optimization. For instance, if evaluating $f(x)$ is very cheap and fast (e.g. miliseconds), then the OSS Vizier service itself may dominate the overall cost and speed. Furthermore, for problems requiring very large numbers of parameters (e.g. 100K+) and evaluations (e.g. 1M+), such as training a large neural network with gradientless methods (Mania et al., 2018; Such et al., 2017), OSS Vizier can also be inappropriate, as such cases can overload the datastore memory with redundant trials which do not need to be kept track of.

Broader Impact. While there are a rich collection of sophisticated and effective AutoML algorithms published every year, broad adoption to practical use cases still remains low, as only 7% of the ICLR 2020 and NeurIPS 2019 papers used a tuning method other than random or grid search (Bouthillier and Varoquaux, 2020). In comparison, Google Vizier is widely used among multiple researchers at Google, including for conference submissions. We hope that the release of OSS Vizier and its similar benefits may significantly improve the reach of AutoML techniques to users.

In terms of potential negative impacts, optimization as a service encourages central storage of data with the attendant risks and benefits. For example, currently through the Client API, a user may request all studies associated with another users, which may cause security and privacy concerns. This may be fixed by limiting user access to only their own studies in the service logic. Furthermore, the host of the service currently has full access to all client data, which is another potential privacy concern. However, from our experience with Google Vizier, the most impactful applications for clients typically occur when parameters and measurements correspond to aggregate data (e.g. the learning rate of a ML algorithm, or e.g. the number of threads in a server) rather than data that describes individuals. Furthermore, data received by OSS Vizier can be obscured to a degree to reduce unwanted exposure to the host. Most notably, names (e.g. study name, parameter and metric names) can be encrypted, and (within limits) differential privacy (Dwork, 2008) approaches, especially for databases (Johnson et al., 2018), can be applied to the parameters values and measurements.

[8] https://github.com/deepmind/xmanager/tree/main/xmanager/vizier. [9] PyGlove will be open-sourced soon.

9 Reproducibility Checklist

1. For all authors...

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We discussed the motivations for why OSS Vizier is designed as a service, and outlined in detail its distributed infrastructure. We further demonstrated (with pseudocode) the two main usages of OSS Vizier, which are to tune users' objects via client-side API, and develop algorithms via Pythia.

   (b) Did you describe the limitations of your work? [Yes] See Section 8.

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 8.

   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] Our paper follows all of the ethics review guidelines.

2. If you are including theoretical results...

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] This is a systems paper.

   (b) Did you include complete proofs of all theoretical results? [N/A] This is a systems paper.

3. If you ran experiments...

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit version), an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] We have provided a README, installation instructions with a `requirements.txt`, numerous integration and unit tests along with PyTypes which demonstrate each code snippet's function.

   (b) Did you include the raw results of running the given instructions on the given code and data? [Yes] Our unit-tests demonstrate the expected results of running all components of our code.

   (c) Did you include scripts and commands that can be used to generate the figures and tables in your paper based on the raw results of the code, data, and instructions given? [N/A] This is a systems paper.

   (d) Did you ensure sufficient code quality such that your code can be safely executed and the code is properly documented? [Yes] Our code follows all standard industry-wide coding practices at Google, which include extensive unit tests with continuous integration, PyType and PyLint enforcement for code cleanliness, and peer review during code submission.

   (e) Did you specify all the training details (e.g., data splits, pre-processing, search spaces, fixed hyperparameter settings, and how they were chosen)? [N/A] This is a systems paper.

   (f) Did you ensure that you compared different methods (including your own) exactly on the same benchmarks, including the same datasets, search space, code for training and hyperparameters for that code? [N/A] This is a systems paper.

   (g) Did you run ablation studies to assess the impact of different components of your approach? [N/A] This is a systems paper.

   (h) Did you use the same evaluation protocol for the methods being compared? [N/A] This is a systems paper.

(i) Did you compare performance over time? [N/A] This is a systems paper.

(j) Did you perform multiple runs of your experiments and report random seeds? [N/A] This is a systems paper.

(k) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A] This is a systems paper.

(l) Did you use tabular or surrogate benchmarks for in-depth evaluations? [N/A] This is a systems paper.

(m) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [N/A] This is a systems paper.

(n) Did you report how you tuned hyperparameters, and what time and resources this required (if they were not automatically tuned by your AutoML method, e.g. in a NAS approach; and also hyperparameters of your own method)? [N/A] This is a systems paper.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [Yes] Our work wraps around other Google libraries such as the Cloud Vizier SDK and Deepmind XManager, which we provided url links for.

(b) Did you mention the license of the assets? [Yes] Both the Cloud Vizier SDK and Deepmind XManager use the Apache 2.0 License.

(c) Did you include any new assets either in the supplemental material or as a URL? [N/A] No new assets were used.

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] No human data was used.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] This is a systems paper without data use.

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] Not applicable.

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] Not applicable.

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] Not applicable.

References

Agarwal, R., Frosst, N., Zhang, X., Caruana, R., and Hinton, G. E. (2020). Neural additive models: Interpretable machine learning with neural nets. *CoRR*, abs/2004.13912v2.

Agarwal, R., Machado, M. C., Castro, P. S., and Bellemare, M. G. (2021). Contrastive behavioral similarity embeddings for generalization in reinforcement learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Aygün, E., Ahmed, Z., Anand, A., Firoiu, V., Glorot, X., Orseau, L., Precup, D., and Mourad, S. (2020). Learning to prove from synthetic theorems. *CoRR*, abs/2006.11259v1.

Balandat, M., Karrer, B., Jiang, D. R., Daulton, S., Letham, B., Wilson, A. G., and Bakshy, E. (2020). Botorch: A framework for efficient monte-carlo bayesian optimization. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Beheshti, Z. and Shamsuddin, S. M. H. (2013). A review of population-based meta-heuristic algorithms. *Int. J. Adv. Soft Comput. Appl*, 5(1):1–35.

Bergstra, J., Yamins, D., and Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR.

Bezerra, L. C. T., López-Ibáñez, M., and Stützle, T. (2016). Automatic component-wise design of multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 20(3):403–417.

Bileschi, M. L., Belanger, D., Bryant, D., Sanderson, T., Carter, B., Sculley, D., DePristo, M. A., and Colwell, L. J. (2022). Using deep learning to annotate the protein universe. *Nature Biotechnology*.

Bouthillier, X. and Varoquaux, G. (2020). Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020. Research report, Inria Saclay Ile de France.

Cassirer, A., Barth-Maron, G., Brevdo, E., Ramos, S., Boyd, T., Sottiaux, T., and Kroiss, M. (2021). Reverb: A framework for experience replay. *CoRR*, abs/2102.04736v1.

Chen, D. (2017). Advisor. https://github.com/tobegit3hub/advisor.

Chen, L., Collins, M. D., Zhu, Y., Papandreou, G., Zoph, B., Schroff, F., Adam, H., and Shlens, J. (2018). Searching for efficient multi-scale architectures for dense image prediction. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 8713–8724.

Chiang, H.-T. L., Faust, A., Fiser, M., and Francis, A. (2019). Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*, 4(2):2007–2014.

Co-Reyes, J. D., Miao, Y., Peng, D., Real, E., Le, Q. V., Levine, S., Lee, H., and Faust, A. (2021). Evolving reinforcement learning algorithms. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197.

Dong, X. and Yang, Y. (2020). Nas-bench-201: Extending the scope of reproducible neural architecture search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Dwork, C. (2008). Differential privacy: A survey of results. In Agrawal, M., Du, D., Duan, Z., and Li, A., editors, *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, volume 4978 of *Lecture Notes in Computer Science*, pages 1–19. Springer.

Eddelbuettel, D., Stokely, M., and Ooms, J. (2014). Rprotobuf: Efficient cross-language data serialization in R. *CoRR*, abs/1401.7372v1.

Facebook (2021). Adaptive experimentation platform. https://ax.dev/.

Faust, A., Francis, A. G., and Mehta, D. (2019). Evolving rewards to automate reinforcement learning. *6th ICML Workshop on Automated Machine Learning (2019)*.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J. T., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2962–2970.

Francis, A., Faust, A., Chiang, H.-T. L., Hsu, J., Kew, J. C., Fiser, M., and Lee, T.-W. E. (2020). Long-range indoor navigation with prm-rl. *IEEE Transactions on Robotics*, 36(4):1115–1134.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. (2017). Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 1487–1495. ACM.

He, X., Zhao, K., and Chu, X. (2021). Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622.

Hoos, H. H. and Stützle, T. (2018). Stochastic local search. In Gonzalez, T. F., editor, *Handbook of Approximation Algorithms and Metaheuristics, Second Edition, Volume 1: Methologies and Traditional Applications*, pages 297–307. Chapman and Hall/CRC.

Hron, J., Bahri, Y., Novak, R., Pennington, J., and Sohl-Dickstein, J. (2020a). Exact posterior distributions of wide bayesian neural networks. *ICML 2020 Workshop on Uncertainty and Robustness in Deep Learning*.

Hron, J., Bahri, Y., Sohl-Dickstein, J., and Novak, R. (2020b). Infinite attention: NNGP and NTK for deep attention networks. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 4376–4386. PMLR.

Johnson, N. M., Near, J. P., and Song, D. (2018). Towards practical differential privacy for SQL queries. *Proc. VLDB Endow.*, 11(5):526–539.

Kochanski, G., Golovin, D., Karro, J., Solnik, B., Moitra, S., and Sculley, D. (2017). Bayesian optimization for a better dessert. *2017 Neural Information Processing Systems Workshop on Bayesian Optimization (BayesOpt 2017)*.

Krause, J., Gulshan, V., Rahimy, E., Karth, P., Widner, K., Corrado, G. S., Peng, L., and Webster, D. R. (2017). Grader variability and the importance of reference standards for evaluating machine learning models for diabetic retinopathy. *CoRR*, abs/1710.01711v3.

Lee, J., Schoenholz, S. S., Pennington, J., Adlam, B., Xiao, L., Novak, R., and Sohl-Dickstein, J. (2020). Finite versus infinite neural networks: an empirical study. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Lee, K. S. and Geem, Z. W. (2005). A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. *Computer methods in applied mechanics and engineering*, 194(36-38):3902–3933.

Li, Y., Shen, Y., Zhang, W., Chen, Y., Jiang, H., Liu, M., Jiang, J., Gao, J., Wu, W., Yang, Z., Zhang, C., and Cui, B. (2021). Openbox: A generalized black-box optimization service. In Zhu, F., Ooi, B. C., and Miao, C., editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 3209–3219. ACM.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., and Hutter, F. (2022). Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

Mania, H., Guy, A., and Recht, B. (2018). Simple random search of static linear policies is competitive for reinforcement learning. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 1805–1814.

ML4AAD (2018). Hpbandster. https://github.com/automl/HpBandSter/.

Nguyen, T., Novak, R., Xiao, L., and Lee, J. (2021). Dataset distillation with infinitely wide convolutional networks. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021 (NeurIPS 2021)*.

Paleyes, A., Pullin, M., Mahsereci, M., Lawrence, N., and González, J. (2019). Emulation of physical processes with emukit. In *Second Workshop on Machine Learning and the Physical Sciences, NeurIPS*.

Parker-Holder, J., Rajan, R., Song, X., Biedenkapp, A., Miao, Y., Eimer, T., Zhang, B., Nguyen, V., Calandra, R., Faust, A., Hutter, F., and Lindauer, M. (2022). Automated reinforcement learning (autorl): A survey and open problems. *CoRR*, abs/2201.03916v1.

Peng, D., Dong, X., Real, E., Tan, M., Lu, Y., Bender, G., Liu, H., Kraft, A., Liang, C., and Le, Q. (2020). Pyglove: Symbolic programming for automated machine learning. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4780–4789. AAAI Press.

Shields, B. J., Stevens, J., Li, J., Parasram, M., Damani, F., Alvarado, J. I. M., Janey, J. M., Adams, R. P., and Doyle, A. G. (2021). Bayesian reaction optimization as a tool for chemical synthesis. *Nature*, 590:89–96.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567v3.

Tiwary, S., Levy, R., Gutenbrunner, P., Soto, F. S., Palaniappan, K. K., Deming, L., Berndl, M., Brant, A., Cimermancic, P., and Cox, J. (2019). High-quality ms/ms spectrum prediction for data-dependent and data-independent acquisition data analysis. *Nature Methods*, 16:519–525.

Verbraeken, J., Wolting, M., Katzy, J., Kloppenburg, J., Verbelen, T., and Rellermeyer, J. S. (2020). A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2):30:1–30:33.

Vieira, C. E. M. (2021). Assessing irace for automated machine and deep learning in computer vision. In *PhD Thesis*.

Wang, W., Tian, Y., Ngiam, J., Yang, Y., Caswell, I., and Parekh, Z. (2020). Learning a multi-domain curriculum for neural machine translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7711–7723, Online. Association for Computational Linguistics.

Yang, X.-S. (2010). Firefly algorithm, stochastic test functions and design optimisation. *International journal of bio-inspired computation*, 2(2):78–84.

Yazdanbakhsh, A., Angermüller, C., Akin, B., Zhou, Y., Jones, A., Hashemi, M., Swersky, K., Chatterjee, S., Narayanaswami, R., and Laudon, J. (2020). Apollo: Transferable architecture exploration. *ML for Systems at the 34th Conference on Neural Information Processing Systems (NeurIPS 2020)*.

Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., and Hutter, F. (2019). Nas-bench-101: Towards reproducible neural architecture search. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114. PMLR.

Zhang, D., Huda, S., Songhori, E. M., Prabhu, K., Le, Q. V., Goldie, A., and Mirhoseini, A. (2022). A full-stack search technique for domain optimized deep learning accelerators. In Falsafi, B., Ferdman, M., Lu, S., and Wenisch, T. F., editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 27–42. ACM.

Zhang, Y., Apley, D. W., and Chen, W. (2020). Bayesian optimization for materials design with mixed quantitative and qualitative variables. *Sci Rep*, 10:4924–.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8697–8710. Computer Vision Foundation / IEEE Computer Society.

Appendix

A Search Space Flexibility

In this section, we describe the ways in which more complex search spaces may be created in OSS Vizier, showcasing its flexibility and applicability to a wide variety of problems.

A.1 Combinatorial Optimization

One of the most common uses for blackbox optimization in research involves combinatorial optimization. In this setting, $\mathcal{X}$ is usually defined via common manipulations over the set $[n] = \{0, 1, ..., n-1\}$, such as permutations or subset selections. Below, we provide example methods to deal with such cases, in the order of most practical to least practical. We note that many of these methods are more suited for evolutionary algorithms which only need to utilize mutations and cross-overs between trials, rather than regression-based methods (e.g. Bayesian Optimization).

A.1.1 Reparameterization. Reparameterization of the search space $\mathcal{X}$ via conceptual means should be considered first, as it is one of the most practical and easiest ways to reduce the complexity of representing $\mathcal{X}$ in OSS Vizier. Mathematically speaking, the high level idea is to construct a more practical search space $\mathcal{Z}$ which can easily be represented in OSS Vizier, and then create a surjective mapping $\Phi : Z \rightarrow X$.

For basic combinatorial objects such as permutations, if we consider the standard permutation space $\mathcal{X} = \{x : x \in [n]^n, x_i \neq x_j \; \forall i \neq j\}$, then we may define $\mathcal{Z} = [n] \times [n-1] \times ... \times [2] \times [1]$ and allow Φ to be the decoding operator for the Lehmer code[10]. If $\mathcal{X} = \{x : x \subseteq [n], |x| = k\}$ involves subset selection, then we may define $\mathcal{Z} = [n] \times [n-1] \times ... \times [n-k+1]$ and apply a similar mapping.

Another common case involves searching over the space of graphs. In such scenarios, there are a multitude of methods to parameterizing the graph, including adjacency matrices via $[n] \times [n]$. An illustrative example can be seen across neural architecture search (NAS) benchmarks. Even though such search spaces correspond to graph objects, ironically, many NAS benchmarks, termed "NASBENCH"s, actually do not use nested or conditional search spaces. For instance, NASBENCH-101 (Ying et al., 2019) uses only a flat adjacency matrix and flat operation list. NASBENCH-201 (Dong and Yang, 2020) is even simpler, as it takes the graph dual of the node-op representation, allowing the search space to be a full feasible set represented by only 5 categorical parameters.

A.1.2 Infeasibility. In some scenarios, we may not be able to find a mapping Φ as in the reparameterization case above, but instead may lift the search space $\mathcal{X}$ into a larger search space $\mathcal{Z}$, where $\mathcal{X} \subset \mathcal{Z}$, and thus perform search on $\mathcal{Z}$ instead. For trials in $\mathcal{Z} - \mathcal{X} = \{z : z \in \mathcal{Z}, z \notin X\}$, OSS Vizier supports reporting these trials as infeasible. As a basic example, if $\mathcal{X} = \{x \in \mathbb{R}^2 : ||x|| \leq 1\}$ defines a disk, then $\mathcal{Z} = [-1, 1]^2$. Another example can be seen with the same NASBENCH-101 (Ying et al., 2019) benchmark described earlier, where some pairs of adjacency matrices and operation lists do not correspond to an actual valid graph, and are thus infeasible.

The main limitation is if $|\mathcal{X}| \ll |\mathcal{Z}|$, the vast bulk of trials may be infeasible, and if so, the search will converge slowly. Furthermore, for the disk case, this can lead to problems during optimization, as it creates a sharp border $\mathcal{X} \cap \mathcal{Z}$ and a flat infeasible region $\mathcal{Z} - \mathcal{X}$. This leads to lack of information about which infeasible points are better/worse than others, and can also make it difficult to find a small feasible region. Modelling techniques such Gaussian Processes also inherently assume the objective function is continuous everywhere, which is incompatible with the discontinuity from the border $\mathcal{X} \cap \mathcal{Z}$.

[10] https://en.wikipedia.org/wiki/Lehmer_code

A.1.3 Serialization. If all else fails, we may avoid the use of the *ParameterSpec* API and simply serialize $x \in \mathcal{X}$ into a string format, which can then be inserted into a Trial's *metadata* field. In cooperation with a custom Pythia policy, this can be very effective.

B Additional OSS Vizier Settings

B.1 Automated Stopping

Automated/early stopping is used commonly when trials can be stopped early to save resources, and is determined by the trial's intermediate measurements. Currently there are two modes to automated stopping which the client can specify in their `StudyConfig`:

- Decay Curve Automated Stopping, in which a Gaussian Process Regressor is built to predict the final objective value of a Trial based on the already completed Trials and the intermediate measurements of the current Trial. Early stopping is requested for the current Trial if there is very low probability to exceed the optimal value found so far.
- Median Automated Stopping, in which a pending trial is stopped if the Trial's best objective value is strictly below the median 'performance' of all completed Trials reported up to the Trial's last measurement. Currently, 'performance' refers to the running average of the objective values reported by the Trial in each measurement.

B.2 Observation Noise

We have found it useful to let the user give Vizer a hint about the amount of noise in their evaluations via the `StudyConfig`. Because the noise/irreproducibility of evaluations is often not well known in advance by users, we give users a broad choice that the noise is either `Low` or `High`:

- `Low`: This implies that the objective function is (nearly) perfectly reproducible, and an algorithm should never repeat the same Trial parameters.
- `High`: This assumes there is enough noise in the evaluations that it is worthwhile for OSS Vizier sometimes to re-evaluate with the same (or nearly) parameter values.

This hint is passed to the Pythia policy, and the policy is free to also use this hint to e.g. adjust priors on the hyperparameters of a Gaussian Process regressor.

C Google Vizier Users and Citations

Besides Google Vizier's extensive internal production usage, below comprises a selected list of publicly available research works[11] which have used Google Vizier, demonstrating its rich research user-base which may directly translate to OSS Vizier's future user-base as well.

Neural Architecture Search. Google Vizier has acted as a core backend for many of the neural architecture search (NAS) efforts at Google, beginning with Google Vizier having been used to hyperparameter tune the RNN controller in the original NAS paper (Zoph and Le, 2017). Over the course of NAS research, Google Vizier has also been used to reliably handle the training of thousands of models (Zoph et al., 2018; Chen et al., 2018), as well as comparisons against different NAS optimization algorithms in NASBENCH-101 (Ying et al., 2019). Furthermore, it serves as the primary distributed backend for PyGlove (Peng et al., 2020), a core evolutionary algorithm API for NAS research across Google.

Hardware and Systems. Google Vizier's tuning led to crucial gains for hardware benchmarking, such as improving JAX's MLPerf scores over TPUs [12]. Google Vizier's multiobjective optimization capabilities were a key component in producing better computer architecture designs in APOLLO (Yazdanbakhsh et al., 2020) [13]. Furthermore, Google Vizier was a key component to *Full-stack Accelerator Search Technique* (FAST) (Zhang et al., 2022), an automated framework for jointly optimizing hardware datapath, software schedule, and compiler passes.

Reinforcement Learning. "AutoRL" (Parker-Holder et al., 2022) has recently seen a great deal of promise in automating reinforcement learning systems. Google Vizier was extensively used as the core component in tuning hyperparameters and rewards in navigation (Faust et al., 2019; Francis et al., 2020; Chiang et al., 2019). Google Vizier's backend was also used to host the Regularized Evolution optimizer (Real et al., 2019), used for evolving RL algorithms (Co-Reyes et al., 2021), where the search space involved combinatorial directed acyclic graphs (DAGs). On the infrastructure side, Google Vizier was used to improve the performance of Reverb (Cassirer et al., 2021), one of the core replay buffer APIs used for most RL projects at Google. (Agarwal et al., 2021)

Biology/Chemistry/Healthcare. Google Vizier's algorithms were used for comparison on several papers related to protein optimization (Bileschi et al., 2022), and was also used to tune RNNs for peptide identification in (Tiwary et al., 2019). For healthcare, Google Vizier was used to tune models for classifying diseases such as diabetic retinopathy (Krause et al., 2017)

General Deep Learning. For fundamental research, Google Vizier was used to tune Neural Additive Models (Agarwal et al., 2020), and has also been the backbone of core research into infinite-width deep neural networks, having tuned (Nguyen et al., 2021; Lee et al., 2020; Hron et al., 2020b,a). For NLP-based tasks, Google Vizier regularly tunes language model training, and has also been used to search feature weights in (Wang et al., 2020), as well improve performance for work on theorem proving (Aygün et al., 2020). Computer vision models such as ones used for the Pixel-3[14] have been tuned by Google Vizier.

Miscallaneous:. As an example of tuning for human-based judgement on objectives unrelated to technology, Google Vizier was used to tune the recipe for cookie-baking (Kochanski et al., 2017).

[11] Full list of Google Vizier's citations: https://scholar.google.com/scholar?oi=bibs&hl=en&cites=14342343058535677299.
[12] Link too long; hyperlink can be found here.
[13] https://ai.googleblog.com/2021/02/machine-learning-for-computer.html
[14] https://ai.googleblog.com/2018/12/top-shot-on-pixel-3.html

D Extended Code Samples

D.1 Automated stopping

Code Block 3 demonstrates the use of automated stopping, when training a standard machine learning model.

```
from vizier import StudyConfig, VizierClient

config = StudyConfig()
... # configure search space and metrics
client = VizierClient.load_or_create_study(
    'cifar10', config, client_id=sys.argv[1]) # Each client should use a unique id.

while suggestions := client.get_suggestions(count=1)
  # Evaluate the suggestion(s) and report the results to OSS Vizier.
  for trial in suggestions:
    for epoch in range(EPOCHS):
      if client.should_trial_stop(trial.id):
         break
       metrics = model.train_and_evaluate(trial.parameters['learning_rate'])
       client.report_metrics(epoch, metrics)
    metrics = model.evaluate()
    client.complete_trial(metrics, trial_id=trial.id)
```

Code Block 3: *Pseudocode* for tuning a model using the included Python client, with early stopping enabled.

D.2 Service Setup

Code Block 4 displays the simple method in which to setup the service on a multithreaded server.

```
from vizier.service import vizier_server
from vizier.service import vizier_service_pb2_grpc

hostname = 'localhost' # Example; usually user-specified
port = 6006 # Example; usually user-specified
address = f'{hostname}:{port}'
servicer = vizier_server.VizierService()

server = grpc.server(futures.ThreadPoolExecutor(max_workers=100))
vizier_service_pb2_grpc.add_VizierServiceServicer_to_server(servicer, server)
server.add_secure_port(address, grpc.local_server_credentials())
server.start()
```

Code Block 4: Pseudocode for setting up the service on a server.

D.3 Proto vs Python API

We provide an example of equivalent methods between PyVizier and corresponding Protocol Buffer objects. Note that clients and algorithm developers should not normally need to modify protos. Such cases are more common if one wishes to add extra layers on top of the service, as mentioned in Subsection 3.1.

```
from vizier.service import study_pb2
from google.protobuf import struct_pb2

param_1 = study_pb2.Trial.Parameter(parameter_id='learning_rate', value=struct_pb2.
    Value(number_value=0.4))
param_2 = study_pb2.Trial.Parameter(parameter_id='model_type', value=struct_pb2.
    Value(string_value='vgg'))
metric_1 = study_pb2.Measurement.Metric(metric_id='accuracy',value=0.4)
metric_2 = study_pb2.Measurement.Metric(metric_id='num_params',value=20423)
final_measurement = study_pb2.Trial.Measurement(metrics=[metric_1,metric_2])
trial = study_pb2.Trial(parameters=[param_1,param_2], final_measurement=
    final_measurement)
```

Code Block 5: Original Protocol Buffer method of creating a Trial.

```
from vizier.pyvizier import ParameterDict, ParameterValue, Measurement, Metric,
    Trial

params=ParameterDict()
params['learning_rate'] = ParameterValue(0.4)
params['model_type'] = ParameterValue('vgg')
final_measurement = Measurement()
final_measurement.metrics['accuracy'] = Metric(0.7)
final_measurement.metrics['num_params'] = Metric(20423)
trial = pv.Trial(parameters=params,final_measurement=final_measurement)
```

Code Block 6: Equivalent method of writing the PyVizier version of the Trial from Code Block 5. Note the significantly more "Pythonic" way of writing code, with a significant reduction in code complexity.

We also provide in Table 2, changes between OSS Vizier's Protocol Buffer names and their corresponding PyVizier names, as well as converter objects.

Protocol Buffer Name	PyVizier Name	Converter
`Study`	`Study`	N/A
`StudySpec`	`SearchSpace` + `StudyConfig`	`SearchSpace` (self) + `StudyConfig` (self)
`ParameterSpec`	`ParameterConfig`	`ParameterConfigConverter`
`Trial`	`Trial`	`TrialConverter`
`Parameter`	`ParameterValue`	`ParameterValueConvereter`
`MetricSpec`	`MetricInformation`	`MetricInformation` (self)
`Measurement`	`Measurement`	`MeasurementConverter`

Table 2: Corresponding names and conversion objects between Protocol Buffer and PyVizier objects. (self) denotes that the PyVizier object has its own immediate `to_proto()` and `from_proto()` functions.

D.4 Implementing an Evolutionary Algorithm

OSS Vizier possesses an abstraction `SerializableDesigner` defined purely in terms of PyVizier without any Pythia dependencies. This interface wraps around most commonly used algorithms which sequentially update their internal states as new observations arrive. The interface is easy to understand and can be wrapped into a Pythia policy using the `SerializableDesignerPolicy` class which handles state management. See Code Block 7 for an example.

```
from vizier import pyvizier as vz

class RegEvo(SerializableDesigner):

  # override
  def suggest(self, count: Optional[int]) -> Sequence[vz.TrialSuggestion]
    """Generate `count` number of mutations and return them."""

  # override
  def update(self, delta: CompletedTrials):
    """Apply selection step and update the population pool."""

  # override
  def dump(self) -> vz.Metadata:
    """Dumps the population pool."""
    md = vz.Metadata()
    md['population'] = json.dumps(...)
    return md

  # override
  def recover(cls: Type['_S'], metadata: vz.Metadata) -> '_S':
    """Restores the population pool."""
    if 'population' not in md:
      raise HarmlessDecodeError('Cannot find key: "population"')
    ... = json.loads(md['population'])

policy = SerializableDesignerPolicy(
    policy_supporter,
    designer_factory=RegEvo.__init__,
    designer_cls=RegEvo)
```

Code Block 7: Example Pseudocode of implementing an evolutionary algorithm as a Pythia policy using `SerializableDesigner` interface.