Open Source Vizier

The OSS Vizier Authors

Jun 01, 2023

DOCUMENTATION

Open Source (OSS) Vizier is a Python-based interface for blackbox optimization and research, based on Google's original internal Vizier, one of the first hyperparameter tuning services designed to work at scale.

Fig. 1: **OSS Vizier's distributed client-server system. Animation by Tom Small.**

CHAPTER

ONE

INSTALLATION

See https://github.com/google/vizier#installation for instructions on installing OSS Vizier.

CHAPTER

TWO

SUPPORT

If you are having issues, please let us know by filing an issue on our issue tracker.

CHAPTER

THREE

LICENSE

OSS Vizier is licensed under the Apache 2.0 License.

3.1 Guides

3.1.1 For Users

Vizier Basics

Below, we provide examples of how to:

- Define a problem statement and study configuration.
- Start a client.
- (Optionally) Connect the client to a server.
- Perform a typical tuning loop.
- Use other client APIs.

Installation and reference imports

```
!pip install google-vizier[jax]
```

```
from vizier import service
from vizier.service import clients
from vizier.service import pyvizier as vz
from vizier.service import servers
```

Setting up the problem statement

Here we setup the problem statement, which contains information about the search space and the metrics to optimize.

```
problem = vz.ProblemStatement()
problem.search_space.root.add_float_param('x', 0.0, 1.0)
problem.search_space.root.add_float_param('y', 0.0, 1.0)
problem.metric_information.append(
    vz.MetricInformation(
        name='maximize_metric', goal=vz.ObjectiveMetricGoal.MAXIMIZE))


def evaluate(x: float, y: float) -> float:
  return x**2 - y**2
```

Setting up the study configuration

The study configuration contains additional information, such as the algorithm to use and level of noise that we think the objective will have.

```
study_config = vz.StudyConfig.from_problem(problem)
study_config.algorithm = 'GAUSSIAN_PROCESS_BANDIT'
```

Setting up the client

Starts a `study_client`, which can be either in **local mode (default)** or **distributed mode.**

Local Mode: The client has no `endpoint` set, and will implicitly create a local Vizier Service which will be shared across other clients in the same Python process. Studies will then be stored locally in a SQL database file located at `service.VIZIER_DB_PATH`.

```
study_client = clients.Study.from_study_config(study_config, owner='owner', study_id=
↪'example_study_id')
print('Local SQL database file located at: ', service.VIZIER_DB_PATH)
```

Distributed mode: The service may be explicitly created, wrapped as a server in a separate process to accept requests from all other client processses. Details such as the `database_url`, `port`, `policy_factory`, etc. can be configured in the server's initializer.

All client processes (on a single machine or over multiple machines) will connect to this server via a globally specified `endpoint`.

```
server = servers.DefaultVizierServer()  # Ideally created on a separate process such as␣
↪a server machine.
clients.environment_variables.server_endpoint = server.endpoint  # Server address.
study_client = clients.Study.from_study_config(study_config, owner='owner', study_id =
↪'example_study_id')  # Now connects to the explicitly created server.
```

Client Parallelization

Regardless of whether the setup is local or distributed, we may simultaneously create multiple clients to work on the same study, useful for parallelizing evaluation workload.

```
another_study_client = clients.Study.from_resource_name(study_client.resource_name)
```

Obtaining suggestions

Start requesting suggestions from the server, for evaluating objectives. Suggestions can be made sequentially (`count=1`) or in batches (`count>1`).

```
for i in range(10):
  suggestions = study_client.suggest(count=1)
  for suggestion in suggestions:
    x = suggestion.parameters['x']
    y = suggestion.parameters['y']
    objective = evaluate(x, y)
    print(f'Iteration {i}, suggestion ({x},{y}) led to objective value {objective}.')
    final_measurement = vz.Measurement({'maximize_metric': objective})
    suggestion.complete(final_measurement)
```

Find optimal trial

Find the best objective so far, with corresponding suggestion value. For multiobjective cases, there may be multiple outputs of `optimal_trials()`, all corresponding to a Pareto-optimal curve.

```
for optimal_trial in study_client.optimal_trials():
  optimal_trial = optimal_trial.materialize()
  print("Optimal Trial Suggestion and Objective:", optimal_trial.parameters,
        optimal_trial.final_measurement)
```

Other client commands

The `study_client` can also send other requests, such as the following:

```
study_client.get_trial(1)  # Get the first trial.
study_client.trials()  # Get all trials so far.

# Obtain only the completed trials.
trial_filter = vz.TrialFilter(status=[vz.TrialStatus.COMPLETED])
study_client.trials(trial_filter=trial_filter)
```

Search Spaces

Below, we provide examples of how to:

- Setup a flat search space consisting of all four parameter types and additional auxiliary parameter types.
- Setup a conditional search space correctly.
- Reparameterize search spaces, which is useful for combinatorial search spaces.
- Use infeasibility to define shaped search spaces.

Installation and reference imports

```
!pip install google-vizier
```

```
import math
from typing import List

from vizier import pyvizier as vz
```

Flat search spaces

Below are the core primitive parameter types and their specifications:

- `DOUBLE`: Continuous range of possible values in the closed interval $[a, b]$ for some real numbers $a \leq b$.
- `INTEGER`: Integer range of possible values in $[a, b] \subset \mathbb{Z}$ for some integers $a \leq b$.
- `DISCRETE`: Finite, ordered set of values from $\mathbb{R}$.
- `CATEGORICAL`: Unordered list of strings.

```
flat_problem = vz.ProblemStatement()
flat_problem_root = flat_problem.search_space.root
flat_problem_root.add_float_param(name='double', min_value=0.0, max_value=1.0)
flat_problem_root.add_int_param(name='int', min_value=1, max_value=10)
flat_problem_root.add_discrete_param(
    name='discrete', feasible_values=[0.1, 0.3, 0.5])
flat_problem_root.add_categorical_param(
    name='categorical', feasible_values=['a', 'b', 'c'])
```

PyVizier also has a `BOOLEAN` parameter which under-the-hood, is a binary `CATEGORICAL` parameter with values `'True'` and `'False'`.

```
flat_problem_root.add_bool_param(name='bool')
```

A default value for seeding the study may be used when constructing a parameter.

```
flat_problem_root.add_float_param(
    name='double_with_default', min_value=0.0, max_value=1.0, default_value=0.5)
```

Scaling

Each of the numerical parameter types (`DOUBLE`, `INTEGER`, `DISCRETE`) may also have a **scaling type**, which toggles whether optimization occurs over a transformed space.

```
# Default scaling used.
flat_problem_root.add_float_param(
    name='double_uniform',
    min_value=0.0,
    max_value=1.0,
    scale_type=vz.ScaleType.LINEAR)

# Points near min_value are more important.
flat_problem_root.add_float_param(
    name='double_log',
    min_value=0.0,
    max_value=1.0,
    scale_type=vz.ScaleType.LOG)

# Points near the max_value are more important.
flat_problem_root.add_float_param(
    name='double_reverse_log',
    min_value=0.0,
    max_value=1.0,
    scale_type=vz.ScaleType.REVERSE_LOG)

# Default scaling used for DISCRETE parameters.
flat_problem_root.add_discrete_param(
    name='discrete_uniform',
    feasible_values=[0.1, 0.3, 0.5],
    scale_type=vz.ScaleType.UNIFORM_DISCRETE)
```

Conditional search spaces

Sometimes, **child parameters** only exist in specific scenarios or *conditions* when a **parent parameter** is equal to one or more specific values.

Example: Momentum hyperparameters are used by the Adam optimizer, but not stochastic gradient descent (SGD).

Caveat: Since the value of a "learning rate" depends strongly on the optimizer being used (e.g. a learning rate of 0.1 to SGD means completely differently to Adam), we must create two separate child parameters, rather than sharing a single one.

```
conditional_problem = vz.ProblemStatement()
conditional_problem_root = conditional_problem.search_space.root
optimizer = conditional_problem_root.add_categorical_param(
    name='optimizer', feasible_values=['sgd', 'adam'])

# SGD child parameters
optimizer.select_values(['sgd']).add_float_param(
    'sgd_learning_rate',
    min_value=0.0001,
    max_value=1.0,
```

(continues on next page)

(continued from previous page)

```
    scale_type=vz.ScaleType.LOG)

# Adam child parameters
optimizer.select_values(['adam']).add_float_param(
    'adam_learning_rate',
    min_value=0.0001,
    max_value=1.0,
    scale_type=vz.ScaleType.LOG)
optimizer.select_values(['adam']).add_float_param(
    'adam_beta1',
    min_value=0.0,
    max_value=1.0,
    scale_type=vz.ScaleType.REVERSE_LOG)
optimizer.select_values(['adam']).add_float_param(
    'adam_beta2',
    min_value=0.0,
    max_value=1.0,
    scale_type=vz.ScaleType.REVERSE_LOG)
```

Combinatorial Reparamterization

When dealing with a combinatorial search space X, one way to easily deal with such cases is to construct a reparameterization. Mathematically, this means finding a practical search space Z and surjective mapping $\Phi : Z \to X$.

Below is an example over the space of permutations of size N, where our mapping utilizes the Lehmer code.

```
N = 10

# Setup search space.
permutation_problem = vz.ProblemStatement()
for n in range(N):
  permutation_problem.search_space.root.add_int_param(
      name=str(n), min_value=0, max_value=n)


def compute_index(trial: vz.Trial) -> int:
  """Computes index from Lehmer code."""
  index = 0
  for n in range(N):
    index += trial.parameters.get_value(str(n)) * math.factorial(n)
  return index


def compute_permutation(index: int) -> List[int]:
  """Outputs a N-permutation as a list of indices."""
  all_indices = list(range(N))
  temp_index = index
  output = []
  for k in range(1, N + 1):
    factorial_value = math.factorial(N - k)
    value = all_indices[temp_index // factorial_value]
```

(continues on next page)

(continued from previous page)

```
    output.append(value)
    all_indices.remove(value)
    temp_index = temp_index % factorial_value
  return output


def phi(trial: vz.Trial) -> List[int]:
  """Maps a suggestion to a permutation."""
  return compute_permutation(compute_index(trial))
```

Infeasibility

Consider an optimization problem where we only consider float parameters (x, y) from the unit disk $x^2 + y^2 \leq 1$. For such a scenario, we may denote any parameters outside of this area to be **infeasible**.

```
disk_problem = vz.ProblemStatement()
disk_problem_root = disk_problem.search_space.root
disk_problem_root.add_float_param(name='x', min_value=-1.0, max_value=1.0)
disk_problem_root.add_float_param(name='y', min_value=-1.0, max_value=1.0)


def evaluate(trial: vz.Trial) -> vz.Trial:
  x = suggestion.parameters['x']
  y = suggestion.parameters['y']
  if x**2 + y**2 <= 1:
    trial.complete(vz.Measurement(metrics={'sum': x + y}))
  else:
    trial.complete(vz.Measurement(), infeasibility_reason='Outside of range.')
  return trial
```

Supported Algorithms

While we service all algorithms to the user in our policy factory, many can be organized by what level of support we provide to them in addition to their applicable search spaces.

Official

The following algorithms can be considered "official" and production-quality:

1. **GP-Bandit** (`GAUSSIAN_PROCESS_BANDIT`): Flat Search Spaces.
2. **Random Search** (`RANDOM_SEARCH`): Flat Search Spaces.
3. **Quasi-Random Search** (`QUASI_RANDOM_SEARCH`): Flat Search Spaces.
4. **Grid Search** (`GRID_SEARCH`): Flat Search Spaces.
5. **Shuffled Grid Search** (`SHUFFLED_GRID_SEARCH`): Flat Search Spaces.
6. **Eagle Strategy** (`EAGLE_STRATEGY`): Flat Search Spaces.

External + Imported

These algorithms are imported and wrapped from external packages (requiring additional installations via `pip install google-vizier[algorithms]`), and thus we cannot fully control their performance:

1. **CMA-ES** (`CMA_ES`): Continuous (`DOUBLE`) Search Spaces.
2. **Emukit Bayesian Optimization** (`EMUKIT_GP_EI`): Flat Search Spaces.

Reproduced

These algorithms are attempted reproductions of their original papers, sometimes using the authors' original implementations as inspiration (but not as direct imports). While we try our best to ensure their quality, we cannot guarantee exact performance:

1. **NSGA-II** (`NSGA2`): Flat Search Spaces.
2. **Bayesian Optimization of Combinatorial Structures** (`BOCS`): Boolean Search Spaces.
3. **Harmonica** (`HARMONICA`): Boolean Search Spaces.

3.1.2 For Developers

Designers

This documentation will allow a developer to use the Designer API for typical algorithm design.

Installation and reference imports

```
!pip install google-vizier[jax]
```

```
from typing import Optional, Sequence
import numpy as np

from vizier import algorithms as vza
from vizier import pythia
from vizier import pyvizier as vz
from vizier.algorithms import designers
```

Designers

The `Designer` API is an intuitive abstraction for writing and *designing* algorithms. It only requires two basic methods, `update()` and `suggest()`, shown below.

The source of truth for `Designer` can be found here.

```
class Designer(...):
  """Suggestion algorithm for sequential usage."""

  @abc.abstractmethod
  def update(self, completed: CompletedTrials, all_active: ActiveTrials) -> None:
    """Updates recently completed and ALL active trials into the designer's state."""

  @abc.abstractmethod
  def suggest(self, count: Optional[int] = None) -> Sequence[vz.TrialSuggestion]:
    """Make new suggestions."""
```

Every time `update()` is called, the `Designer` will get any newly `COMPLETED` trials since the last `update()` call, and will get all `ACTIVE` trials at the current moment in time.

Note: Trials which may have been provided as `ACTIVE` in previous `update()` calls, can be provided as `COMPLETED` in subsequent `update()` calls.

GP-Bandit Designer Example

The following example, using the default GP-Bandit algorithm, shows how to interact with Vizier designers.

```
# The problem statement (which parameters are being optimized)
problem = vz.ProblemStatement()
problem.search_space.root.add_float_param('x', 0.0, 1.0)
problem.search_space.root.add_float_param('y', 0.0, 1.0)
problem.metric_information.append(
    vz.MetricInformation(
        name='maximize_metric', goal=vz.ObjectiveMetricGoal.MAXIMIZE))

# Create a new designer object
designer = gp_bandit.VizierGPBandit(problem)
# Ask the designer for 5 suggestions
suggestions = designer.suggest(count=2)
```

In this case, since the designer was not update with any `COMPLETED` or `ACTIVE` trials, it will produce suggestions which will look like:

```
[TrialSuggestion(parameters=ParameterDict(_items={'x': 0.5, 'y': 0.5}),␣
↪metadata=Metadata((namespace:, items: {'seeded': 'center'}), current_namespace=)),
 TrialSuggestion(parameters=ParameterDict(_items={'x': 0.10274669379450661, 'y': 0.
↪10191725529767912}), metadata=Metadata((namespace:, items: {}), current_namespace=))]
```

Note that the first suggestion is seeded at the center of the search space, and the second suggestion is random. If we call `designer.suggest()` again before calling `update()`, the designer will produce an identical first suggestion at the center of the search space, and a second random suggestion.

Only when we call `update()`, will the designer update its internal state and generate different suggestions:

```
completed_trials = []
for suggestion in suggestions:
  metric_value = np.random.random()  # Make up a fake metric value.
  suggestion.to_trial().complete(
      vz.Measurement(metrics={'maximize_metric': metric_value})
```

(continues on next page)

(continued from previous page)

```
  )

# Update the designer with the completed trials.
designer.update(vza.CompletedTrials(completed_trials), vza.ActiveTrials())

# Ask for more suggestions.
new_suggestions = designer.suggest(count=2)
```

Thus `COMPLETED` trials should be incrementally updated, while all `ACTIVE` trials are passed to the designer in every `update()` call.

A `Designer` can also be seeded with pre-existing data. Consider the following example:

```
# Make a fresh designer.
designer = designers.VizierGPBandit(problem)

# Create completed trials representing pre-existing training data.
trials = [vz.Trial(parameters={'x': 0.5, 'y': 0.6}).complete(vz.Measurement(metrics={
↪'maximize_metric': 0.3}))]
designer.update(vza.CompletedTrials(trials), vza.ActiveTrials())

# As the designer for suggestions.
suggestions = designer.suggest(count=2)
```

In this case, the designer will **not** return a first trial seeded at the center of the search space, since it has been updated with completed trials. The new suggestions will look something like:

```
[TrialSuggestion(parameters=ParameterDict(_items={'x': 0.7199945005054509, 'y': 0.
↪3800034493548722}), ...]
```

Additional References

- Our designers folder contains examples of designers.
- Our evolution folder contains examples of creating evolutionary designers, such as NSGA2.
- Our designer testing routine contains up-to-date examples on interacting with designers.

Pythia Policies and Hosting Designers

This documentation will allow a developer to:

- Understand the basic structure of a Pythia Policy.
- Host Designers in the service.

Installation and reference imports

```
!pip install google-vizier
```

```
from typing import Optional, Sequence

from vizier import pythia
from vizier import algorithms
from vizier.service import pyvizier as vz
from vizier._src.algorithms.policies import designer_policy
from vizier._src.algorithms.evolution import nsga2
```

Pythia Policies

The Pythia Service maps algorithm names to `Policy` objects. All algorithms which need to be hosted on the server must eventually be wrapped into a `Policy`.

Every `Policy` is injected with a `PolicySupporter`, which is a client used for fetching data from the datastore. This design choice serves two core purposes:

1. The `Policy` is effectively stateless, and thus can be deleted and recovered at any time (e.g. due to a server preemption or failure).
2. Consequently, this avoids needing to save an explicit and potentially complicated algorithm state. Instead, the "algorithm state" can be recovered purely from the entire study containing (`metadata`, `study_config`, `trials`).

We show the `Policy` abstract class explicitly below. Exact class entrypoint can be found here.

```
class Policy(abc.ABC):
  """Interface for Pythia Policy subclasses."""

  @abc.abstractmethod
  def suggest(self, request: SuggestRequest) -> SuggestDecision:
    """Compute suggestions that Vizier will eventually hand to the user."""

  @abc.abstractmethod
  def early_stop(self, request: EarlyStopRequest) -> EarlyStopDecisions:
    """Decide which Trials Vizier should stop."""

  @property
  def should_be_cached(self) -> bool:
    """Returns True if it's safe & worthwhile to cache this Policy in RAM."""
    return False
```

Fundamental Rule of Service Pythia Policies

For algorithms used in the Pythia Service, the fundamental rule is to assume that a Pythia policy class instance will only call once per user interaction:

- `__init__`
- `suggest()`

and be immediately deleted afterwards. Thus a typical policy will use a `stateless_algorithm` and roughly look like:

```
class TypicalPolicy(Policy):

  def __init__(self, policy_supporter: PolicySupporter):
    self._policy_supporter = policy_supporter

  def suggest(self, request: SuggestRequest) -> SuggestDecision:
    all_completed = policy_supporter.GetTrials(status_matches=COMPLETED)
    all_active = policy_supporter.GetTrials(status_matches=ACTIVE)
    suggestions = stateless_algorithm(all_completed, all_active)
    return SuggestDecision(suggestions)
```

Example Pythia Policy

Here, we write a toy policy, where we only act on `CATEGORICAL` parameters for simplicity. The `make_parameters` function will simply for-loop over every category and then cycle back.

```
def make_parameters(
    search_space: vz.SearchSpace, index: int
) -> vz.ParameterDict:
  parameter_dict = vz.ParameterDict()
  for parameter_config in search_space.parameters:
    if parameter_config.type != vz.ParamterType.CATEGORICAL:
      raise ValueError("This function only supports CATEGORICAL parameters.")
    feasible_values = parameter_config.feasible_values
    parameter_dict[parameter_config.name] = vz.ParameterValue(
        value=feasible_values[index % len(feasible_values)]
    )
  return parameter_dict
```

To collect the `index` from the database, we will use the `PolicySupporter` to obtain the maximum trial ID based on completed and active trials.

```
def get_next_index(policy_supporter: pythia.PolicySupporter):
  """Returns current trial index."""
  completed = policy_supporter.GetTrials(status_matches=vz.TrialStatus.COMPLETED)
  active = policy_supporter.GetTrials(status_matches=vz.TrialStatus.ACTIVE)
  trial_ids = [t.id for t in completed + active]

  if trial_ids:
    return max(trial_ids)
  return 0
```

We can now put it all together into our Pythia Policy.

```
class MyPolicy(pythia.Policy):

  def __init__(self, policy_supporter: pythia.PolicySupporter):
    self._policy_supporter = policy_supporter

  def suggest(self, request: pythia.SuggestRequest) -> pythia.SuggestDecision:
    """Gets number of Trials to propose, and produces Trials."""
    suggest_decision_list = []
    for _ in range(request.count):
      index = get_next_index(self._policy_supporter)
      parameters = make_parameters(request.study_config.search_space, index)
      suggest_decision_list.append(vz.TrialSuggestion(parameters=parameters))
    return pythia.SuggestDecision(
        suggestions=suggest_decision_list, metadata=vz.MetadataDelta()
    )
```

Wrapping Designers as Pythia Policies

Consider if your algorithm code fits in the simpler Designer abstraction, which avoids needing to deal with distributed systems logic.

For example, the same exact behavior above can be re-written as a `Designer`:

```
class MyDesigner(algorithms.Designer):

  def __init__(self, study_config: vz.StudyConfig):
    self._study_config = study_config
    self._completed_trials = []
    self._active_trials = []

  def update(
      self,
      completed: algorithms.CompletedTrials,
      all_active: algorithms.ActiveTrials,
  ) -> None:
    self._completed_trials.extend(completed.trials)
    self._active_trials = all_active.trials

  def suggest(
      self, count: Optional[int] = None
  ) -> Sequence[vz.TrialSuggestion]:
    if count is None:
      return []
    trial_ids = [t.id for t in self._completed_trials + self._active_trials]
    current_index = max(trial_ids)
    return [
        make_parameters(self._study_config.search_space, current_index + i)
        for i in range(count)
    ]
```

The entire designer (if deleted or preempted) can conveniently be recovered in just a **single** call of `update()` after

`__init__`.

Thus we may immediately wrap `MyDesigner` into a Pythia Policy with the following Pythia `suggest()` implementation:

- Create the designer temporarily.
- Update the temporary designer with **all** previously completed trials and active trials.
- Obtain suggestions from the temporary designer.

This is done conveniently with the `DesignerPolicy` wrapper (code):

```
class DesignerPolicy(Policy):
  """Wraps a Designer into a Pythia Policy."""

  def __init__(self, supporter: PolicySupporter, designer_factory: Factory[Designer]):
    self._supporter = supporter
    self._designer_factory = designer_factory

  def suggest(self, request: SuggestRequest) -> SuggestDecision:
    completed = self._supporter.GetTrials(status_matches=COMPLETED)
    active = self._supporter.GetTrials(status_matches=ACTIVE)
    designer.update(CompletedTrials(completed), ActiveTrials(active))
    return SuggestDecision(designer.suggest(request.count))
```

Below is the actual act of wrapping:

```
designer_factory = lambda study_config: MyDesigner(study_config)
supporter: PolicySupporter = ... # Assume PolicySupporter was created.
pythia_policy = DesignerPolicy(supporter, designer_factory)
```

Serializing Designer States

The above method can gradually become slower as the number of completed trials in the study increases.

Thus we may consider storing a compressed representation of the algorithm state instead. Examples include:

- The coordinate position in a grid search algorithm.
- The population for evolutionary algorithms such as NSGA2.
- Directory location for stored neural network weights.

As a simple example, consider the case if our designer stores a `_counter` of **all** suggestions it has made:

```
class CounterDesigner(Designer):

  def __init__(self, ...):
    ...
    self._counter = 0

  def suggest(self, count: Optional[int] = None) -> Sequence[TrialSuggestion]:
    ...
    self._counter += len(suggestions)
    return suggestions
```

Vizier offers two Designer subclasses, both of which will use the `Metadata` primitive to store algorithm state data:

- `SerializableDesigner` will use additional `recover/dump` methods and should be used if the entire algorithm state can be easily serialized and can be saved and restored in full.
- `PartiallySerializableDesigner` will use additional `load/dump` methods and be used if the algorithm has subcomponents that are not easily serializable. State recovery will be handled by calling the Designer's `__init__` (with same arguments as before) and then `load`.

They can also be converted into Pythia Policies using `SerializableDesignerPolicy` and `PartiallySerializableDesignerPolicy` respectively.

Below is an example modifying our `CounterDesigner` into `CounterSerialDesigner` and `CounterPartialDesigner` respectively:

```
class CounterSerialDesigner(algorithms.SerializableDesigner):

  def __init__(self, counter: int):
    self._counter = counter

  @classmethod
  def recover(cls, metadata: vz.Metadata) -> CounterSerialDesigner:
    return cls(metadata['counter'])

  def dump(self) -> vz.Metadata:
    metadata = vz.Metadata()
    metadata['counter'] = str(self._counter)
    return metadata


class CounterPartialDesigner(algorithms.PartiallySerializableDesigner):

  def load(self, metadata: vz.Metadata) -> None:
    self._counter = int(metadata['counter'])

  def dump(self) -> vz.Metadata:
    metadata = vz.Metadata()
    metadata['counter'] = str(self._counter)
    return metadata
```

Additional References

- Our policies folder contains examples of Pythia policies.

Early Stopping

This notebook will allow a developer to:

- Understand the Early Stopping API.
- Write Pythia policies for early stopping.

Installation and reference imports

```
!pip install google-vizier
```

```
import numpy as np

from vizier import pythia
```

Early Stopping

In hyperparameter optimization, early stopping is a useful mechanism to prevent wasted resources by stopping unpromising trials. Two main considerations for determining whether to stop an active trial are:

- **At a macro level, how a trial's performance compares to the rest of the trials globally.** For example, we may stop a trial if it is predicted to significantly underperform compared to the history of trials so far in the study.
- **At a micro level, how a trial's intermediate measurements are changing over time.** For example, in a classification task, overfitting may be happening when test accuracy starts to decrease.

API

Based on the above considerations, to allow full flexibility to consider when to stop a trial, we thus use the following abridged API below. Exact class entrypoint can be found here.

The `EarlyStopRequest` takes in a set of trial ID's for early stopping consideration. However, note that trials outside of this set can also be stopped.

```
class EarlyStopRequest:
  """Early stopping request."""

  trial_ids: Optional[FrozenSet[int]]
```

In addition, we have the `EarlyStopDecision` to denote a single trial's stopping condition and the plural `EarlyStopDecisions` for a set of trials:

```
class EarlyStopDecision:
  """Stopping decision on a single trial."""

  id: int
  should_stop: bool
```

```
class EarlyStopDecisions:
  """This is the output of the Policy.early_stop() method."""

  decisions: list[EarlyStopDecision]
  metadata: vz.MetadataDelta
```

They will be used in the Pythia policy's `early_stop` method:

```
class Policy(abc.ABC):
  """Interface for Pythia2 Policy subclasses."""
```

(continues on next page)

(continued from previous page)

```
  @abc.abstractmethod
  def early_stop(self, request: EarlyStopRequest) -> EarlyStopDecisions:
    """Decide which Trials Vizier should stop."""
```

Example usage

As an example, suppose our rule is to stop all requested trials whose 50th intermediate measurement is too low, e.g. bottom 10% of all trials so far.

```
class MyEarlyStoppingPolicy(pythia.Policy):
  """Stops requested trial if its 50th measurement is too low."""

  def __init__(self, policy_supporter: pythia.PolicySupporter, index: int = 50):
    self._policy_supporter = policy_supporter
    self._index = index

  def early_stop(self,
                 request: pythia.EarlyStopRequest) -> pythia.EarlyStopDecisions:
    metric_name = request.study_config.metric_information.item().name

    # Obtain cutoff for 10th percentile.
    all_trials = self._policy_supporter.GetTrials(study_guid=request.study_guid)
    all_metrics = []
    for trial in all_trials:
      if len(trial.measurements) > self._index:
        all_metrics.append(trial.measurements[self._index].metrics[metric_name])
    cutoff = np.percentile(all_metrics, 10)

    # Filter requested trials by cutoff.
    considered_trials = [
        trial for trial in all_trials if trial.id in request.trial_ids
    ]
    stopping_decisions = []
    for trial in considered_trial:
      if considered_trial.measurmenets[
          self._index].metrics[metric_name] < cutoff:
        decision = pythia.EarlyStopDecision(
            id=trial.id, reason='Below cutoff', should_stop=True)
      else:
        decision = pythia.EarlyStopDecision(
            id=trial.id, reason='Above cutoff', should_stop=False)
      stopping_decisions.append(decision)
    return pythia.EarlyStopDecisions(decisions=stopping_decisions)
```

Metadata

We provide a guide below on common developer uses of the `Metadata` primitive.

OSS Vizier can store `Metadata` in both the `ProblemStatement` and each `TrialSuggestion/Trial`, with common use cases:

- Containing additional information outside of standard parameter types.
- Allowing user code to store small amounts of state information inside OSS Vizier, attached to the OSS Vizier study.
- Wrapping search spaces and corresponding algorithms which are naturally incompatible with OSS Vizier's default API, to still allow a distributed backend service.

Installation and reference imports

```
!pip install google-vizier
```

```
from vizier import pyvizier as vz
from google.protobuf import any_pb2
```

Metadata basics

The `Metadata` is a key-value store, where:

- Keys are UTF-8 strings.
- Values can be strings or protocol buffers.

While values of type `int`, `float`, and more complex objects can also be used, **the developer is responsible for serializing / unserializing said objects.**

```
metadata = vz.Metadata()
metadata['proto'] = any_pb2.Any(...)
metadata['string'] = 'hello'
```

Additionally, `Metadata` can act as a "dictionary of dictionaries", i.e. a hierarchy of dictionaries, via its `Namespace` functionality via calling `.ns()`, which creates another `Metadata` which shares data with the original.

```
child_metadata = metadata.ns('child')

grandchild_metadata = child_metadata.ns('child')
grandchild_metadata['string'] = 'goodbye'

assert metadata.ns('child').ns('child')['string'] == 'goodbye'
```

ProblemStatement Metadata

The `ProblemStatement` object contains a `metadata` attribute, ideally for storing global metadata related to the study. Note that `Metadata` will not be used in the optimization process, UNLESS there is a custom algorithm configured to use it.

Below is a usage example when training an image classifier, where one may wish to store training-related attributes in `Metadata`.

```
problem_statement = vz.ProblemStatement()
problem_statement.metadata['dataset'] = 'cifar10'
problem_statement.metadata['architecture'] = 'resnet_18'
```

Trial Metadata

`TrialSuggestion` and subclass `Trial` also contain a `metadata` attribute. This in contrast, should be used to store metadata related to the specific Trial.

In the image classification case, examples would be the type of GPU used for training and if the training worker has been preempted.

```
trial = vz.Trial()
trial.metadata['gpu_used'] = 'P100'
trial.metadata['preempted'] = 'True'
```

OSS Vizier as a backend via `Metadata`

As an advanced developer use case, one may extend OSS Vizier's search space capabilities using `Metadata`. Custom algorithms can provide full freedom in expressing more complex search spaces (e.g. graphs) using `Metadata`.

Example use cases:

- Combinatorial optimization, where the search space may consist of graphs or multiple selection (e.g. $\binom{N}{K}$) primitives. Algorithms commonly include evolutionary methods, which also require custom mutation operations.
- Free-form textual data used for suggestions (and maybe even evaluation metrics!), as common with language-based applications.

```
# Setup combinatorial search space.
choose_problem = vz.ProblemStatement()
choose_problem.metadata = vz.Metadata({'N': '10', 'K': '3'})

# Example of a suggestion proposed by a custom algorithm.
suggestion = vz.TrialSuggestion()
suggestion.metadata['chosen_indices'] = '[0, 3, 7]'
```

The algorithm behavior can even be changed mid-optimization with `Metadata` using a client! This is in fact used extensively in our integrations with Pyglove to allow a running Pythia policy to change search spaces or mutations online.

```
# Original mutation rate.
mutation_problem = vz.ProblemStatement()
mutation_problem.metadata = vz.Metadata({'mutation_rate': '0.1'})
```

(continues on next page)

(continued from previous page)

```
# ...
# Assume algorithm started running in the Pythia service.
# ...

# Set new mutation rate.
study_metadata = vz.Metadata()
study_metadata['mutation_rate'] = '0.2'

# Prevent this trial from being used in the population.
trial_metadata = vz.Metadata({'use_in_population' = 'False'})
trial_id = 1

# Assume client was already created, and commit the metadata update.
metadata_delta = pyvizier.MetadataDelta(
    on_study=study_metadata, on_trials={trial_id: trial_metadata})
client.update_metadata(metadata_delta)
```

Predictors

This documentation will allow a developer to understand and use the `Predictor` API.

Installation and reference imports

```
!pip install google-vizier[jax]
```

```
import numpy as np
from vizier._src.benchmarks.experimenters.synthetic import bbob
from vizier.algorithms import designers
from vizier import algorithms as vza
from vizier import pyvizier as vz
import matplotlib.pyplot as plt
```

Predictors

The `Predictor` exposes a `predict()` method which takes `TrialSuggestions` as inputs and returns their corresponding objective value predictions, represented by a `Prediction` class.

The source of truth for predictors can be found here.

```
class Prediction:
  """Container to hold predictions."""

  mean: chex.Array
  stddev: chex.Array
  metadata: Optional[Metadata] = None
```

(continues on next page)

(continued from previous page)

```
class Predictor(abc.ABC):
  """Mixin for algorithms to expose prediction API."""

  @abc.abstractmethod
  def predict(
      self,
      trials: Sequence[TrialSuggestion],
      ...
  ) -> Prediction:
```

In some cases involving a underlying probabilistic model, there's a need to sample the posterior distribution in order to obtain the mean and standard deviation. In this case, the API allows specifying the the random key and number of samples (hidden arguments as `...` in `predict()` above).

GP-Bandit Predict Example

The `VizierGPBandit` class acts as both a `Designer` and a `Predictor` and allows users to obtain the underlying GP model's mean and standard deviation for a given set of points. The example below demonstrates this capability.

Setup problem statement and objective:

```
# The problem statement (which parameters are being optimized)
problem = vz.ProblemStatement()
problem.search_space.root.add_float_param('x', -5.0, 5.0)
problem.metric_information.append(
    vz.MetricInformation(
        name='obj', goal=vz.ObjectiveMetricGoal.MAXIMIZE))

# The real objective function used for generating observations.
f = lambda x: bbob.Weierstrass(np.array([x]))
```

Create observations (i.e. completed trials) of the objective function.

```
# Generate suggestions.
observations = designers.QuasiRandomDesigner(problem.search_space).suggest(30)

# Compute the real objective value and complete the trials.
trials = []
for idx, obs in enumerate(observations):
  trials.append(
      obs.to_trial(idx).complete(
          vz.Measurement(metrics={'obj': f(obs.parameters['x'].value)})
      )
  )
```

Create a `VizierGPBandit` designer and update it with the observations.

Note: When two `VizierGPBandit` designers are updated with identical trials, they may still produce slightly different models and predictions due to inherent stochasticity during the training process.

```
# Create the GPBandit designer.
gp_designer = designers.VizierGPBandit(problem)

# Update the GP-Bandit designer with completed trials.
gp_designer.update(vza.CompletedTrials(trials), vza.ActiveTrials())
```

Generate predictions in arbitrary points.

```
# Generate predictions.
suggestions = designers.GridSearchDesigner(problem.search_space, double_grid_
↪resolution=500).suggest(500)
predictions = gp_designer.predict(suggestions)
```

Plot the predictions.

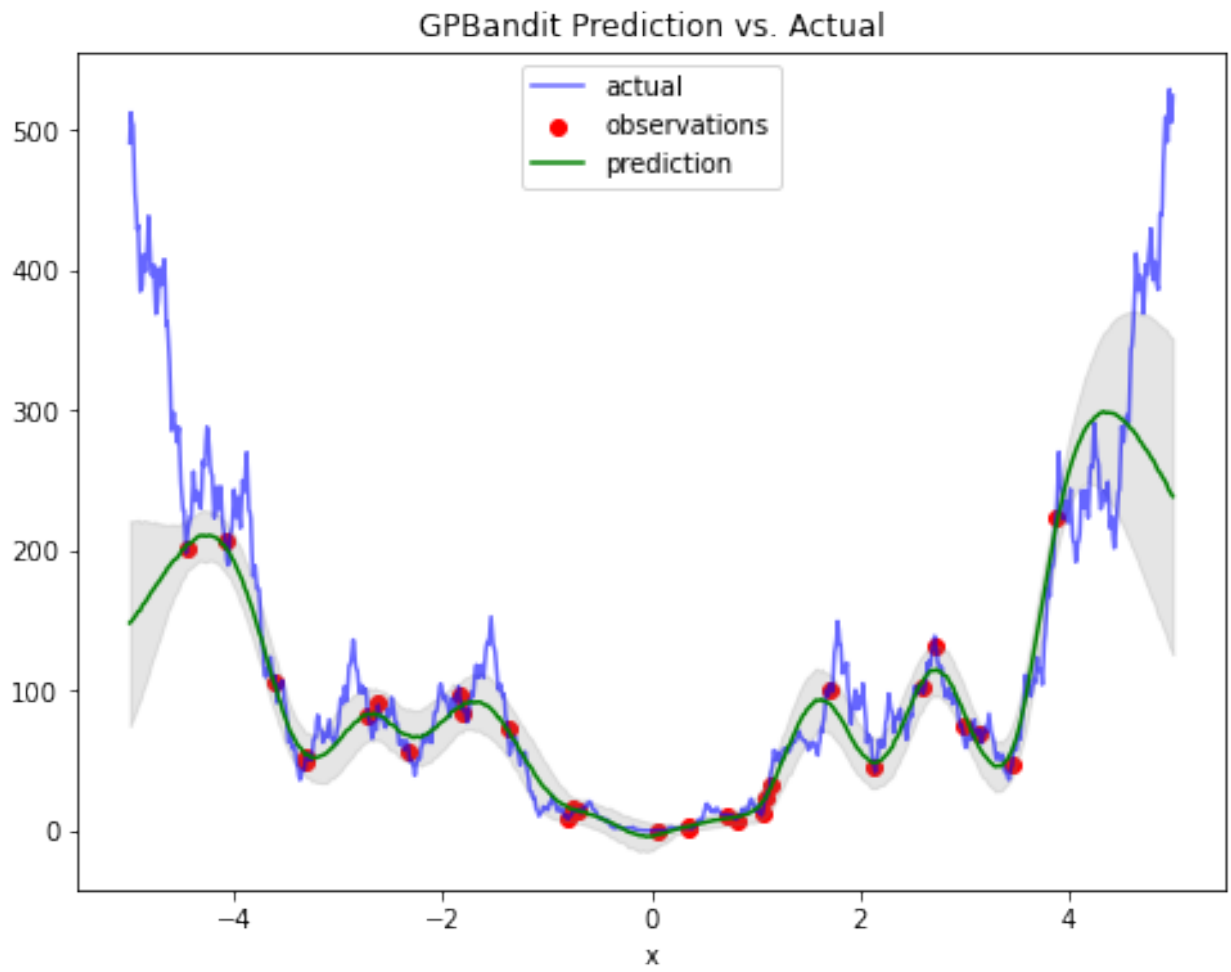
```
plt.figure(figsize=(8, 6))

# Visualize the real objective function.
xs = np.linspace(-5, 5, num=1000)
ys = [f(x) for x in xs]
plt.plot(xs, ys, label='actual', color='blue', alpha=0.6)

# Visualize the observation points.
obs_x = [obs.parameters['x'].value for obs in observations]
obs_y = [f(x) for x in obs_x]
plt.scatter(obs_x, obs_y, label='observations', marker='o', color='red')

# Visualize the predictions and confidence bounds.
pred_x = [suggestion.parameters['x'].value for suggestion in suggestions]
plt.plot(pred_x, predictions.mean, label='prediction', color='green')
lower = predictions.mean - predictions.stddev
upper = predictions.mean + predictions.stddev
plt.fill_between(pred_x, lower, upper, color='grey', alpha=0.2)

# Add legend and title.
plt.legend(loc='best')
plt.title(f'GPBandit Prediction vs. Actual')
plt.xlabel('x')
plt.show()
```



3.1.3 For Benchmarking

Creating Benchmarks

We provide a guide below on creating benchmarks, through the use of either:

- Standard search space primitives.
- Metadata for complex search spaces.

Installation and reference imports

```
!pip install google-vizier
```

```
import abc
import random
from typing import Sequence
from vizier import pyvizier
from vizier.benchmarks import experimenters
```

Experimenters

The core base class of any objective function is the `Experimenter` class, which simply contains a method to evaluate a `Trial` and a `ProblemStatement` to describe its search space and metrics. The exact entry into the class can be found here.

```
class Experimenter(metaclass=abc.ABCMeta):
  """Abstract base class for Experimenters."""

  @abc.abstractmethod
  def evaluate(self, suggestions: Sequence[pyvizier.Trial]):
    """Evaluates and mutates the Trials in-place."""
    pass

  @abc.abstractmethod
  def problem_statement(self) -> pyvizier.ProblemStatement:
    """The search configuration generated by this experimenter."""
    pass
```

Below is an example of a basic 1D objective function $f(x) = x^2$.

```
class Basic1DExperimenter(experimenters.Experimenter):

  def evaluate(self, suggestions: Sequence[pyvizier.Trial]):
    problem_statement = self.problem_statement()
    for suggestion in suggestions:
      x = suggestion.parameters['x'].value
      objective = x**2
      suggestion.complete(
          pyvizier.Measurement(metrics={
              problem_statement.single_objective_metric_name: objective
          }))

  def problem_statement(self) -> pyvizier.ProblemStatement:
    problem_statement = pyvizier.ProblemStatement()
    root = problem_statement.search_space.root
    root.add_float_param(name='x', min_value=-1.0, max_value=1.0)
    problem_statement.metric_information.append(
        pyvizier.MetricInformation(
            name='main_objective', goal=pyvizier.ObjectiveMetricGoal.MAXIMIZE))
    return problem_statement
```

We may thus evaluate a suggestion. Note that such suggestions are actually `Trials`, to allow maximum flexibility.

```
basic_experimenter = Basic1DExperimenter()
trial = pyvizier.Trial()
trial.parameters['x'] = 0.1

basic_experimenter.evaluate([trial])
assert trial.final_measurement.metrics['main_objective'].value == 0.1 ** 2
```

Metadata-based Experimenters

Similar to using the `Metadata` primitive to create custom algorithms and complex search spaces, creating custom `Experimenters` provides the freedom to define custom objective functions.

As an example, suppose our search space consisted of unbounded-length sequences consisting of some vocabulary (e.g. the letters 'A' to 'Z' if considering the space of English words), and we wish to maximize the sequence's average ASCII value.

```
class VocabularyExperimenter(experimenters.Experimenter):

  def evaluate(self, suggestions: Sequence[pyvizier.Trial]):
    problem_statement = self.problem_statement()
    for suggestion in suggestions:
      x = suggestion.metadata['word']
      objective = float(sum([ord(c) for c in x])) / len(x)
      suggestion.complete(
          pyvizier.Measurement(metrics={
              problem_statement.single_objective_metric_name: objective
          }))

  def problem_statement(self) -> pyvizier.ProblemStatement:
    problem_statement = pyvizier.ProblemStatement()
    problem_statement.metadata['vocab'] = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    problem_statement.metric_information.append(
        pyvizier.MetricInformation(
            name='main_objective', goal=pyvizier.ObjectiveMetricGoal.MAXIMIZE))
    return problem_statement
```

Below is an example of constructing a valid suggestion and evaluating it.

```
vocab_experimenter = VocabularyExperimenter()
vocabulary = vocab_experimenter.problem_statement().metadata['vocab']
trial = pyvizier.Trial()
trial.metadata['word'] = str(
    [random.randint(0, len(vocabulary)) for _ in range(10)])

vocab_experimenter.evaluate([trial])
print('Average ASCII value is:',
      trial.final_measurement.metrics['main_objective'].value)
```

Running Benchmarks

We will demonstrate below how to use our benchmark runner pipeline.

Installation and reference imports

```
!pip install google-vizier
```

```
from vizier import algorithms as vza
from vizier import benchmarks
from vizier.algorithms import designers
from vizier.benchmarks import experimenters
```

Example experimenter and designer factory which we will use later.

```
experimenter = experimenters.NumpyExperimenter(
    experimenters.bbob.Sphere, experimenters.bbob.DefaultBBOBProblemStatement(5)
)

designer_factory = designers.GridSearchDesigner.from_problem
```

Algorithms and Experimenters

Every study can be seen conceptually as a simple loop between an algorithm and objective. In terms of code, the algorithm corresponds to a `Designer`/`Policy` and objective to an `Experimenter`.

Below is a simple sequential loop.

```
designer = designer_factory(experimenter.problem_statement)

for _ in range(100):
  suggestion = designer.suggest()[0]
  trial = suggestion.to_trial()
  experimenter.evaluate([trial])
  completed_trials = vza.CompletedTrials([trial])
  designer.update(completed_trials)
```

As seen above however, one modification we can make is to use variable batch sizes, rather than only suggesting and evaluating one-by-one. More generally, certain implementation details may arise:

- How many parallel suggestions should the algorithm generate?
- How many suggestions can be evaluated at once?
- Should we use early stopping on certain unpromising trials?
- Should we use a custom stopping condition instead of a fixed for-loop?
- Can we swap in a different algorithm mid-loop?
- Can we swap in a different objective mid-loop?

API

The code flexibility needed to simulate these real-life scenarios may cause complications as the evaluation benchmark may no longer be stateless. In order to broadly cover such scenarios, our API introduces the `BenchmarkSubroutine`:

```
class BenchmarkSubroutine(Protocol):
  """Abstraction for core benchmark routines.

  Benchmark protocols are modular alterations of BenchmarkState by reference.
  """

  def run(self, state: BenchmarkState) -> None:
    """Abstraction to alter BenchmarkState by reference."""
```

All routines use and potentially modify a `BenchmarkState`, which holds information about the objective via an `Experimenter` and the algorithm itself wrapped by an `AlgorithmRunnerProtocol`.

```
class BenchmarkState:
  """State of a benchmark run. It is altered via benchmark protocols."""

  experimenter: Experimenter
  algorithm: runner_protocol.AlgorithmRunnerProtocol

  @classmethod
  def from_designer_factory(cls, designer_factory: DesignerFactory,
                            experimenter: Experimenter) -> 'BenchmarkState':

  @classmethod
  def from_policy_factory(cls, policy_factory: PolicyFactory,
                          experimenter: Experimenter) -> 'BenchmarkState':
```

To wrap multiple `BenchmarkSubRoutines` together, we can use the `BenchmarkRunner`:

```
class BenchmarkRunner(BenchmarkSubroutine):
  """Run a sequence of subroutines, all repeated for a few iterations."""

  # A sequence of benchmark subroutines that alter BenchmarkState.
  benchmark_subroutines: Sequence[BenchmarkSubroutine]
  # Number of times to repeat applying benchmark_subroutines.
  num_repeats: int

  def run(self, state: BenchmarkState) -> None:
    """Run algorithm with benchmark subroutines with repetitions."""
```

Example usage

Below is a typical example of simple suggestion and evaluation:

```
runner = benchmarks.BenchmarkRunner(
    benchmark_subroutines=[
        benchmark_runner.GenerateSuggestions(),
        benchmark_runner.EvaluateActiveTrials()
    ],
    num_repeats=100)

benchmark_state = benchmarks.BenchmarkState.from_designer_factory(
    designer_factory=designer_factory, experimenter=experimenter)

runner.run(benchmark_state)
```

We may obtain the evaluated trials via the `benchmark_state`, which contains a `PolicySupporter` via its `algorithm` field:

```
all_trials = benchmark_state.algorithm.supporter.GetTrials()
print(all_trials)
```

Note that this design is maximally informative on everything that has happened so far in the study. For instance, we may also query incomplete/unused suggestions using the `PolicySupporter`.

References

- Benchmark Runners can be found in `benchmark_runner.py`.

3.2 Advanced Topics

3.2.1 Tensorflow Probability

Bayesian Optimization Modeling

The goal of this tutorial is to introduce Bayesian optimization workflows in OSS Vizier, including the underlying TensorFlow Probability (TFP) components and JAX/Flax functionality. The target audience is researchers and practitioners already well-versed in Bayesian optimization, who want to **define and train their own Gaussian Process surrogate models** for Bayesian optimization in OSS Vizier.

Additional resources for TFP

If you're new to TFP, a good place to start is A tour of TensorFlow Probability. TFP began as a TensorFlow-only library, but now has a JAX backend that is entirely independent of TensorFlow (such that "Tensor-Friendly Probability" might be a better backronym). This Colab uses TFP's JAX backend (see the "Imports" cell for how to import it).

Additional resources for Flax

OSS Vizier's Bayesian Optimization models are defined as Flax modules.

Imports

```
import chex
import jax
from jax import numpy as jnp, random, tree_util
import numpy as np
import optax
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from tensorflow_probability.substrates import jax as tfp
from typing import Any

# Vizier models can freely access modules from vizier._src
from vizier._src.benchmarks.experimenters.synthetic import bbob
from vizier._src.jax.optimizers import optimizers
from vizier._src.jax import stochastic_process_model as spm

tfd = tfp.distributions
tfb = tfp.bijectors
tfpk = tfp.math.psd_kernels
```

Defining a GP surrogate model and hyperparameters

To write a GP surrogate model, first write a coroutine that yields parameter specifications (`ModelParameter`) and returns a GP distribution. Downstream, the parameter specifications are used to define Flax module parameters. The inputs to the coroutine function represent the index points of the GP (in the remainder of this Colab, we refer to "inputs" and "index points" interchangeably).

The rationale for the coroutine design is that it lets us automate the application of the parameter constraint and initialization functions (corresponding to hyperpriors, e.g.), and enables simultaneous specification of the model parameters and how their values are used to instantiate a GP.

Coroutine example

The following cell shows a coroutine defining a GP with a squared exponential kernel and two parameters: the length scale of the kernel and the observation noise variance of the GP.

```
def simple_gp_coroutine(inputs: chex.Array=None):
  length_scale = yield spm.ModelParameter.from_prior(
      tfd.Gamma(1., 1., name='length_scale'))
  amplitude = 2.  # Non-trainable parameters may be defined as constants.
  kernel = tfpk.ExponentiatedQuadratic(
      amplitude=amplitude, length_scale=length_scale)
  observation_noise_variance = yield spm.ModelParameter(
      init_fn=lambda x: jnp.exp(random.normal(x)),
      constraint=spm.Constraint(bounds=(0.0, 100.0), bijector=tfb.Softplus()),
      regularizer=lambda x: x**2,
      name='observation_noise_variance')
  return tfd.GaussianProcess(
      kernel,
      index_points=inputs,
      observation_noise_variance=observation_noise_variance)
```

ModelParameter

`ModelParameter` may be used to define hyperpriors.

Parameter specifications from priors

The length scale parameter has a Gamma prior. This is equivalent to defining a `ModelParameter` with a regularizer that computes the Gamma negative log likelihood and an initialization function that samples from the Gamma distribution. As the constraint was not specified, a default one is assigned which is the "default event space bijector" of the TFP distribution (each TFP distribution has a constraining bijector that maps the real line to the support of the distribution).

Specifying parameters explicitly

Observation noise variance, which is passed to the Gaussian process and represents the scalar variance of zero-mean Gaussian noise in the observed labels, is not given a `tfd.Distribution` prior. Instead, it has its initialization, constraining, and regularization functions defined individually. Note that the initialization function is in the constrained space.

Constraints

ModelParameter allows to define constraints on the model parameters using the 'Constraint' object which is initiated with a tuple of 'bounds' and 'bijector' function.

Though the constraints are defined as part of the ModelParameter the Flax model itself does not use them, but rather it expects to receive parameter values already in the constrained space. This means that it's the responsibility of the user/optimizer to pass the GP parameter values that are already in the constrained space.

Exercise: Write a GP model

Write an ARD Gaussian Process model with three parameters: `signal_variance`, `length_scale`, and `observation_noise_variance`. (This is a slightly simplified version of the Vizier GP.)

- `signal_variance` and `observation noise_variance` are both:
  - regularized by the function $f(x) = 0.01 \log(x)^2$
  - bounded to be positive.
- `signal_variance` parameterizes a Matern 5/2 kernel, where the amplitude of the kernel is the square root of `signal_variance`. Use `tfpk.MaternFiveHalves`.
- `length_scale` has a $LogNormal(0, 1)$ prior for each dimension. Assume there are 4 dimensions, and use `tfd.Sample` to build a 4-dimensional distribution consisting of IID LogNormal distributions. (Note that the `length_scale` parameter is a vector – all other parameters are scalars.)
- In TFP, ARD kernels are implemented with `tfpk.FeatureScaled`, with `scale_diag` representing the length scale along each dimension.

```
def vizier_gp_coroutine(inputs: chex.Array=None):
  pass
```

Solution

```
data_dimensionality = 2

def vizier_gp_coroutine(inputs: chex.Array=None):
  """A coroutine that follows the `ModelCoroutine` protocol."""
  signal_variance = yield spm.ModelParameter(
      init_fn=lambda x: tfb.Softplus()(random.normal(x)),
      constraint=spm.Constraint(bounds=(0.0, 100.0), bijector=tfb.Softplus()),
      regularizer=lambda x: 0.01 * jnp.log(x)**2,
      name='signal_variance')
  length_scale = yield spm.ModelParameter.from_prior(
    tfd.Sample(
        tfd.LogNormal(loc=0., scale=1.),
        sample_shape=[data_dimensionality],
        name='length_scale'),
    constraint=spm.Constraint(bounds=(0.0, None)))
  kernel = tfpk.MaternFiveHalves(
      amplitude=jnp.sqrt(signal_variance), validate_args=True)
  kernel = tfpk.FeatureScaled(
      kernel, scale_diag=length_scale, validate_args=True)
  observation_noise_variance = yield spm.ModelParameter(
      init_fn=lambda x: jnp.exp(random.normal(x)),
      constraint=spm.Constraint(bounds=(0.0, 100.0), bijector=tfb.Softplus()),
      regularizer=lambda x: 0.01 * jnp.log(x)**2,
      name='observation_noise_variance')
  return tfd.GaussianProcess(
      kernel=kernel,
      index_points=inputs,
      observation_noise_variance=observation_noise_variance,
      validate_args=True)
```

To build a GP Flax module, instantiate a `StochasticProcessModel` with a GP coroutine as shown below. The module runs the coroutine in the `setup` and `__call__` methods to initialize the parameters and then instantiate the GP object with the given parameters.

Recall that Flax modules have two primary methods: `init`, which initializes parameters, and `apply`, which computes the model's forward pass given a set of parameters and input data.

```
model = spm.StochasticProcessModel(coroutine=vizier_gp_coroutine)

# Sample some fake data.
# Assume we have `num_points` observations, each with `dim` features.
num_points = 12

# Sample a set of index points.
index_points = np.random.normal(
    size=[num_points, data_dimensionality]).astype(np.float32)

# Sample function values observed at the index points
observations = np.random.normal(size=[num_points]).astype(np.float32)

# Call the Flax module's `init` method to obtain initial parameter values.
init_params = model.init(random.PRNGKey(0), index_points)
```

We can observe the initial parameters values of the Flax model and see that they match with the 'ModelParameter' definitions in our coroutine.

```
print(init_params['params'])
```

To instantiate a GP with a set of parameters and index points, use the Flax module's `apply` method. `apply` also returns the regularization `losses` for the parameters, in `mutables`. The regularization `losses` are treated as mutable state because they are recomputed internally with each forward pass of the model. For more on mutable state in Flax, see this tutorial.

```
gp, mutables = model.apply(
    init_params,
    index_points,
    mutable=['losses'])
assert isinstance(gp, tfd.GaussianProcess)
```

Optimizing hyperparameters

Exercise: Loss function

Write down a loss function that takes a parameters dict and returns the loss value, using `model.apply`. The function will close over the observed data.

The loss should be the sum of the GP negative log likelihood and the regularization losses. The regularization loss values are computed when the module is called, using the `ModelParameter` regularization functions. They are stored in a mutable variable collection called `"losses"`, using the Flax method `sow`.

```
def loss_fn(params):
  ...
  return loss, {}  # Return an empty dict as auxiliary state.
```

Solution

```
def loss_fn(params):
  gp, mutables = model.apply({'params': params},
                             index_points,
                             mutable=['losses'])
  loss = (-gp.log_prob(observations) +
          jax.tree_util.tree_reduce(jnp.add, mutables['losses']))  # add the␣
↪regularization losses.
  return loss, {}
```

The gradients of the loss have the same structure as the `params` dict.

```
grads = jax.grad(loss_fn, has_aux=True)(init_params['params'])[0]
print(grads)
```

We can use `jax.tree_util` to take a step along the gradient (though in practice, with Optax, we can use `update` and `apply_updates` to update the parameters at each train step).

```
learning_rate = 1e-3
updated_params = jax.tree_util.tree_map(
    lambda p, g: p - learning_rate * g,
    init_params['params'],
    grads)
print(updated_params)
```

Optimize hyperparameters with Vizier optimizers

Flax modules are often optimized using Optax which requires the developer to write a routine that initializes parameter values and then repeatedly computes the loss function gradients and updates the parameter values accordingly.

Vizier Optimizers is a library of optimizers that automate the process of finding the optimal Flax parameter values and wrap optimizers from libraries such as Optax and Jaxopt in a common interface. To use a Vizier Optimizer you have to specify the following:

- `setup` function which is used to generate the initial parameter values.
- `loss_fn` function which is used for computing the loss function value and gradients. For example, the loss function of a GP model would be a marginal likelihood plus the parameters regularizations.
- `rng` PRNGKey for controlling pseudo randomization.
- `constraints` on the parameters (optional).

Below we use the Vizier `JaxoptLbfgsB` optimizer to run a constrained L-BFGS-B algorithm. Unconstrainted optimizers (e.g. Adam) use a bijector function to map between the unconstrained space where the search is performed, and the constrained space where the loss function is evaluated. On the contrary, constrained optimizers (e.g. L-BGFS-B) use the constraint bounds directly in the search process.

To pass the constraints bounds to the `JaxoptLbfgsB` optimizer we use the `spm.get_constraints` function that traverse the parameters defined in the module coroutine and extract their bounds.

```
setup = lambda rng: model.init(rng, index_points)['params']
model_optimizer = optimizers.JaxoptLbfgsB(
      random_restarts=20, best_n=None
```

(continues on next page)

(continued from previous page)

```
  )
constraints = spm.get_constraints(model)
optimal_params, _ = model_optimizer(setup, loss_fn, random.PRNGKey(0),
                                    constraints=constraints)
```

Predict on new inputs, conditional on observations

To compute the posterior predictive GP on unseen points, conditioned on observed data, use the `precompute_predictive` and `posterior_predictive` methods of the Flax module. `precompute_predictive` must be called first; it runs and stores the Cholesky decomposition of the kernel matrix for the observed data. `posterior_predictive` then returns a posterior predictive GP at new index points, avoiding recomputation of the Cholesky.

```
# Precompute the Cholesky.
_, pp_state = model.apply(
    {'params': optimal_params},
    index_points,
    observations,
    mutable=['predictive'],
    method=model.precompute_predictive)

# Predict on new index points.
predictive_index_points = np.random.normal(
    size=[5, data_dimensionality]).astype(np.float32)
pp_dist = model.apply(
    {'params': optimal_params, **pp_state},
    predictive_index_points,
    index_points,
    observations,
    method=model.posterior_predictive)

# `posterior_predictive` returns a TFP distribution, whose mean, variance, and
# samples we can use to compute an acquisition function.
assert pp_dist.mean().shape == (5,)
```

Optimize a black-box function

For an end-to-end example of Bayesian optimization, we'll use the GP surrogate model defined above along with an Upper Confidence Bound acquisition function to try to find the maximum of the Weierstrass function. First, visualize the function surface.

```
# Use the Weierstrass function from Vizier's Black-Box Optimization Benchmarking
# (BBOB) library.
bb_fun = bbob.Weierstrass

# Sample a set of index points in a 2D space.
num_points = 6
max_x = np.array(2.).astype(np.float32)
index_points = random.uniform(
```

(continues on next page)

(continued from previous page)

```
    random.PRNGKey(3),
    shape=[num_points, data_dimensionality], dtype=jnp.float32) * max_x

# Compute function values observed at the index points.
observations = np.apply_along_axis(
    bb_fun, axis=1, arr=index_points).astype(np.float32)

# Define a grid of points in the function domain for plotting.
n_grid = 100
x = y = np.linspace(0, max_x, n_grid, dtype=np.float32)
X, Y = np.meshgrid(x, y)
x_grid = np.vstack([X.ravel(), Y.ravel()]).T
y_grid = np.apply_along_axis(bb_fun, axis=1, arr=x_grid)
Z = y_grid.reshape(X.shape)

# Plot the black-box function values.
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, alpha=0.5)
ax.scatter(index_points[:, 0], index_points[:, 1], observations, color='r',
           label='Initial observed data')
plt.title('Black-box (Weierstrass) function values and observed data')
plt.legend()
plt.show()
```

Next, run a few iterations of Bayesian optimization to maximize the black-box function given the observed data. A single iteration consists of the following steps:

1. Optimize the GP hyperparameters.
2. Find a suggestion that maximizes an Upper Confidence Bound acquisition function. In this example, we use grid search for the optimization.
3. Evaluate the black-box function on the suggestion and append it to the set of observed data.

(Note that this simple Bayesopt algorithm is for educational purposes and that we'd expect Vizier's GP bandit algorithm to give better results.)

```
num_bayesopt_iter = 5

# At each iteration, redefine the loss function given the current observed data.
def build_loss_fn(index_points, observations):
  def loss_fn(params):
    gp, mutables = model.apply({'params': params},
                               index_points,
                               mutable=['losses'])
    loss = (-gp.log_prob(observations) +
            jax.tree_util.tree_reduce(jnp.add, mutables['losses']))  # add the␣
↪regularization losses.
    return loss, {}
  return loss_fn

for i in range(num_bayesopt_iter):
  # Update the loss function to condition on all observed data.
```

(continues on next page)

(continued from previous page)

```
  loss_fn = build_loss_fn(index_points, observations)

  # Optimize the GP hyperparameters.
  optimal_params, _ = model_optimizer(setup, loss_fn, random.PRNGKey(0),
                                      constraints=constraints)

  # Compute the posterior predictive distribution over a grid of points in the
  # function domain (x_grid).
  _, pp_state = model.apply(
      {'params': optimal_params},
      index_points,
      observations,
      mutable=['predictive'],
      method=model.precompute_predictive)
  pp_dist = model.apply(
      {'params': optimal_params, **pp_state},
      x_grid,
      index_points,
      observations,
      method=model.posterior_predictive)

  # Compute the acquisition function value at each point in the grid.
  pred_mean = pp_dist.mean()
  ucb_vec = pred_mean + 2. * pp_dist.stddev()

  # Find the grid point with the highest acquisition function value.
  ind = np.argmax(ucb_vec)

  # Evaluate the black box function at the selected point.
  f_val = bb_fun(x_grid[ind])

  # Visualize the surrogate model mean and acquisition function surface at this
  # iteration.
  fig = plt.figure(figsize=(10, 10))
  ax = fig.add_subplot(121, projection='3d')
  W = pred_mean.reshape(X.shape)
  ax.plot_surface(X, Y, W, alpha=0.5)
  ax.scatter(index_points[:, 0], index_points[:, 1], observations, color='r',
             label='Observed data')
  ax.set_title('Observed data and posterior predictive GP mean')
  ax.legend()

  ax = fig.add_subplot(122, projection='3d')
  ucb = ucb_vec.reshape(X.shape)
  ax.plot_surface(X, Y, ucb, alpha=0.5)
  ax.scatter(*x_grid[ind], ucb_vec[ind], color='r', label='New suggestion')
  ax.set_title('Acquisition function')
  ax.legend()
  plt.show()

  # Append the new suggestion and function value to the set of observations.
  index_points = np.concatenate([index_points, x_grid[ind][np.newaxis]])
```

(continues on next page)

(continued from previous page)

```
  observations = np.concatenate(
      [observations, np.array(f_val).astype(np.float32)[np.newaxis]])

  print(f'Iteration: {i}')
  print(f'Acquisition function value at suggestion: {ucb_vec[ind]}')
  print(f'Black-box function value at suggestion: {f_val}')
```

Deeper dive on selected topics in TFP

As shown above, the Flax GP model makes use of a number of TFP components:

- Distributions specify parameter priors (e.g. `tfd.Gamma`). The stochastic process model itself is also a TFP distribution, `tfd.GaussianProcess`.
- Bijectors (e.g. `tfb.Softplus`) are used to constrain parameters for optimization, and may also be used for input/output warping.
- PSD kernels (e.g. `tfpk.ExponentiatedQuadratic`) specify the kernel function for the stochastic process.

The next sections of this Colab introduce these and how they're used in Bayesopt modeling.

`tfd.GaussianProcess` and friends

The stochastic process Flax modules return a TFP distribution in the Gaussian Process family (an instance of `tfd.GaussianProcess`, `tfd.StudentTProcess`, or `tfde.MultiTaskGaussianProcess`).

This Colab doesn't go into detail on TFP distributions, since advanced usage and implementation of distributions is rarely required for Bayesopt modeling with Vizier. For an overview of TFP distributions, see TensorFlow Distributions: A Gentle Introduction.

Some of the methods of the Gaussian Process distribution are demonstrated below. Gaussian Process Regression in TFP is also worth reading.

```
# Build a kernel function (see "PSD kernels" section below) and GP.
num_points = 6
index_points = random.uniform(
    random.PRNGKey(3),
    shape=[num_points, data_dimensionality], dtype=jnp.float32)
observations = random.uniform(
    random.PRNGKey(4),
    shape=[num_points], dtype=jnp.float32)

kernel = tfpk.MaternFiveHalves(
    amplitude=2.,
    length_scale=0.3,
    validate_args=True  # Run additional runtime checks; possibly expensive.
    )
observation_noise_variance = jnp.ones([], dtype=observations.dtype)
gp = tfd.GaussianProcess(
    kernel,
    index_points=index_points,
    observation_noise_variance=observation_noise_variance,
    always_yield_multivariate_normal=True,  # See commentary below.
```

(continues on next page)

(continued from previous page)

```
    cholesky_fn=lambda x: tfp.experimental.distributions.marginal_fns.retrying_
↪cholesky(x)[0],  # See commentary below.
    validate_args=True)

# Take 4 samples from the GP at the index points.
s = gp.sample(4, seed=random.PRNGKey(0))
assert s.shape == (4, num_points)

# Compute the log likelihood of the sampled values.
lp = gp.log_prob(s)
assert lp.shape == (4,)

# GPs can also be instantiated without index points, in which case the index
# points must be passed to method calls.
gp_no_index_pts = tfd.GaussianProcess(
    kernel,
    observation_noise_variance=observation_noise_variance)
s = gp_no_index_pts.sample(index_points=index_points, seed=random.PRNGKey(0))

# Predictive GPs conditioned on observations can be built with
# `GaussianProcess.posterior_predictive`. The Flax module's
# `precompute_predictive` and `posterior_predictive` methods call this GP method.
gprm = gp.posterior_predictive(
    observations=observations,
    predictive_index_points=predictive_index_points)

# `gprm` is an instance of `tfd.GaussianProcessRegressionModel`. This class can
# also be instantiated directly (as a side note -- this isn't necessary for
# modeling with Vizier).
same_gprm = tfd.GaussianProcessRegressionModel(
    kernel,
    index_points=predictive_index_points,
    observation_index_points=index_points,
    observations=observations,
    observation_noise_variance=observation_noise_variance)
```

Aside from the kernel, index points, and noise variance, there are two constructor args of `GaussianProcess` to be aware of:

- `always_yield_multivariate_normal`. By default, if there is only a single index point (`index_points` has shape `[1, d]`), then the Gaussian process has a univariate marginal distribution, so methods like `mean`, `stddev`, and `sample` will return a scalar. (If `index_points` has shape `[n, d]` the output of these methods will have shape `[n]`). To override the behavior for the univariate case and return arrays of shape `[1]` instead of scalars, set `always_yield_multivariate_normal=True`.
- `cholesky_fn` is a callable that takes a matrix and returns a Cholesky-like lower triangular factor. The default function adds a jitter of 1e-6 to the diagonal and then calls `jnp.linalg.cholesky`. An alternative, used in the Vizier GP, is `tfp.experimental.distributions.marginal_fns.retrying_cholesky`, which adds progressively larger jitter until the Cholesky decomposition succeeds.

A side note on batch shape in TFP

tl;dr: Don't worry about batch shape.

TFP objects have a notion of batch shape, which is useful for vectorized computations. For more on this, see Understanding TensorFlow Distributions Shapes.

For the purposes of Bayesopt in Vizier, JAX's `vmap` means that our TFP objects can have a single parameterization with empty batch shape. For example, in the following loss function takes a scalar `amplitude`, and the kernel and GP both have empty batch shape.

```
def loss_fn(amplitude):  # `a` is a scalar.
  k = tfpk.ExponentiatedQuadratic(amplitude=amplitude)  # batch shape []
  gp = tfd.GaussianProcess(k, index_points=index_points)   # batch shape []
  return -gp.log_prob(observations)

initial_amplitude = np.random.uniform(size=[50])

losses = jax.vmap(loss_fn)(initial_amplitude)
assert losses.shape == (50,)
```

We could also vectorize the loss computation by using a batched GP. In this simple case, the code is identical except that `vmap` is removed. Now, the kernel and GP represent a "batch" of kernels and GPs, each with different parameter values. Working with batch shape requires additional accounting on the part of the user to ensure that parameter shapes broadcast correctly, the correct dimensions are reduced over, etc. For Vizier's use case, we find it simpler to rely on `vmap`.

```
def loss_fn(amplitude):  # `a` has shape [50].
  k = tfpk.ExponentiatedQuadratic(amplitude=amplitude)  # batch shape [50]
  gp = tfd.GaussianProcess(k, index_points=index_points)   # batch shape [50]
  return -gp.log_prob(observations)

initial_amplitude = np.random.uniform(size=[50])

# No vmap.
losses = loss_fn(initial_amplitude)
assert losses.shape == (50,)
```

Bijectors

TFP bijectors represent (mostly) invertible, smooth functions. For Bayesopt modeling in Vizier, they are used to:

- to constrain parameter values for optimization in an unconstrained space.
- For input warping or output warping (e.g. the Yeo Johnson bijector).

Each bijector implements at least 3 methods:

- `forward`,
- `inverse`, and
- (at least) one of `forward_log_det_jacobian` and `inverse_log_det_jacobian`.

When bijectors are used to transform distributions (with `tfd.TransformedDistribution`), the log det Jacobian ensures that the transformation is volume-preserving and the distribution's PDF still integrates to 1.

Bijectors also cache the forward and inverse computations, and log-det-Jacobians. This has two purposes:

- Avoid repeating potentially expensive computations (as with the `CholeskyOuterProduct` bijector).
- Maintain numerical precision so that `b.inverse(b.forward(x)) == x`. Below is an illustration of preservation of numerical precision.

Although TFP library bijectors are written in TensorFlow (and automatically converted to JAX with TFP's rewrite machinery), user-defined bijectors can be written in JAX directly. For example, a complete JAX reimplementation of the `Exp` bijector is below. TFP's library already contains an `Exp` bijector and it's JAX supported, so it isn't actually necessary to implement this.

While it's rare that Vizier users will have to implement new TFP components, we include this as an example to show how it would be done using TFP's JAX backend, since all TFP library bijectors are written in TensorFlow.

Imports

```
from jax import numpy as jnp
import numpy as np
from tensorflow_probability.substrates import jax as tfp

tfd = tfp.distributions
tfb = tfp.bijectors
tfpk = tfp.math.psd_kernels
```

```
class Exp(tfb.AutoCompositeTensorBijector):

  def __init__(self,
               validate_args=False,
               name='exp'):
    """Instantiates the `Exp` bijector."""
    parameters = dict(locals())
    super(Exp, self).__init__(
        forward_min_event_ndims=0,
        validate_args=validate_args,
        parameters=parameters,  # TODO(emilyaf): explain why this is necessary.
        name=name)

  @classmethod
  def _parameter_properties(cls, dtype):
    return dict()

  @classmethod
  def _is_increasing(cls):
    return True

  def _forward(self, x):
      return jnp.exp(x)

  def _inverse(self, y):
      return jnp.log(y)

  def _inverse_log_det_jacobian(self, y):
    return -jnp.log(y)
```

(continues on next page)

(continued from previous page)

```
# Make sure it gives the same results as the TFP library bijector.
x = np.random.normal(size=[5])
tfp_exp = tfb.Exp()
my_exp = Exp()
np.testing.assert_allclose(tfp_exp.forward(x), my_exp.forward(x))
np.testing.assert_allclose(tfp_exp.forward_log_det_jacobian(x),
                           my_exp.forward_log_det_jacobian(x), rtol=1e-6)
```

TFP's bijector library includes:

- Simple bijectors (for example, there are many more):
  - `Scale(k)` multiplies its input by `k`.
  - `Shift(k)` adds `k` to its input.
  - `Sigmoid()` computes the sigmoid function.
  - `FillScaleTriL()` packs its input, a vector, into a lower-triangular matrix.
  - ...
- `Invert` wraps any bijector instance and swaps its forward and inverse methods, e.g. `inv_sigmoid = tfb.Invert(tfb.Sigmoid())`.
- `Chain` composes a series of bijectors. The function $f(x) = 3 + 2x$ can be expressed as `tfb.Chain([tfb.Shift(3.), tfb.Scale(2.)])`. Note that the bijectors in the list are applied from right to left.
- `JointMap` applies a nested structure of bijectors to an identical nested structure of inputs. `build_constraining_bijector`, shown above, returns a `JointMap` which applies a nested structure of bijectors to an identical nested structure of inputs. Vizier `get_constraints` function could be used to generate a `JointMap` based on the `Constraint`s of the `ModelParameter`s defined in the coroutine.
- `Restructure` packs the elements of one nested structure (e.g. a list) into a different structure (e.g. a dict). `spm.build_restructure_bijector`, for example, is a `Chain` bijector that takes a vector of parameters, splits it into a list, and packs the elements of the list into a dictionary with the same structure as the Flax parameters dict.

Exercise: Bijectors

Write a bijector (with `Chain`) that computes the function $f(x) = e^{x^2+1}$.

```
b = tfb.Chain([...])

f = lambda x: jnp.exp(x**2 + 1)
x = np.random.normal(size=[5])
np.testing.assert_allclose(f(x), b.forward(x))
```

Solution

```
b = tfb.Chain([tfb.Exp(), tfb.Shift(1.), tfb.Square()])
```

PSD kernels

TFP's PSD kernels compute positive semidefinite kernel functions. A PSD kernel instance is a required arg to TFP's Gaussian Process distribution, so specifying a GP model coroutine will generally involve defining a PSD kernel as an intermediate.

PSD kernel subclasses take hyperparameters, such as amplitude and length scale, as constructor args. They have three primary public methods: `apply`, `matrix`, and `tensor`, each of which computes the kernel function pairwise on inputs in different ways:

- `apply` computes the value of the kernel function at a pair of (batches of) input locations. It's the only required method for subclasses: `matrix` and `tensor` are implemented in terms of `apply` (except when a more efficient method exists to compure pairwise kernel matrices).
- `matrix` computes the value of the kernel *pairwise* on two (batches of) lists of input examples. When the two collections are the same the result is called the Gram matrix. `matrix` is the most important method for GPs.
- `tensor` generalizes `matrix`, taking rank `k1` and `k2` collections of input examples to a rank `k1 + k2` collection of kernel values. (We mention `tensor` for completeness, but it isn't relevant to GPs).

PSD kernels have somewhat complex shape semantics, due to the need to define which input dimensions should be included in pairwise computations and which should be treated as batch dimensions (denoting independent sets of input points.)

Imports

```
from jax import numpy as jnp
import numpy as np
from tensorflow_probability.python.internal import dtype_util
from tensorflow_probability.substrates import jax as tfp

tfd = tfp.distributions
tfpk = tfp.math.psd_kernels
```

Some examples of PSD kernel usage:

```
# Construct a MaternFiveHalves kernel (with empty batch shape).
amplitude = 2.
length_scale = 0.5
k = tfpk.MaternFiveHalves(
    amplitude=amplitude, length_scale=length_scale)

# Randomly sample some input data.
num_features = 5
num_observations = 12
x = np.random.normal(size=[num_observations, num_features])
```

(continues on next page)

(continued from previous page)

```
# `matrix` computes pairwise kernel values for the Cartesian product over the
# second-to-rightmost dimension of the inputs. Following the terminology in the
# PSD kernel docstring, there is a single example dimension (and single feature
# dimension).
assert k.matrix(x, x).shape == (12, 12)

# Calling `matrix` on inputs of shape [12, d] and [10, d] results in a kernel
# matrix of shape (12, 10)
y = np.random.normal(size=[10, num_features])
assert k.matrix(x, y).shape == (12, 10)
```

ARD kernels in TFP are implemented with the `FeatureScaled` kernel.

```
length_scale = np.random.uniform(size=[num_features])
ard_kernel = tfpk.FeatureScaled(
    tfpk.MaternFiveHalves(amplitude=np.float64(0.3)),
    scale_diag=length_scale)
```

Sums and products of PSD kernels are easy to compute, via operator overloading.

```
matern = tfpk.MaternFiveHalves(amplitude=2.)
squared_exponential = tfpk.ExponentiatedQuadratic(length_scale=0.1)
sum_kernel = matern + squared_exponential

np.testing.assert_allclose(
    sum_kernel.matrix(x, x),
    matern.matrix(x, x) + squared_exponential.matrix(x, x))
```

Exercise: Implemented a squared exponential kernel

As an exercise, try implementing a squared exponential PSD kernel:

```
k(x, y) = amplitude**2 * exp(-||x - y||**2 / (2 * length_scale**2))
```

In TFP library kernels (see TFP's squared exponential kernel), there are other details to consider, like handling of different dtypes, accepting either `length_scale` or `inverse_length_scale`, and ensuring that kernel batch shapes broadcast correctly with inputs.

For the purpose of the exercise we can ignore these, and `apply` can be written as a straightforward implementation of the kernel function. (New PSD kernels added to TFP would have to treat this more carefully, and existing kernels serve as good guides).

Try implementing `_apply` below (the solution is a couple cells down).

```
class MyExponentiatedQuadratic(tfpk.AutoCompositeTensorPsdKernel):

  def __init__(self,
               amplitude,
               length_scale):
    self.amplitude = amplitude
    self.length_scale = length_scale
    super(MyExponentiatedQuadratic, self).__init__(
```

(continues on next page)

(continued from previous page)

```
        feature_ndims=1,
        dtype=jnp.float32,
        name='MyExponentiatedQuadratic',
        validate_args=False)

  @classmethod
  def _parameter_properties(cls, dtype):
    # All TFP objects have parameter properties, which contain information on
    # the shape and domain of the parameters. The Softplus bijector is
    # associated with both the amplitude and length scale parameters, and may be
    # used to constrain these parameters to be positive. These bijectors are NOT
    # automatically applied when the kernel is called -- users may apply them
    # explicitly when doing unconstrained parameter optimization, e.g.
    return dict(
        amplitude=parameter_properties.ParameterProperties(
            default_constraining_bijector_fn=(
                lambda: tfb.Softplus(low=dtype_util.eps(dtype)))),
        length_scale=parameter_properties.ParameterProperties(
            default_constraining_bijector_fn=(
                lambda: tfb.Softplus(low=dtype_util.eps(dtype))))))

  def _apply(self, x1, x2, example_ndims=0):
    del example_ndims  # Can ignore this arg.
    pass
```

Make sure this kernel gives the same output as `ExponentiatedQuadratic` in the TFP library.

```
my_kernel = MyExponentiatedQuadratic(amplitude=2., length_scale=0.5)
tfp_kernel = tfpk.ExponentiatedQuadratic(amplitude=2., length_scale=0.5)
np.testing.assert_allclose(my_kernel.matrix(x, y), tfp_kernel.matrix(x, y), rtol=1e-5)
```

Solution

```
class MyExponentiatedQuadratic(tfpk.AutoCompositeTensorPsdKernel):

  def __init__(self,
               amplitude,
               length_scale):
    self.amplitude = amplitude
    self.length_scale = length_scale
    super(MyExponentiatedQuadratic, self).__init__(
        feature_ndims=1,
        dtype=jnp.float32,
        name='MyExponentiatedQuadratic',
        validate_args=False)

  @classmethod
  def _parameter_properties(cls, dtype):
    return dict(
        amplitude=parameter_properties.ParameterProperties(
```

(continues on next page)

(continued from previous page)

```
            default_constraining_bijector_fn=(
                lambda: tfb.Softplus(low=dtype_util.eps(dtype)))),
        length_scale=parameter_properties.ParameterProperties(
            default_constraining_bijector_fn=(
                lambda: tfb.Softplus(low=dtype_util.eps(dtype)))))

  def _apply(self, x1, x2, example_ndims=0):
    del example_ndims
    pairwise_sq_distance = jnp.sum((x1 - x2)**2, axis=-1)
    return jnp.exp(-0.5 * pairwise_sq_distance / self.length_scale ** 2) * self.
↪amplitude ** 2
```

Debugging tips

JAX

JAX's has a number of useful debugging tools including:

- `jax.debug.print` to print values, even inside of jit-compiled code.
- jit-able runtime error checking with `jax.experimental.checkify`.
- `jax_debug_nans` flag to automatically detect when NaNs are produced in jit-compiled code.
- `disable_jit`, a context manager that disables `jit()` behavior.

TFP

- TFP objects (bijectors, distributions, PSD kernels) have a `validate_args` boolean arg to `__init__`. If `True`, it runs additional (possibly expensive) runtime checks, e.g. to verify that parameters like `length_scale` are nonnegative. In TFP, we enable `validate_args` in unit tests, and use it as a debugging tool.
- Reproducibility: All functions and methods in TFP rely on random number generation, such as the `sample` method of distributions, take a `seed` arg, which in JAX is an instance of `jax.random.PRNGKey`. This arg is mandatory in TFP-on-JAX, and ensures reproducible random number generation. See the `jax.random` documentation for more details.
- Tests of sample statistics: TFP's internal `test_util` module includes `assertAllMeansClose`, which asserts that the mean of a sample is as expected, and diagnoses the statistical significance of failures.

```
#@title Imports
from jax import numpy as jnp, tree_util
from tensorflow_probability.substrates import jax as tfp

tfd = tfp.distributions
tfpk = tfp.math.psd_kernels
```

```
# Demo of `validate_args`.
print('Without runtime arg validation, the kernel with negative amplitude happily builds.
↪')
k = tfpk.MaternFiveHalves(amplitude=-1., validate_args=False)
```

```
print('With runtime arg validation:')
k = tfpk.MaternFiveHalves(amplitude=-1., validate_args=True)
```

What is "AutoCompositeTensor"?

You might have noticed that the base classes of the bijectors and PSD kernels are `AutoCompositeTensorBijector` and `AutoCompositeTensorPSDKernel`. In TensorFlow, objects that inherit from `CompositeTensor` have a recipe that allows them to be flattened into collections of Tensors and rebuilt, so that they can cross `tf.function` boundaries and interact with TF control flow similarly to Tensors (e.g., be passed in a `while_loop`'s carried state). JAX has a similar notion called Pytree. Subclassing the `AutoCompositeTensor*` versions of TFP base classes means that the class will be registered as a Pytree node (making use of shared CompositeTensor/Pytree machinery in TFP). For the Flax model to return a GP in JIT-compiled code, it's necessary for the GP and its PSD kernel to be Pytrees.

```
gp = tfd.GaussianProcess(
    tfpk.MaternFiveHalves(length_scale=jnp.ones([5])),
    observation_noise_variance=jnp.array([0.5]))
gp_flat, gp_tree = tree_util.tree_flatten(gp)
print(f'GP flattened into arrays: {gp_flat}')
rebuilt_gp = tree_util.tree_unflatten(gp_tree, gp_flat)
assert isinstance(rebuilt_gp, tfd.GaussianProcess)
```

3.2.2 PyGlove

OSS Vizier as a Backend

We demonstrate how OSS Vizier can be used as a distributed backend for PyGlove-based tuning tasks.

This assumes the user is already familiar with PyGlove primitives.

Installation and reference imports

```
!pip install google-vizier
!pip install pyglove
```

```
import multiprocessing
import multiprocessing.pool
import os

import pyglove as pg
from vizier import pyglove as pg_vizier
from vizier.service import vizier_server
```

Preliminaries

In the original PyGlove setting, one can normally perform evolutionary computation, for example:

```
search_space = pg.Dict(x=pg.floatv(0.0, 1.0), y=pg.floatv(0.0, 1.0))
algorithm = pg.evolution.regularized_evolution()
num_trials = 100


def evaluator(value: pg.Dict):
  return value.x**2 - value.y**2


for value, feedback in pg.sample(
    search_space,
    algorithm=algorithm,
    num_examples=num_trials,
    name='basic_run',
):
  reward = evaluator(value)
  feedback(reward=reward)
```

However, in many real-world scenarios, the evaluator may be much more expensive. For example, in neural architecture search applications, `evaluator` may be the result of an entire neural network training pipeline.

This leads to the need for a **backend**, in order to:

1. Distribute the evaluations over multiple workers.
2. Store the valuable results reliably and handle worker faults.

Initializing the OSS Vizier backend

The main initializer to call is `vizier.pyglove.init(...)`, **which should only be called once per process (not thread).** This function will edit global Python variables for determining values such as:

1. Prefix for study names.
2. Endpoint of the `VizierService` for storing data and handling requests.
3. Port for the `PythiaService` for computing suggestions.

In the local case, this can be called as-is:

```
pg_vizier.init('my_study')
```

Alternatively, if using a remote server, the endpoint can be specified as well:

```
server = vizier_server.DefaultVizierServer(host=hostname)  # Normally hosted on a remote␣
↪machine.
pg_vizier.init('my_study', vizier_endpoint=server.endpoint)
```

Parallelization

Due to the OSS Vizier backend, all workers may conveniently use exactly the same evaluation loop to work on a study:

```
num_workers = 10


def work_fn(worker_id):
  print(f"Worker ID: {worker_id}")
  for value, feedback in pg.sample(
      search_space,
      algorithm=algorithm,
      num_examples=num_trials // num_workers,
      name="worker_run",
  ):
    reward = evaluator(value)
    feedback(reward=reward)
```

There are three common forms of parallelization over the evaluation computation:

1. Multiple threads, single process.
2. Multiple processes, single machine.
3. Multiple machines.

Each of these cases defines the "worker", which can be a thread, process or machine respectively. We demonstrate examples of every type of parallelization below.

Multiple threads, single process

```
with multiprocessing.pool.ThreadPool(num_workers) as pool:
  pool.map(work_fn, range(num_workers))
```

Multiple processes, single machine

```
processes = []
for worker_id in range(num_workers):
  p = multiprocessing.Process(target=work_fn, args=(worker_id,))
  p.start()
  processes.append(p)

for p in processes:
  p.join()
```

Multiple machines

```
# Server Machine
server = vizier_server.DefaultVizierServer(host=hostname)
```

```
# Worker Machine
worker_id = os.uname()[1]
pg_vizier.init('my_study', vizier_endpoint=server.endpoint)
work_fn(worker_id)
```

3.3 API Reference

3.3.1 Code Structure

Frequently Used Import Targets

Includes a brief summary of important symbols and modules.

Service Users

If you write client code interacting with the OSS Vizier service, use these import targets:

- ``**from vizier.service import pyvizier as vz**``: Exposes the same set of symbol names as `vizier.pyvizier`. `vizier.service.pyvizier.Foo` is a subclass or an alias of `vizier.pyvizier.Foo`, and can be converted into protobufs.
- ``**from vizier.service import ...**``: Include binaries and internal utilities.

Algorithm Developers

If you write algorithm code (Designers or Pythia policies) in OSS Vizier, use these import targets:

- ``**from vizier import pyvizier as vz**``: Pure python building blocks of OSS Vizier. Cross-platform code, including Pythia policies, must use this `pyvizier` instance.
  - `Trial` and `ProblemStatement` are important classes.
- ``**from vizier.pyvizier import converters**``: Convert between `pyvizier` objects and numpy arrays.
  - `TrialToNumpyDict`: Converts parameters (and metrics) into a dict of numpy arrays. Preferred conversion method if you intended to train an embedding of categorical/discrete parameters, or data includes missing parameters or metrics.
  - `TrialToArrayConverter`: Converts parameters (and metrics) into an array.
- ``**from vizier.interfaces import serializable**``
  - `PartiallySerializable`, `Serializable`

Algorithm Abstractions

- **``from vizier import pythia``**
  - `Policy`, `PolicySupporter`: Key abstractions.
  - `LocalPolicyRunner`: Use it for running a `Policy` in RAM.
- **``from vizier import algorithms``**
  - `Designer`: Stateful algorithm abstraction.
  - `DesignerPolicy`: Wraps `Designer` into a Pythia Policy.
  - `GradientFreeMaximizer`: For optimizing acquisition functions.
  - `(Partially)SerializableDesigner`: Designers who wish to optimize performance by saving states.

Tensorflow Modules

- **``from vizier import tfp``**: Tensorflow-Probability utilities.
  - `acquisitions`: Acquisition functions module.
    - `AcquisitionFunction`: Abstraction.
    - `UpperConfidenceBound`, `ExpectedImprovement`, etc.
  - `bijectors`: Bijectors module.
    - `YeoJohnson`: Implements both Yeo-Johnson and Box-Cox transformations.
    - `optimal_power_transformation`: Returns the optimal power transformation.
    - `flip_sign`: returns a sign-flip bijector.
- **``from vizier import keras as vzk``**
  - `vzk.layers`: Layers usually wrapping `tfp` classes.
    - `variable_from_prior`: Utility layer for handling regularized variables.
  - `vzk.optim`: Wrappers around optimizers in `tfp`or `keras`.
  - `vzk.models`: Most of the useful models don't easily fit into Keras' `Model` abstraction, but we may add some for display.

3.4 Highlights

3.4.1 Applications

OSS Vizier is used in the following:

Codebases

- Vertex AI
- PyGlove
- OptFormer
- Init2winit
- Tensorflow Federated
- Tensorflow GNN
- CFU-Playground
- OpenML (Converter)

Guides

- Deep Learning Tuning Playbook

Papers

- Fishy: Layerwise Fisher Approximation for Higher-order Neural Network Optimization
- Massively Scaling Heteroscedastic Classifiers
- Towards Learning Universal Hyperparameter Optimizers with Transformers
- Task Selection for AutoML System Evaluation

3.4.2 Media

OSS Vizier has been featured in:

Articles

- Google Research, 2022 & Beyond: Algorithmic Advances
- MarkTechPost
- The Sequence
- ML News by Weights & Biases
- Analytics India Magazine
- This Week in AI by Lighting AI
- gHacks
- WebBigdata (Japanese)
- Random Access (Spanish)
- Electronic Smith
- Deep Learning Weekly

Videos/Talks

- AutoML Seminar 2023 Talk
- AutoML Conference 2022 Paper Presentation
- AutoML Conference 2022 AutoRL Tutorial
- ML News by Yannic Kilcher