Optuna Documentation

Release 3.6.0

Optuna Contributors.

Mar 18, 2024

CONTENTS:

Optuna is an automatic hyperparameter optimization software framework, particularly designed for machine learning. It features an imperative, *define-by-run* style user API. Thanks to our *define-by-run* API, the code written with Optuna enjoys high modularity, and the user of Optuna can dynamically construct the search spaces for the hyperparameters.

KEY FEATURES

Optuna has modern functionalities as follows:

- Lightweight, versatile, and platform agnostic architecture
    - Handle a wide variety of tasks with a simple installation that has few requirements.
- Pythonic search spaces
    - Define search spaces using familiar Python syntax including conditionals and loops.
- Efficient optimization algorithms
    - Adopt state-of-the-art algorithms for sampling hyperparameters and efficiently pruning unpromising trials.
- Easy parallelization
    - Scale studies to tens or hundreds of workers with little or no changes to the code.
- Quick visualization
    - Inspect optimization histories from a variety of plotting functions.

BASIC CONCEPTS

We use the terms *study* and *trial* as follows:

- Study: optimization based on an objective function

- Trial: a single execution of the objective function

Please refer to sample code below. The goal of a *study* is to find out the optimal set of hyperparameter values (e.g., `classifier` and `svm_c`) through multiple *trials* (e.g., `n_trials=100`). Optuna is a framework designed for the automation and the acceleration of the optimization *studies*.

CO Open in Colab

```python
import ...

# Define an objective function to be minimized.
def objective(trial):

    # Invoke suggest methods of a Trial object to generate hyperparameters.
    regressor_name = trial.suggest_categorical('classifier', ['SVR', 'RandomForest'])
    if regressor_name == 'SVR':
        svr_c = trial.suggest_float('svr_c', 1e-10, 1e10, log=True)
        regressor_obj = sklearn.svm.SVR(C=svr_c)
    else:
        rf_max_depth = trial.suggest_int('rf_max_depth', 2, 32)
        regressor_obj = sklearn.ensemble.RandomForestRegressor(max_depth=rf_max_depth)

    X, y = sklearn.datasets.fetch_california_housing(return_X_y=True)
    X_train, X_val, y_train, y_val = sklearn.model_selection.train_test_split(X, y,
→random_state=0)

    regressor_obj.fit(X_train, y_train)
    y_pred = regressor_obj.predict(X_val)

    error = sklearn.metrics.mean_squared_error(y_val, y_pred)

    return error  # An objective value linked with the Trial object.

study = optuna.create_study()  # Create a new study.
study.optimize(objective, n_trials=100)  # Invoke optimization of the objective function.
```
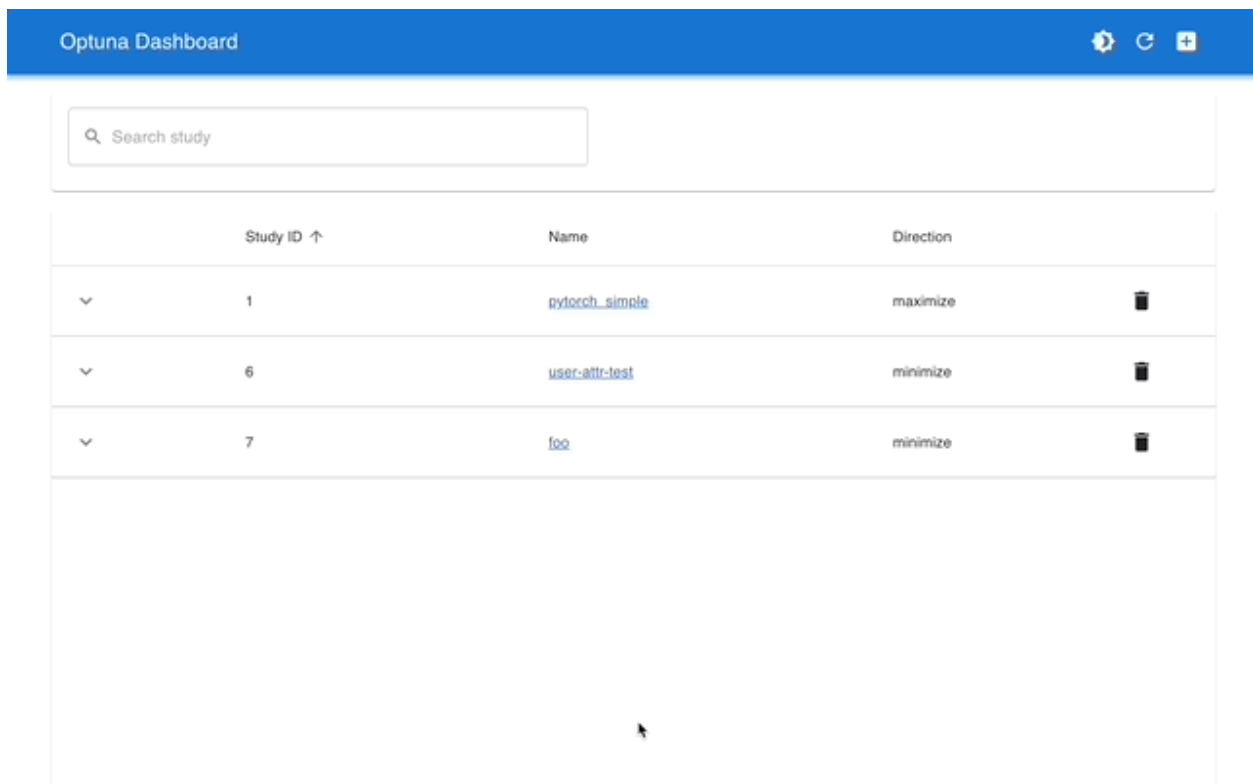
WEB DASHBOARD

Optuna Dashboard is a real-time web dashboard for Optuna. You can check the optimization history, hyperparameter importance, etc. in graphs and tables. You don't need to create a Python script to call Optuna's visualization functions. Feature requests and bug reports are welcome!

`optuna-dashboard` can be installed via pip:

```
$ pip install optuna-dashboard
```

Tip: Please check out the getting started section of Optuna Dashboard's official documentation.

COMMUNICATION

- GitHub Discussions for questions.

- GitHub Issues for bug reports and feature requests.

CONTRIBUTION

Any contributions to Optuna are welcome! When you send a pull request, please follow the contribution guide.

LICENSE

MIT License (see LICENSE).

Optuna uses the codes from SciPy and fdlibm projects (see Third-party License).

REFERENCE

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In KDD (arXiv).

7.1 Installation

Optuna supports Python 3.7 or newer.

We recommend to install Optuna via pip:

```
$ pip install optuna
```

You can also install the development version of Optuna from master branch of Git repository:

```
$ pip install git+https://github.com/optuna/optuna.git
```

You can also install Optuna via conda:

```
$ conda install -c conda-forge optuna
```

7.2 Tutorial

If you are new to Optuna or want a general introduction, we highly recommend the below video.

7.2.1 Key Features

Showcases Optuna's Key Features.

1. 10_key_features/001_first

2. 10_key_features/002_configurations

3. 10_key_features/003_efficient_optimization_algorithms

4. 10_key_features/004_distributed

5. 10_key_features/005_visualization

7.2.2 Recipes

Showcases the recipes that might help you using Optuna with comfort.

- 20_recipes/001_rdb
- 20_recipes/002_multi_objective
- 20_recipes/003_attributes
- 20_recipes/004_cli
- 20_recipes/005_user_defined_sampler
- 20_recipes/006_user_defined_pruner
- 20_recipes/007_optuna_callback
- 20_recipes/008_specify_params
- 20_recipes/009_ask_and_tell
- 20_recipes/010_reuse_best_trial
- 20_recipes/011_journal_storage
- Human-in-the-loop Optimization with Optuna Dashboard
- 20_recipes/012_artifact_tutorial
- 20_recipes/013_wilcoxon_pruner

7.3 API Reference

7.3.1 optuna

The `optuna` module is primarily used as an alias for basic Optuna functionality coded in other modules. Currently, two modules are aliased: (1) from `optuna.study`, functions regarding the Study lifecycle, and (2) from `optuna.exceptions`, the TrialPruned Exception raised when a trial is pruned.

`optuna.create_study`	Create a new `Study`.
`optuna.load_study`	Load the existing `Study` that has the specified name.
`optuna.delete_study`	Delete a `Study` object.
`optuna.copy_study`	Copy study from one storage to another.
`optuna.get_all_study_names`	Get all study names stored in a specified storage.
`optuna.get_all_study_summaries`	Get all history of studies stored in a specified storage.
`optuna.TrialPruned`	Exception for pruned trials.

optuna.create_study

optuna.**create_study**(**, storage=None, sampler=None, pruner=None, study_name=None, direction=None, load_if_exists=False, directions=None*)

Create a new *Study*.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)
```

> **Parameters**
>
> - **storage** (*str | storages.BaseStorage | None*) – Database URL. If this argument is set to None, in-memory storage is used, and the *Study* will not be persistent.
>
>   ---
>   **Note:**
>
>   When a database URL is passed, Optuna internally uses SQLAlchemy to handle the database. Please refer to SQLAlchemy's document for further details. If you want to specify non-default options to SQLAlchemy Engine, you can instantiate *RDBStorage* with your desired options and pass it to the storage argument instead of a URL.
>
>   ---
>
> - **sampler** (*'samplers.BaseSampler' | None*) – A sampler object that implements background algorithm for value suggestion. If None is specified, *TPESampler* is used during single-objective optimization and *NSGAIISampler* during multi-objective optimization. See also *samplers*.
>
> - **pruner** (*pruners.BasePruner | None*) – A pruner object that decides early stopping of unpromising trials. If None is specified, *MedianPruner* is used as the default. See also *pruners*.
>
> - **study_name** (*str | None*) – Study's name. If this argument is set to None, a unique name is generated automatically.
>
> - **direction** (*str | StudyDirection | None*) – Direction of optimization. Set minimize for minimization and maximize for maximization. You can also pass the corresponding *StudyDirection* object. direction and directions must not be specified at the same time.
>
>   ---
>   **Note:** If none of *direction* and *directions* are specified, the direction of the study is set to "minimize".
>
>   ---
>
> - **load_if_exists** (*bool*) – Flag to control the behavior to handle a conflict of study names. In the case where a study named study_name already exists in the storage, a

DuplicatedStudyError is raised if `load_if_exists` is set to `False`. Otherwise, the creation of the study is skipped, and the existing one is returned.

- **directions** (*Sequence[str | StudyDirection] | None*) – A sequence of directions during multi-objective optimization. `direction` and `directions` must not be specified at the same time.

Returns

A *Study* object.

Return type

Study

See also:

optuna.create_study() is an alias of *optuna.study.create_study()*.

See also:

The rdb tutorial provides concrete examples to save and resume optimization using RDB.

optuna.load_study

optuna.**load_study**(***, *study_name*, *storage*, *sampler=None*, *pruner=None*)

Load the existing *Study* that has the specified name.

Example

```
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study(storage="sqlite:///example.db", study_name="my_study")
study.optimize(objective, n_trials=3)

loaded_study = optuna.load_study(study_name="my_study", storage="sqlite:///example.
↪db")
assert len(loaded_study.trials) == len(study.trials)
```

Parameters

- **study_name** (*str | None*) – Study's name. Each study has a unique name as an identifier. If `None`, checks whether the storage contains a single study, and if so loads that study. `study_name` is required if there are multiple studies in the storage.

- **storage** (*str | storages.BaseStorage*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.

- **sampler** (*'samplers.BaseSampler' | None*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *TPESampler* is used as the default. See also *samplers*.

- **pruner** (`pruners.BasePruner` | *None*) – A pruner object that decides early stopping of unpromising trials. If `None` is specified, *MedianPruner* is used as the default. See also *pruners*.

> **Return type**
> > *Study*

See also:

`optuna.load_study()` is an alias of `optuna.study.load_study()`.

optuna.delete_study

optuna.**delete_study**(**, study_name, storage*)

> Delete a *Study* object.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(study_name="example-study", storage="sqlite:///example.
→db")
study.optimize(objective, n_trials=3)

optuna.delete_study(study_name="example-study", storage="sqlite:///example.db")
```

> **Parameters**
> - **study_name** (`str`) – Study's name.
> - **storage** (`str` | *BaseStorage*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.
>
> **Return type**
> > None

See also:

`optuna.delete_study()` is an alias of `optuna.study.delete_study()`.

optuna.copy_study

optuna.**copy_study**(*, *from_study_name*, *from_storage*, *to_storage*, *to_study_name=None*)

Copy study from one storage to another.

The direction(s) of the objective(s) in the study, trials, user attributes and system attributes are copied.

Note: *copy_study()* copies a study even if the optimization is working on. It means users will get a copied study that contains a trial that is not finished.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(
    study_name="example-study",
    storage="sqlite:///example.db",
)
study.optimize(objective, n_trials=3)

optuna.copy_study(
    from_study_name="example-study",
    from_storage="sqlite:///example.db",
    to_storage="sqlite:///example_copy.db",
)

study = optuna.load_study(
    study_name=None,
    storage="sqlite:///example_copy.db",
)
```

Parameters

- **from_study_name** (*str*) – Name of study.

- **from_storage** (*str | BaseStorage*) – Source database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.

- **to_storage** (*str | BaseStorage*) – Destination database URL.

- **to_study_name** (*str | None*) – Name of the created study. If omitted, `from_study_name` is used.

Raises

DuplicatedStudyError – If a study with a conflicting name already exists in the destination storage.

Return type
    None

optuna.get_all_study_names

optuna.**get_all_study_names**(*storage*)

Get all study names stored in a specified storage.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(study_name="example-study", storage="sqlite:///example.
↪db")
study.optimize(objective, n_trials=3)

study_names = optuna.study.get_all_study_names(storage="sqlite:///example.db")
assert len(study_names) == 1

assert study_names[0] == "example-study"
```

Parameters
    storage (*str | BaseStorage*) – Database URL such as `sqlite:///example.db`. Please
    see also the documentation of *create_study()* for further details.

Returns
    List of all study names in the storage.

Return type
    list[str]

See also:

optuna.get_all_study_names() is an alias of *optuna.study.get_all_study_names()*.

optuna.get_all_study_summaries

optuna.**get_all_study_summaries**(*storage*, *include_best_trial=True*)

Get all history of studies stored in a specified storage.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(study_name="example-study", storage="sqlite:///example.
↪db")
study.optimize(objective, n_trials=3)

study_summaries = optuna.study.get_all_study_summaries(storage="sqlite:///example.db
↪")
assert len(study_summaries) == 1

study_summary = study_summaries[0]
assert study_summary.study_name == "example-study"
```

Parameters

- **storage** (*str | BaseStorage*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.

- **include_best_trial** (*bool*) – Include the best trials if exist. It potentially increases the number of queries and may take longer to fetch summaries depending on the storage.

Returns

List of study history summarized as *StudySummary* objects.

Return type

list[StudySummary]

See also:

optuna.get_all_study_summaries() is an alias of *optuna.study.get_all_study_summaries()*.

optuna.TrialPruned

exception optuna.**TrialPruned**

Exception for pruned trials.

This error tells a trainer that the current *Trial* was pruned. It is supposed to be raised after *optuna.trial.Trial.should_prune()* as shown in the following example.

See also:

optuna.TrialPruned is an alias of *optuna.exceptions.TrialPruned*.

Example

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)
```

add_note()

> Exception.add_note(note) – add a note to the exception

7.3.2 optuna.artifacts

The *artifacts* module provides the way to manage artifacts (output files) in Optuna.

class optuna.artifacts.**FileSystemArtifactStore**(*base_path*)

> An artifact store for file systems.
>
> > **Parameters**
> >
> > > **base_path** (*str | Path*) – The base path to a directory to store artifacts.

Example

```python
import os

import optuna
from optuna.artifacts import FileSystemArtifactStore
from optuna.artifacts import upload_artifact


base_path = "./artifacts"
os.makedirs(base_path, exist_ok=True)
artifact_store = FileSystemArtifactStore(base_path=base_path)


def objective(trial: optuna.Trial) -> float:
    ... = trial.suggest_float("x", -10, 10)
    file_path = generate_example(...)
    upload_artifact(trial, file_path, artifact_store)
    return ...
```

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

class optuna.artifacts.**Boto3ArtifactStore**(*bucket_name*, *client=None*, *, *avoid_buf_copy=False*)

>An artifact backend for Boto3.

>>**Parameters**

>>>- **bucket_name** (*str*) – The name of the bucket to store artifacts.

>>>- **client** (*S3Client | None*) – A Boto3 client to use for storage operations. If not specified, a new client will be created.

>>>- **avoid_buf_copy** (*bool*) – If True, skip procedure to copy the content of the source file object to a buffer before uploading it to S3 ins. This is default to False because using upload_fileobj() method of Boto3 client might close the source file object.

>**Example**

```python
import optuna
from optuna.artifacts import upload_artifact
from optuna.artifacts import Boto3ArtifactStore


artifact_store = Boto3ArtifactStore("my-bucket")


def objective(trial: optuna.Trial) -> float:
    ... = trial.suggest_float("x", -10, 10)
    file_path = generate_example(...)
    upload_artifact(trial, file_path, artifact_store)
    return ...
```

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

---

class optuna.artifacts.**GCSArtifactStore**(*bucket_name*, *client=None*)

An artifact backend for Google Cloud Storage (GCS).

> **Parameters**
>
> - **bucket_name** (*str*) – The name of the bucket to store artifacts.
>
> - **client** (*google.cloud.storage.Client | None*) – A google-cloud-storage *Client* to use for storage operations. If not specified, a new client will be created with default settings.

Example

```python
import optuna
from optuna.artifacts import GCSArtifactStore, upload_artifact


artifact_backend = GCSArtifactStore("my-bucket")


def objective(trial: optuna.Trial) -> float:
    ... = trial.suggest_float("x", -10, 10)
    file_path = generate_example(...)
    upload_artifact(trial, file_path, artifact_backend)
    return ...
```

Before running this code, you will have to install *gcloud* and run

```
gcloud auth application-default login
```

so that the Cloud Storage library can automatically find the credential.

Note: Added in v3.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.4.0.

---

class optuna.artifacts.**Backoff**(*backend*, ***, *max_retries=10*, *multiplier=2*, *min_delay=0.1*, *max_delay=30*)

An artifact store's middleware for exponential backoff.

Example

```python
import optuna
from optuna.artifacts import upload_artifact
from optuna.artifacts import Boto3ArtifactStore
from optuna.artifacts import Backoff


artifact_store = Backoff(Boto3ArtifactStore("my-bucket"))
```

```python
def objective(trial: optuna.Trial) -> float:
    ... = trial.suggest_float("x", -10, 10)
    file_path = generate_example(...)
    upload_artifact(trial, file_path, artifact_store)
    return ...
```

> **Parameters**
>
> - **backend** (*ArtifactStore*) –
>
> - **max_retries** (*int*) –
>
> - **multiplier** (*float*) –
>
> - **min_delay** (*float*) –
>
> - **max_delay** (*float*) –

optuna.artifacts.**upload_artifact**(*study_or_trial*, *file_path*, *artifact_store*, *, *storage=None*, *mimetype=None*, *encoding=None*)

> Upload an artifact to the artifact store.
>
> **Parameters**
>
> - **study_or_trial** (*Trial | FrozenTrial | Study*) – A *Trial* object, a *FrozenTrial*, or a *Study* object.
>
> - **file_path** (*str*) – A path to the file to be uploaded.
>
> - **artifact_store** (*ArtifactStore*) – An artifact store.
>
> - **storage** (*BaseStorage | None*) – A storage object. If trial is not a *Trial* object, this argument is required.
>
> - **mimetype** (*str | None*) – A MIME type of the artifact. If not specified, the MIME type is guessed from the file extension.
>
> - **encoding** (*str | None*) – An encoding of the artifact, which is suitable for use as a `Content-Encoding` header (e.g. gzip). If not specified, the encoding is guessed from the file extension.
>
> **Returns**
> > An artifact ID.
>
> **Return type**
> > str

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

7.3.3 optuna.cli

The `cli` module implements Optuna's command-line functionality.

For detail, please see the result of

```
$ optuna --help
```

See also:

The cli tutorial provides use-cases with examples.

7.3.4 optuna.distributions

The `distributions` module defines various classes representing probability distributions, mainly used to suggest initial hyperparameter values for an optimization trial. Distribution classes inherit from a library-internal `BaseDistribution`, and is initialized with specific parameters, such as the `low` and `high` endpoints for a `IntDistribution`.

Optuna users should not use distribution classes directly, but instead use utility functions provided by `Trial` such as `suggest_int()`.

`optuna.distributions.FloatDistribution`	A distribution on floats.
`optuna.distributions.IntDistribution`	A distribution on integers.
`optuna.distributions.UniformDistribution`	A uniform distribution in the linear domain.
`optuna.distributions.` `LogUniformDistribution`	A uniform distribution in the log domain.
`optuna.distributions.` `DiscreteUniformDistribution`	A discretized uniform distribution in the linear domain.
`optuna.distributions.` `IntUniformDistribution`	A uniform distribution on integers.
`optuna.distributions.` `IntLogUniformDistribution`	A uniform distribution on integers in the log domain.
`optuna.distributions.` `CategoricalDistribution`	A categorical distribution.
`optuna.distributions.distribution_to_json`	Serialize a distribution to JSON format.
`optuna.distributions.json_to_distribution`	Deserialize a distribution in JSON format.
`optuna.distributions.` `check_distribution_compatibility`	A function to check compatibility of two distributions.

optuna.distributions.FloatDistribution

class optuna.distributions.**FloatDistribution**(*low*, *high*, *log=False*, *step=None*)

A distribution on floats.

This object is instantiated by `suggest_float()`, and passed to `samplers` in general.

Note: When `step` is not `None`, if the range $[\text{low}, \text{high}]$ is not divisible by step, high will be replaced with the maximum of $k \times \text{step} + \text{low} < \text{high}$, where k is an integer.

Parameters

- **low** (*float*) –
- **high** (*float*) –
- **log** (*bool*) –
- **step** (*None | float*) –

low

Lower endpoint of the range of the distribution. `low` is included in the range. `low` must be less than or equal to `high`. If `log` is `True`, `low` must be larger than 0.

high

Upper endpoint of the range of the distribution. `high` is included in the range. `high` must be greater than or equal to `low`.

log

If `log` is `True`, this distribution is in log-scaled domain. In this case, all parameters enqueued to the distribution must be positive values. This parameter must be `False` when the parameter `step` is not `None`.

step

A discretization step. `step` must be larger than 0. This parameter must be `None` when the parameter `log` is `True`.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

Test whether the range of this distribution contains just a single value.

> **Returns**
>
> `True` if the range of this distribution contains just a single value, otherwise `False`.
>
> **Return type**
>
> bool

to_external_repr(*param_value_in_internal_repr*)

Convert internal representation of a parameter value into external representation.

> **Parameters**
>
> **param_value_in_internal_repr** (*float*) – Optuna's internal representation of a parameter value.
>
> **Returns**
>
> Optuna's external representation of a parameter value.
>
> **Return type**
>
> *Any*

to_internal_repr(*param_value_in_external_repr*)

    Convert external representation of a parameter value into internal representation.

    Parameters

        param_value_in_external_repr (*float*) – Optuna's external representation of a parameter value.

    Returns

        Optuna's internal representation of a parameter value.

    Return type

        float

optuna.distributions.IntDistribution

class optuna.distributions.**IntDistribution**(*low*, *high*, *log=False*, *step=1*)

    A distribution on integers.

    This object is instantiated by *suggest_int()*, and passed to *samplers* in general.

    Note: When step is not None, if the range $[\text{low}, \text{high}]$ is not divisible by step, high will be replaced with the maximum of $k \times \text{step} + \text{low} < \text{high}$, where k is an integer.

    Parameters

        • **low** (*int*) –

        • **high** (*int*) –

        • **log** (*bool*) –

        • **step** (*int*) –

low

    Lower endpoint of the range of the distribution. low is included in the range. low must be less than or equal to high. If log is True, low must be larger than or equal to 1.

high

    Upper endpoint of the range of the distribution. high is included in the range. high must be greater than or equal to low.

log

    If log is True, this distribution is in log-scaled domain. In this case, all parameters enqueued to the distribution must be positive values. This parameter must be False when the parameter step is not 1.

step

    A discretization step. step must be a positive integer. This parameter must be 1 when the parameter log is True.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

> Test whether the range of this distribution contains just a single value.
>
> > **Returns**
> >> True if the range of this distribution contains just a single value, otherwise False.
> >
> > **Return type**
> >> bool

to_external_repr(*param_value_in_internal_repr*)

> Convert internal representation of a parameter value into external representation.
>
> > **Parameters**
> >> **param_value_in_internal_repr** (*float*) – Optuna's internal representation of a parameter value.
> >
> > **Returns**
> >> Optuna's external representation of a parameter value.
> >
> > **Return type**
> >> int

to_internal_repr(*param_value_in_external_repr*)

> Convert external representation of a parameter value into internal representation.
>
> > **Parameters**
> >> **param_value_in_external_repr** (*int*) – Optuna's external representation of a parameter value.
> >
> > **Returns**
> >> Optuna's internal representation of a parameter value.
> >
> > **Return type**
> >> float

optuna.distributions.UniformDistribution

class optuna.distributions.**UniformDistribution**(*low*, *high*)

> A uniform distribution in the linear domain.
>
> This object is instantiated by *suggest_float()*, and passed to *samplers* in general.
>
> > **Parameters**
> >> - **low** (*float*) –
> >> - **high** (*float*) –

low

> Lower endpoint of the range of the distribution. `low` is included in the range. `low` must be less than or equal to `high`.

high

> Upper endpoint of the range of the distribution. `high` is included in the range. `high` must be greater than or equal to `low`.

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use *FloatDistribution* instead.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

> Test whether the range of this distribution contains just a single value.
>
> > **Returns**
> >   `True` if the range of this distribution contains just a single value, otherwise `False`.
> >
> > **Return type**
> >   bool

to_external_repr(*param_value_in_internal_repr*)

> Convert internal representation of a parameter value into external representation.
>
> > **Parameters**
> >   **param_value_in_internal_repr** (*float*) – Optuna's internal representation of a parameter value.
> >
> > **Returns**
> >   Optuna's external representation of a parameter value.
> >
> > **Return type**
> >   *Any*

to_internal_repr(*param_value_in_external_repr*)

> Convert external representation of a parameter value into internal representation.
>
> > **Parameters**
> >   **param_value_in_external_repr** (*float*) – Optuna's external representation of a parameter value.
> >
> > **Returns**
> >   Optuna's internal representation of a parameter value.

> **Return type**
>> float

optuna.distributions.LogUniformDistribution

class optuna.distributions.**LogUniformDistribution**(*low*, *high*)

 A uniform distribution in the log domain.

 This object is instantiated by *suggest_float()* with `log=True`, and passed to *samplers* in general.

> **Parameters**
>> - **low** (*float*) –
>> - **high** (*float*) –

low

 Lower endpoint of the range of the distribution. `low` is included in the range. `low` must be larger than 0. `low` must be less than or equal to `high`.

high

 Upper endpoint of the range of the distribution. `high` is included in the range. `high` must be greater than or equal to `low`.

Warning: Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Use *FloatDistribution* instead.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

 Test whether the range of this distribution contains just a single value.

> **Returns**
>> True if the range of this distribution contains just a single value, otherwise False.

> **Return type**
>> bool

to_external_repr(*param_value_in_internal_repr*)

 Convert internal representation of a parameter value into external representation.

> **Parameters**
>> **param_value_in_internal_repr** (*float*) – Optuna's internal representation of a parameter value.

> **Returns**
>> Optuna's external representation of a parameter value.
>
> **Return type**
>> *Any*

to_internal_repr(*param_value_in_external_repr*)

> Convert external representation of a parameter value into internal representation.
>
> **Parameters**
>> **param_value_in_external_repr** (*float*) – Optuna's external representation of a parameter value.
>
> **Returns**
>> Optuna's internal representation of a parameter value.
>
> **Return type**
>> float

optuna.distributions.DiscreteUniformDistribution

class optuna.distributions.**DiscreteUniformDistribution**(*low*, *high*, *q*)

> A discretized uniform distribution in the linear domain.
>
> This object is instantiated by *suggest_float()* with `step` argument, and passed to *samplers* in general.

Note: If the range $[\text{low}, \text{high}]$ is not divisible by q, high will be replaced with the maximum of $kq + \text{low} < \text{high}$, where k is an integer.

> **Parameters**
>> - **low** (*float*) – Lower endpoint of the range of the distribution. `low` is included in the range. `low` must be less than or equal to `high`.
>>
>> - **high** (*float*) – Upper endpoint of the range of the distribution. `high` is included in the range. `high` must be greater than or equal to `low`.
>>
>> - **q** (*float*) – A discretization step. `q` must be larger than 0.

low

> Lower endpoint of the range of the distribution. `low` is included in the range.

high

> Upper endpoint of the range of the distribution. `high` is included in the range.

Warning: Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Use *FloatDistribution* instead.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

Attributes

q	Discretization step.

property q: `float`

   Discretization step.

   DiscreteUniformDistribution is a subtype of *FloatDistribution*. This property is a proxy for its `step` attribute.

single()

   Test whether the range of this distribution contains just a single value.

   Returns
      `True` if the range of this distribution contains just a single value, otherwise `False`.

   Return type
      bool

to_external_repr(*param_value_in_internal_repr*)

   Convert internal representation of a parameter value into external representation.

   Parameters
      param_value_in_internal_repr (*float*) – Optuna's internal representation of a parameter value.

   Returns
      Optuna's external representation of a parameter value.

   Return type
      Any

to_internal_repr(*param_value_in_external_repr*)

   Convert external representation of a parameter value into internal representation.

   Parameters
      param_value_in_external_repr (*float*) – Optuna's external representation of a parameter value.

   Returns
      Optuna's internal representation of a parameter value.

   Return type
      float

optuna.distributions.IntUniformDistribution

class optuna.distributions.**IntUniformDistribution**(*low*, *high*, *step=1*)

A uniform distribution on integers.

This object is instantiated by *suggest_int()*, and passed to *samplers* in general.

Note: If the range [low, high] is not divisible by step, high will be replaced with the maximum of $k \times \text{step} + \text{low} <$ high, where k is an integer.

> **Parameters**
> - **low** (*int*) –
> - **high** (*int*) –
> - **step** (*int*) –

low

Lower endpoint of the range of the distribution. `low` is included in the range. `low` must be less than or equal to `high`.

high

Upper endpoint of the range of the distribution. `high` is included in the range. `high` must be greater than or equal to `low`.

step

A discretization step. `step` must be a positive integer.

Warning: Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Use *IntDistribution* instead.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

Test whether the range of this distribution contains just a single value.

> **Returns**
> True if the range of this distribution contains just a single value, otherwise False.

> **Return type**
> bool

to_external_repr(*param_value_in_internal_repr*)

Convert internal representation of a parameter value into external representation.

> **Parameters**
>> **param_value_in_internal_repr** ([*float*](#)) – Optuna's internal representation of a parameter value.
>
> **Returns**
>> Optuna's external representation of a parameter value.
>
> **Return type**
>> int

to_internal_repr(*param_value_in_external_repr*)

Convert external representation of a parameter value into internal representation.

> **Parameters**
>> **param_value_in_external_repr** ([*int*](#)) – Optuna's external representation of a parameter value.
>
> **Returns**
>> Optuna's internal representation of a parameter value.
>
> **Return type**
>> float

optuna.distributions.IntLogUniformDistribution

class optuna.distributions.**IntLogUniformDistribution**(*low*, *high*, *step=1*)

A uniform distribution on integers in the log domain.

This object is instantiated by [*suggest_int()*](#), and passed to [*samplers*](#) in general.

> **Parameters**
>> - **low** ([*int*](#)) –
>> - **high** ([*int*](#)) –
>> - **step** ([*int*](#)) –

low

> Lower endpoint of the range of the distribution. `low` is included in the range and must be larger than or equal to 1. `low` must be less than or equal to `high`.

high

> Upper endpoint of the range of the distribution. `high` is included in the range. `high` must be greater than or equal to `low`.

step

> A discretization step. `step` must be a positive integer.

> **Warning:** Deprecated in v2.0.0. `step` argument will be removed in the future. The removal of this feature is currently scheduled for v4.0.0, but this schedule is subject to change.
>
> Samplers and other components in Optuna relying on this distribution will ignore this value and assume that `step` is always 1. User-defined samplers may continue to use other values besides 1 during the deprecation.

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is
> currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/
> releases/tag/v3.0.0.
>
> Use *IntDistribution* instead.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

Test whether the range of this distribution contains just a single value.

> **Returns**
>
> True if the range of this distribution contains just a single value, otherwise False.
>
> **Return type**
>
> bool

to_external_repr(*param_value_in_internal_repr*)

Convert internal representation of a parameter value into external representation.

> **Parameters**
>
> **param_value_in_internal_repr** (*float*) – Optuna's internal representation of a parameter value.
>
> **Returns**
>
> Optuna's external representation of a parameter value.
>
> **Return type**
>
> int

to_internal_repr(*param_value_in_external_repr*)

Convert external representation of a parameter value into internal representation.

> **Parameters**
>
> **param_value_in_external_repr** (*int*) – Optuna's external representation of a parameter value.
>
> **Returns**
>
> Optuna's internal representation of a parameter value.
>
> **Return type**
>
> float

optuna.distributions.CategoricalDistribution

class optuna.distributions.**CategoricalDistribution**(*choices*)

> A categorical distribution.
>
> This object is instantiated by *suggest_categorical()*, and passed to *samplers* in general.
>
> > **Parameters**
> > > **choices** (*Sequence[None | bool | int | float | str]*) – Parameter value candidates.
> > > choices must have one element at least.

Note: Not all types are guaranteed to be compatible with all storages. It is recommended to restrict the types of the choices to None, bool, int, float and str.

> **choices**
>
> > Parameter value candidates.

Methods

single()	Test whether the range of this distribution contains just a single value.
to_external_repr(param_value_in_internal_repr)	Convert internal representation of a parameter value into external representation.
to_internal_repr(param_value_in_external_repr)	Convert external representation of a parameter value into internal representation.

single()

> Test whether the range of this distribution contains just a single value.
>
> > **Returns**
> > > True if the range of this distribution contains just a single value, otherwise False.
> >
> > **Return type**
> > > bool

to_external_repr(*param_value_in_internal_repr*)

> Convert internal representation of a parameter value into external representation.
>
> > **Parameters**
> > > **param_value_in_internal_repr** (*float*) – Optuna's internal representation of a parameter value.
> >
> > **Returns**
> > > Optuna's external representation of a parameter value.
> >
> > **Return type**
> > > None | bool | int | float | str

to_internal_repr(*param_value_in_external_repr*)

> Convert external representation of a parameter value into internal representation.
>
> > **Parameters**
> > > **param_value_in_external_repr** (*None | bool | int | float | str*) – Optuna's external representation of a parameter value.

Returns
 Optuna's internal representation of a parameter value.

Return type
 float

optuna.distributions.distribution_to_json

optuna.distributions.**distribution_to_json**(*dist*)

 Serialize a distribution to JSON format.

 Parameters
  dist (`BaseDistribution`) – A distribution to be serialized.

 Returns
  A JSON string of a given distribution.

 Return type
  str

optuna.distributions.json_to_distribution

optuna.distributions.**json_to_distribution**(*json_str*)

 Deserialize a distribution in JSON format.

 Parameters
  json_str (`str`) – A JSON-serialized distribution.

 Returns
  A deserialized distribution.

 Return type
  BaseDistribution

optuna.distributions.check_distribution_compatibility

optuna.distributions.**check_distribution_compatibility**(*dist_old*, *dist_new*)

 A function to check compatibility of two distributions.

 It checks whether `dist_old` and `dist_new` are the same kind of distributions. If `dist_old` is *CategoricalDistribution*, it further checks `choices` are the same between `dist_old` and `dist_new`. Note that this method is not supposed to be called by library users.

 Parameters

 • **dist_old** (`BaseDistribution`) – A distribution previously recorded in storage.

 • **dist_new** (`BaseDistribution`) – A distribution newly added to storage.

 Return type
  None

7.3.5 optuna.exceptions

The *exceptions* module defines Optuna-specific exceptions deriving from a base *OptunaError* class. Of special importance for library users is the *TrialPruned* exception to be raised if *optuna.trial.Trial.should_prune()* returns True for a trial that should be pruned.

optuna.exceptions.OptunaError	Base class for Optuna specific errors.
optuna.exceptions.TrialPruned	Exception for pruned trials.
optuna.exceptions.CLIUsageError	Exception for CLI.
optuna.exceptions.StorageInternalError	Exception for storage operation.
optuna.exceptions.DuplicatedStudyError	Exception for a duplicated study name.

optuna.exceptions.OptunaError

exception optuna.exceptions.**OptunaError**

Base class for Optuna specific errors.

add_note()

Exception.add_note(note) – add a note to the exception

optuna.exceptions.TrialPruned

exception optuna.exceptions.**TrialPruned**

Exception for pruned trials.

This error tells a trainer that the current *Trial* was pruned. It is supposed to be raised after *optuna.trial.Trial.should_prune()* as shown in the following example.

See also:

optuna.TrialPruned is an alias of *optuna.exceptions.TrialPruned*.

Example

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100
```

```
    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)
```

> **add_note()**
> > Exception.add_note(note) – add a note to the exception

optuna.exceptions.CLIUsageError

exception optuna.exceptions.**CLIUsageError**

> Exception for CLI.
>
> CLI raises this exception when it receives invalid configuration.
>
> **add_note()**
> > Exception.add_note(note) – add a note to the exception

optuna.exceptions.StorageInternalError

exception optuna.exceptions.**StorageInternalError**

> Exception for storage operation.
>
> This error is raised when an operation failed in backend DB of storage.
>
> **add_note()**
> > Exception.add_note(note) – add a note to the exception

optuna.exceptions.DuplicatedStudyError

exception optuna.exceptions.**DuplicatedStudyError**

> Exception for a duplicated study name.
>
> This error is raised when a specified study name already exists in the storage.
>
> **add_note()**
> > Exception.add_note(note) – add a note to the exception

7.3.6 optuna.importance

The *importance* module provides functionality for evaluating hyperparameter importances based on completed trials in a given study. The utility function *get_param_importances()* takes a *Study* and optional evaluator as two of its inputs. The evaluator must derive from `BaseImportanceEvaluator`, and is initialized as a *FanovaImportanceEvaluator* by default when not passed in. Users implementing custom evaluators should refer to either *FanovaImportanceEvaluator*, *MeanDecreaseImpurityImportanceEvaluator*, or *PedAnovaImportanceEvaluator* as a guide, paying close attention to the format of the return value from the Evaluator's `evaluate` function.

Note: *FanovaImportanceEvaluator* takes over 1 minute when given a study that contains 1000+ trials. We published optuna-fast-fanova library, that is a Cython accelerated fANOVA implementation. By using it, you can get hyperparameter importances within a few seconds. If `n_trials` is more than 10000, the Cython implementation takes more than a minute, so you can use *PedAnovaImportanceEvaluator* instead, enabling the evaluation to finish in a second.

`optuna.importance.get_param_importances`	Evaluate parameter importances based on completed trials in the given study.
`optuna.importance.FanovaImportanceEvaluator`	fANOVA importance evaluator.
`optuna.importance.MeanDecreaseImpurityImportanceEvaluator`	Mean Decrease Impurity (MDI) parameter importance evaluator.
`optuna.importance.PedAnovaImportanceEvaluator`	PED-ANOVA importance evaluator.

optuna.importance.get_param_importances

optuna.importance.**get_param_importances**(*study*, ***, *evaluator=None*, *params=None*, *target=None*, *normalize=True*)

Evaluate parameter importances based on completed trials in the given study.

The parameter importances are returned as a dictionary where the keys consist of parameter names and their values importances. The importances are represented by non-negative floating point numbers, where higher values mean that the parameters are more important. The returned dictionary is ordered by its values in a descending order. By default, the sum of the importance values are normalized to 1.0.

If `params` is `None`, all parameter that are present in all of the completed trials are assessed. This implies that conditional parameters will be excluded from the evaluation. To assess the importances of conditional parameters, a `list` of parameter names can be specified via `params`. If specified, only completed trials that contain all of the parameters will be considered. If no such trials are found, an error will be raised.

If the given study does not contain completed trials, an error will be raised.

Note: If `params` is specified as an empty list, an empty dictionary is returned.

See also:

See *plot_param_importances()* to plot importances.

> **Parameters**
> * **study** (Study) – An optimized study.

- **evaluator** (*BaseImportanceEvaluator | None*) – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *FanovaImportanceEvaluator*.

Note: *FanovaImportanceEvaluator* takes over 1 minute when given a study that contains 1000+ trials. We published optuna-fast-fanova library, that is a Cython accelerated fANOVA implementation. By using it, you can get hyperparameter importances within a few seconds. If n_trials is more than 10000, the Cython implementation takes more than a minute, so you can use *PedAnovaImportanceEvaluator* instead, enabling the evaluation to finish in a second.

- **params** (*List[str] | None*) – A list of names of parameters to assess. If None, all parameters that are present in all of the completed trials are assessed.
- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to evaluate importances. If it is None and study is being used for single-objective optimization, the objective values are used. target must be specified if study is being used for multi-objective optimization.

Note: Specify this argument if study is being used for multi-objective optimization. For example, to get the hyperparameter importance of the first objective, use target=lambda t: t.values[0] for the target parameter.

- **normalize** (*bool*) – A boolean option to specify whether the sum of the importance values should be normalized to 1.0. Defaults to True.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Returns

A dict where the keys are parameter names and the values are assessed importances.

Return type

Dict[str, float]

optuna.importance.FanovaImportanceEvaluator

class optuna.importance.**FanovaImportanceEvaluator**(**, n_trees=64, max_depth=64, seed=None*)

fANOVA importance evaluator.

Implements the fANOVA hyperparameter importance evaluation algorithm in An Efficient Approach for Assessing Hyperparameter Importance.

fANOVA fits a random forest regression model that predicts the objective values of *COMPLETE* trials given their parameter configurations. The more accurate this model is, the more reliable the importances assessed by this class are.

Note: This class takes over 1 minute when given a study that contains 1000+ trials. We published optuna-fast-fanova library, that is a Cython accelerated fANOVA implementation. By using it, you can get hyperparameter importances within a few seconds.

Note: Requires the sklearn Python package.

Note: The performance of fANOVA depends on the prediction performance of the underlying random forest model. In order to obtain high prediction performance, it is necessary to cover a wide range of the hyperparameter search space. It is recommended to use an exploration-oriented sampler such as *RandomSampler*.

Note: For how to cite the original work, please refer to https://automl.github.io/fanova/cite.html.

Parameters

- **n_trees** (*int*) – The number of trees in the forest.

- **max_depth** (*int*) – The maximum depth of the trees in the forest.

- **seed** (*int | None*) – Controls the randomness of the forest. For deterministic behavior, specify a value other than None.

Methods

evaluate(study[, params, target])	Evaluate parameter importances based on completed trials in the given study.

evaluate(*study*, *params=None*, ***, *target=None*)

Evaluate parameter importances based on completed trials in the given study.

Note: This method is not meant to be called by library users.

See also:

Please refer to *get_param_importances()* for how a concrete evaluator should implement this method.

Parameters

- **study** (*Study*) – An optimized study.

- **params** (*List[str] | None*) – A list of names of parameters to assess. If None, all parameters that are present in all of the completed trials are assessed.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to evaluate importances. If it is None and `study` is being used for single-objective optimization, the objective values are used. Can also be used for other trial attributes, such as the duration, like `target=lambda t:  t.duration.total_seconds()`.

   Note: Specify this argument if `study` is being used for multi-objective optimization. For example, to get the hyperparameter importance of the first objective, use `target=lambda t:  t.values[0]` for the target parameter.

Returns
>   A `dict` where the keys are parameter names and the values are assessed importances.

Return type
>   *Dict*[str, float]

optuna.importance.MeanDecreaseImpurityImportanceEvaluator

class optuna.importance.**MeanDecreaseImpurityImportanceEvaluator**(*, *n_trees=64*, *max_depth=64*,
>   *seed=None*)

Mean Decrease Impurity (MDI) parameter importance evaluator.

This evaluator fits fits a random forest regression model that predicts the objective values of *COMPLETE* trials given their parameter configurations. Feature importances are then computed using MDI.

Note: This evaluator requires the sklearn Python package and is based on sklearn.ensemble.RandomForestClassifier.feature_importances_.

Parameters

- **n_trees** (*int*) – Number of trees in the random forest.

- **max_depth** (*int*) – The maximum depth of each tree in the random forest.

- **seed** (*int | None*) – Seed for the random forest.

Methods

evaluate(study[, params, target])	Evaluate parameter importances based on completed trials in the given study.

evaluate(*study*, *params=None*, ***, *target=None*)
>   Evaluate parameter importances based on completed trials in the given study.

Note: This method is not meant to be called by library users.

See also:

Please refer to *get_param_importances()* for how a concrete evaluator should implement this method.

Parameters

- **study** (*Study*) – An optimized study.

- **params** (*List[str] | None*) – A list of names of parameters to assess. If None, all parameters that are present in all of the completed trials are assessed.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to evaluate importances. If it is None and study is being used for single-objective optimization, the objective values are used. Can also be used for other trial attributes, such as the duration, like `target=lambda t: t.duration.total_seconds()`.

Note: Specify this argument if `study` is being used for multi-objective optimization. For example, to get the hyperparameter importance of the first objective, use `target=lambda t:  t.values[0]` for the target parameter.

Returns
> A `dict` where the keys are parameter names and the values are assessed importances.

Return type
> *Dict*[str, float]

optuna.importance.PedAnovaImportanceEvaluator

class optuna.importance.**PedAnovaImportanceEvaluator**(**, baseline_quantile=0.1,*
evaluate_on_local=True)

PED-ANOVA importance evaluator.

Implements the PED-ANOVA hyperparameter importance evaluation algorithm.

PED-ANOVA fits Parzen estimators of *COMPLETE* trials better than a user-specified baseline. Users can specify the baseline by a quantile. The importance can be interpreted as how important each hyperparameter is to get the performance better than baseline.

For further information about PED-ANOVA algorithm, please refer to the following paper:

- PED-ANOVA: Efficiently Quantifying Hyperparameter Importance in Arbitrary Subspaces

---

Note: The performance of PED-ANOVA depends on how many trials to consider above baseline. To stabilize the analysis, it is preferable to include at least 5 trials above baseline.

Note: Please refer to the original work.

Parameters

- **baseline_quantile** (*float*) – Compute the importance of achieving top-`baseline_quantile` quantile objective value. For example, `baseline_quantile=0.1` means that the importances give the information of which parameters were important to achieve the top-10% performance during optimization.

- **evaluate_on_local** (*bool*) – Whether we measure the importance in the local or global space. If `True`, the importances imply how importance each parameter is during optimization. Meanwhile, `evaluate_on_local=False` gives the importances in the specified search_space. `evaluate_on_local=True` is especially useful when users modify search space during optimization.

Example

An example of using PED-ANOVA is as follows:

```python
import optuna
from optuna.importance import PedAnovaImportanceEvaluator


def objective(trial):
    x1 = trial.suggest_float("x1", -10, 10)
    x2 = trial.suggest_float("x2", -10, 10)
    return x1 + x2 / 1000


study = optuna.create_study()
study.optimize(objective, n_trials=100)
evaluator = PedAnovaImportanceEvaluator()
evaluator.evaluate(study)
```

Note: Added in v3.6.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.6.0.

Methods

evaluate(study[, params, target])	Evaluate parameter importances based on completed trials in the given study.

evaluate(*study*, *params=None*, ***, *target=None*)

Evaluate parameter importances based on completed trials in the given study.

Note: This method is not meant to be called by library users.

See also:

Please refer to *get_param_importances()* for how a concrete evaluator should implement this method.

> **Parameters**
> - **study** (*Study*) – An optimized study.
> - **params** (*list[str] | None*) – A list of names of parameters to assess. If *None*, all parameters that are present in all of the completed trials are assessed.
> - **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to evaluate importances. If it is *None* and `study` is being used for single-objective optimization, the objective values are used. Can also be used for other trial attributes, such as the duration, like `target=lambda t: t.duration.total_seconds()`.
>
> > **Note:** Specify this argument if `study` is being used for multi-objective optimization. For example, to get the hyperparameter importance of the first objective, use `target=lambda`

> t: t.values[0] for the target parameter.

> **Returns**
>> A `dict` where the keys are parameter names and the values are assessed importances.

> **Return type**
>> dict[str, float]

7.3.7 optuna.integration

The `integration` module contains classes used to integrate Optuna with external machine learning frameworks.

Note: Optuna's integration modules for third-party libraries have started migrating from Optuna itself to a package called *optuna-integration*. Please check the repository and the documentation.

For most of the ML frameworks supported by Optuna, the corresponding Optuna integration class serves only to implement a callback object and functions, compliant with the framework's specific callback API, to be called with each intermediate step in the model training. The functionality implemented in these callbacks across the different ML frameworks includes:

(1) Reporting intermediate model scores back to the Optuna trial using `optuna.trial.Trial.report()`,

(2) According to the results of `optuna.trial.Trial.should_prune()`, pruning the current model by raising `optuna.TrialPruned()`, and

(3) Reporting intermediate Optuna data such as the current trial number back to the framework, as done in `MLflowCallback`.

For scikit-learn, an integrated `OptunaSearchCV` estimator is available that combines scikit-learn BaseEstimator functionality with access to a class-level `Study` object.

Dependencies of each integration

We summarize the necessary dependencies for each integration.

Integration	Dependencies
AllenNLP	allennlp, torch, psutil, jsonnet
BoTorch	botorch, gpytorch, torch
Catalyst	catalyst
CatBoost	catboost
ChainerMN	chainermn
Chainer	chainer
pycma	cma
Dask	distributed
FastAI (v1, v2)	fastai
Keras	keras
LightGBMTuner	lightgbm, scikit-learn
LightGBMPruningCallback	lightgbm
MLflow	mlflow
MXNet	mxnet
PyTorch Distributed	torch
PyTorch (Ignite)	pytorch-ignite
PyTorch (Lightning)	pytorch-lightning
SHAP	scikit-learn, shap
Scikit-learn	pandas, scipy, scikit-learn
Scikit-optimize	scikit-optimize
SKorch	skorch
TensorBoard	tensorboard, tensorflow
TensorFlow	tensorflow, tensorflow-estimator
TensorFlow + Keras	tensorflow
Weights & Biases	wandb
XGBoost	xgboost

7.3.8 optuna.logging

The *logging* module implements logging using the Python `logging` package. Library users may be especially interested in setting verbosity levels using *set_verbosity()* to one of `optuna.logging.CRITICAL` (aka `optuna.logging.FATAL`), `optuna.logging.ERROR`, `optuna.logging.WARNING` (aka `optuna.logging.WARN`), `optuna.logging.INFO`, or `optuna.logging.DEBUG`.

optuna.logging.get_verbosity	Return the current level for the Optuna's root logger.
optuna.logging.set_verbosity	Set the level for the Optuna's root logger.
optuna.logging.disable_default_handler	Disable the default handler of the Optuna's root logger.
optuna.logging.enable_default_handler	Enable the default handler of the Optuna's root logger.
optuna.logging.disable_propagation	Disable propagation of the library log outputs.
optuna.logging.enable_propagation	Enable propagation of the library log outputs.

optuna.logging.get_verbosity

optuna.logging.**get_verbosity**()

    Return the current level for the Optuna's root logger.

Example

Get the default verbosity level.

```python
import optuna

# The default verbosity level of Optuna is `optuna.logging.INFO`.
print(optuna.logging.get_verbosity())
# 20
print(optuna.logging.INFO)
# 20

# There are logs of the INFO level.
study = optuna.create_study()
study.optimize(objective, n_trials=5)
# [I 2021-10-31 05:35:17,232] A new study created ...
# [I 2021-10-31 05:35:17,238] Trial 0 finished with value: ...
# [I 2021-10-31 05:35:17,245] Trial 1 finished with value: ...
# ...
```

    Returns
        Logging level, e.g., optuna.logging.DEBUG and optuna.logging.INFO.

    Return type
        int

Note: Optuna has following logging levels:

- optuna.logging.CRITICAL, optuna.logging.FATAL
- optuna.logging.ERROR
- optuna.logging.WARNING, optuna.logging.WARN
- optuna.logging.INFO
- optuna.logging.DEBUG

optuna.logging.set_verbosity

optuna.logging.**set_verbosity**(*verbosity*)

    Set the level for the Optuna's root logger.

Example

Set the logging level optuna.logging.WARNING.

```python
import optuna

# There are INFO level logs.
study = optuna.create_study()
study.optimize(objective, n_trials=10)
# [I 2021-10-31 02:59:35,088] Trial 0 finished with value: 16.0 ...
# [I 2021-10-31 02:59:35,091] Trial 1 finished with value: 1.0 ...
# [I 2021-10-31 02:59:35,096] Trial 2 finished with value: 1.0 ...

# Setting the logging level WARNING, the INFO logs are suppressed.
optuna.logging.set_verbosity(optuna.logging.WARNING)
study.optimize(objective, n_trials=10)
```

> **Parameters**
> > **verbosity** (*int*) – Logging level, e.g., optuna.logging.DEBUG and optuna.logging.
> > INFO.
>
> **Return type**
> > None

Note: Optuna has following logging levels:

- optuna.logging.CRITICAL, optuna.logging.FATAL

- optuna.logging.ERROR

- optuna.logging.WARNING, optuna.logging.WARN

- optuna.logging.INFO

- optuna.logging.DEBUG

---

optuna.logging.disable_default_handler

optuna.logging.**disable_default_handler**()
> Disable the default handler of the Optuna's root logger.

Example

Stop and then resume logging to sys.stderr.

```python
import optuna

study = optuna.create_study()

# There are no logs in sys.stderr.
optuna.logging.disable_default_handler()
study.optimize(objective, n_trials=10)
```

```
# There are logs in sys.stderr.
optuna.logging.enable_default_handler()
study.optimize(objective, n_trials=10)
# [I 2020-02-23 17:00:54,314] Trial 10 finished with value: ...
# [I 2020-02-23 17:00:54,356] Trial 11 finished with value: ...
# ...
```

> **Return type**
> > None

optuna.logging.enable_default_handler

optuna.logging.**enable_default_handler**()

> Enable the default handler of the Optuna's root logger.
>
> Please refer to the example shown in *disable_default_handler()*.
>
> > **Return type**
> > > None

optuna.logging.disable_propagation

optuna.logging.**disable_propagation**()

> Disable propagation of the library log outputs.
>
> Note that log propagation is disabled by default. You only need to use this function to stop log propagation when you use *enable_propagation()*.
>
> **Example**
>
> Stop propagating logs to the root logger on the second optimize call.

```
import optuna
import logging

optuna.logging.disable_default_handler()  # Disable the default handler.
logger = logging.getLogger()

logger.setLevel(logging.INFO)  # Setup the root logger.
logger.addHandler(logging.FileHandler("foo.log", mode="w"))

optuna.logging.enable_propagation()  # Propagate logs to the root logger.

study = optuna.create_study()

logger.info("Logs from first optimize call")  # The logs are saved in the logs file.
study.optimize(objective, n_trials=10)

optuna.logging.disable_propagation()  # Stop propogating logs to the root logger.
```

```python
logger.info("Logs from second optimize call")
# The new logs for second optimize call are not saved.
study.optimize(objective, n_trials=10)

with open("foo.log") as f:
    assert f.readline().startswith("A new study created")
    assert f.readline() == "Logs from first optimize call\n"
    # Check for logs after second optimize call.
    assert f.read().split("Logs from second optimize call\n")[-1] == ""
```

> **Return type**
>> None

optuna.logging.enable_propagation

optuna.logging.**enable_propagation**()

> Enable propagation of the library log outputs.
>
> Please disable the Optuna's default handler to prevent double logging if the root logger has been configured.

> ### Example
>
> Propagate all log output to the root logger in order to save them to the file.
>
> ```python
> import optuna
> import logging
>
>
> logger = logging.getLogger()
>
> logger.setLevel(logging.INFO)  # Setup the root logger.
> logger.addHandler(logging.FileHandler("foo.log", mode="w"))
>
> optuna.logging.enable_propagation()  # Propagate logs to the root logger.
> optuna.logging.disable_default_handler()  # Stop showing logs in sys.stderr.
>
> study = optuna.create_study()
>
> logger.info("Start optimization.")
> study.optimize(objective, n_trials=10)
>
> with open("foo.log") as f:
>     assert f.readline().startswith("A new study created")
>     assert f.readline() == "Start optimization.\n"
> ```

> **Return type**
>> None

7.3.9 optuna.pruners

The *pruners* module defines a *BasePruner* class characterized by an abstract *prune()* method, which, for a given trial and its associated study, returns a boolean value representing whether the trial should be pruned. This determination is made based on stored intermediate values of the objective function, as previously reported for the trial using *optuna.trial.Trial.report()*. The remaining classes in this module represent child classes, inheriting from *BasePruner*, which implement different pruning strategies.

> **Warning:** Currently *pruners* module is expected to be used only for single-objective optimization.

See also:

pruning tutorial explains the concept of the pruner classes and a minimal example.

See also:

user_defined_pruner tutorial could be helpful if you want to implement your own pruner classes.

optuna.pruners.BasePruner	Base class for pruners.
optuna.pruners.MedianPruner	Pruner using the median stopping rule.
optuna.pruners.NopPruner	Pruner which never prunes trials.
optuna.pruners.PatientPruner	Pruner which wraps another pruner with tolerance.
optuna.pruners.PercentilePruner	Pruner to keep the specified percentile of the trials.
optuna.pruners.SuccessiveHalvingPruner	Pruner using Asynchronous Successive Halving Algorithm.
optuna.pruners.HyperbandPruner	Pruner using Hyperband.
optuna.pruners.ThresholdPruner	Pruner to detect outlying metrics of the trials.
optuna.pruners.WilcoxonPruner	Pruner based on the Wilcoxon signed-rank test.

optuna.pruners.BasePruner

class optuna.pruners.**BasePruner**

    Base class for pruners.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

abstract prune(*study*, *trial*)

    Judge whether the trial should be pruned based on the reported values.

    Note that this method is not supposed to be called by library users. Instead, *optuna.trial.Trial.report()* and *optuna.trial.Trial.should_prune()* provide user interfaces to implement pruning mechanism in an objective function.

        Parameters

            • **study** (Study) – Study object of the target study.

            • **trial** (FrozenTrial) – FrozenTrial object of the target trial. Take a copy before modifying this object.

> **Returns**
>> A boolean value representing whether the trial should be pruned.
>
> **Return type**
>> bool

optuna.pruners.MedianPruner

class optuna.pruners.**MedianPruner**(*n_startup_trials=5*, *n_warmup_steps=0*, *interval_steps=1*, *, *n_min_trials=1*)

Pruner using the median stopping rule.

Prune if the trial's best intermediate result is worse than median of intermediate results of previous trials at the same step.

Example

We minimize an objective function with the median stopping rule.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.MedianPruner(
        n_startup_trials=5, n_warmup_steps=30, interval_steps=10
    ),
```

```
)
study.optimize(objective, n_trials=20)
```

Parameters

- **n_startup_trials** (*int*) – Pruning is disabled until the given number of trials finish in the same study.

- **n_warmup_steps** (*int*) – Pruning is disabled until the trial exceeds the given number of step. Note that this feature assumes that `step` starts at zero.

- **interval_steps** (*int*) – Interval in number of steps between the pruning checks, offset by the warmup steps. If no value has been reported at the time of a pruning check, that particular check will be postponed until a value is reported.

- **n_min_trials** (*int*) – Minimum number of reported trial results at a step to judge whether to prune. If the number of reported intermediate values from all trials at the current step is less than `n_min_trials`, the trial will not be pruned. This can be used to ensure that a minimum number of trials are run to completion without being pruned.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, *optuna.trial.Trial.report()* and *optuna.trial.Trial.should_prune()* provide user interfaces to implement pruning mechanism in an objective function.

Parameters

- **study** (*Study*) – Study object of the target study.

- **trial** (*FrozenTrial*) – FrozenTrial object of the target trial. Take a copy before modifying this object.

Returns

A boolean value representing whether the trial should be pruned.

Return type

bool

optuna.pruners.NopPruner

class optuna.pruners.**NopPruner**

Pruner which never prunes trials.

Example

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            assert False, "should_prune() should always return False with this␣
↪pruner."
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize", pruner=optuna.pruners.NopPruner())
study.optimize(objective, n_trials=20)
```

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, *optuna.trial.Trial.report()* and *optuna.trial.Trial.should_prune()* provide user interfaces to implement pruning mechanism in an objective function.

> **Parameters**
>> - **study** (Study) – Study object of the target study.
>>
>> - **trial** (FrozenTrial) – FrozenTrial object of the target trial. Take a copy before modifying this object.
>
> **Returns**
>> A boolean value representing whether the trial should be pruned.
>
> **Return type**
>> bool

optuna.pruners.PatientPruner

class optuna.pruners.**PatientPruner**(*wrapped_pruner*, *patience*, *min_delta=0.0*)

> Pruner which wraps another pruner with tolerance.

Example

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(
```

(continues on next page)

```
    direction="maximize",
    pruner=optuna.pruners.PatientPruner(optuna.pruners.MedianPruner(), patience=1),
)
study.optimize(objective, n_trials=20)
```

Parameters

- **wrapped_pruner** (`BasePruner` | *None*) – Wrapped pruner to perform pruning when
  `PatientPruner` allows a trial to be pruned. If it is `None`, this pruner is equivalent to early-
  stopping taken the intermediate values in the individual trial.

- **patience** (`int`) – Pruning is disabled until the objective doesn't improve for `patience`
  consecutive steps.

- **min_delta** (`float`) – Tolerance value to check whether or not the objective improves. This
  value should be non-negative.

Note: Added in v2.8.0 as an experimental feature. The interface may change in newer versions without prior
notice. See https://github.com/optuna/optuna/releases/tag/v2.8.0.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, `optuna.trial.Trial.`
`report()` and `optuna.trial.Trial.should_prune()` provide user interfaces to implement pruning
mechanism in an objective function.

Parameters

- **study** (`Study`) – Study object of the target study.

- **trial** (`FrozenTrial`) – FrozenTrial object of the target trial. Take a copy before modify-
  ing this object.

Returns

A boolean value representing whether the trial should be pruned.

Return type

bool

optuna.pruners.PercentilePruner

class optuna.pruners.**PercentilePruner**(*percentile*, *n_startup_trials=5*, *n_warmup_steps=0*, *interval_steps=1*, ***, *n_min_trials=1*)

Pruner to keep the specified percentile of the trials.

Prune if the best intermediate value is in the bottom percentile among trials at the same step.

Example

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.PercentilePruner(
        25.0, n_startup_trials=5, n_warmup_steps=30, interval_steps=10
    ),
)
study.optimize(objective, n_trials=20)
```

Parameters

- **percentile** (*float*) – Percentile which must be between 0 and 100 inclusive (e.g., When given 25.0, top of 25th percentile trials are kept).

- **n_startup_trials** (*int*) – Pruning is disabled until the given number of trials finish in the same study.

- **n_warmup_steps** (*int*) – Pruning is disabled until the trial exceeds the given number of step. Note that this feature assumes that step starts at zero.

- **interval_steps** (*int*) – Interval in number of steps between the pruning checks, offset by the warmup steps. If no value has been reported at the time of a pruning check, that particular check will be postponed until a value is reported. Value must be at least 1.

- **n_min_trials** (*int*) – Minimum number of reported trial results at a step to judge whether to prune. If the number of reported intermediate values from all trials at the current step is less than n_min_trials, the trial will not be pruned. This can be used to ensure that a minimum number of trials are run to completion without being pruned.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, `optuna.trial.Trial.report()` and `optuna.trial.Trial.should_prune()` provide user interfaces to implement pruning mechanism in an objective function.

Parameters

- **study** (*Study*) – Study object of the target study.

- **trial** (*FrozenTrial*) – FrozenTrial object of the target trial. Take a copy before modifying this object.

Returns

A boolean value representing whether the trial should be pruned.

Return type

bool

optuna.pruners.SuccessiveHalvingPruner

class optuna.pruners.**SuccessiveHalvingPruner**(*min_resource='auto'*, *reduction_factor=4*, *min_early_stopping_rate=0*, *bootstrap_count=0*)

Pruner using Asynchronous Successive Halving Algorithm.

Successive Halving is a bandit-based algorithm to identify the best one among multiple configurations. This class implements an asynchronous version of Successive Halving. Please refer to the paper of Asynchronous Successive Halving for detailed descriptions.

Note that, this class does not take care of the parameter for the maximum resource, referred to as R in the paper. The maximum resource allocated to a trial is typically limited inside the objective function (e.g., step number in simple_pruning.py, EPOCH number in chainer_integration.py).

See also:

Please refer to report().

Example

We minimize an objective function with `SuccessiveHalvingPruner`.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(
    direction="maximize", pruner=optuna.pruners.SuccessiveHalvingPruner()
)
study.optimize(objective, n_trials=20)
```

Parameters

- **min_resource** (`str` | `int`) – A parameter for specifying the minimum resource allocated to a trial (in the paper this parameter is referred to as r). This parameter defaults to 'auto' where the value is determined based on a heuristic that looks at the number of required steps for the first trial to complete.

  A trial is never pruned until it executes $\text{min\_resource} \times \text{reduction\_factor}^{\text{min\_early\_stopping\_rate}}$ steps (i.e., the completion point of the first rung). When the trial completes the first rung, it will be promoted to the next rung only if the value of the trial is placed in the top $\frac{1}{\text{reduction\_factor}}$ fraction of the all trials that already have reached the point (otherwise it will be pruned there). If the trial won the competition, it runs until the next completion point (i.e., $\text{min\_resource} \times \text{reduction\_factor}^{(\text{min\_early\_stopping\_rate}+\text{rung})}$ steps) and repeats the same procedure.

  Note: If the step of the last intermediate value may change with each trial, please manually

specify the minimum possible step to `min_resource`.

- **reduction_factor** (`int`) – A parameter for specifying reduction factor of promotable trials (in the paper this parameter is referred to as η). At the completion point of each rung, about $\frac{1}{\text{reduction\_factor}}$ trials will be promoted.

- **min_early_stopping_rate** (`int`) – A parameter for specifying the minimum early-stopping rate (in the paper this parameter is referred to as s).

- **bootstrap_count** (`int`) – Minimum number of trials that need to complete a rung before any trial is considered for promotion into the next rung.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, `optuna.trial.Trial.report()` and `optuna.trial.Trial.should_prune()` provide user interfaces to implement pruning mechanism in an objective function.

> **Parameters**
> - **study** (`Study`) – Study object of the target study.
> - **trial** (`FrozenTrial`) – FrozenTrial object of the target trial. Take a copy before modifying this object.
>
> **Returns**
> A boolean value representing whether the trial should be pruned.
>
> **Return type**
> bool

optuna.pruners.HyperbandPruner

class optuna.pruners.**HyperbandPruner**(*min_resource=1*, *max_resource='auto'*, *reduction_factor=3*, *bootstrap_count=0*)

Pruner using Hyperband.

As SuccessiveHalving (SHA) requires the number of configurations n as its hyperparameter. For a given finite budget B, all the configurations have the resources of $\frac{B}{n}$ on average. As you can see, there will be a trade-off of B and $\frac{B}{n}$. Hyperband attacks this trade-off by trying different n values for a fixed budget.

Note:

- In the Hyperband paper, the counterpart of *RandomSampler* is used.
- Optuna uses *TPESampler* by default.
- The benchmark result shows that *optuna.pruners.HyperbandPruner* supports both samplers.

Note: If you use HyperbandPruner with *TPESampler*, it's recommended to consider setting larger `n_trials` or `timeout` to make full use of the characteristics of *TPESampler* because *TPESampler* uses some (by default, 10) *Trial*s for its startup.

As Hyperband runs multiple *SuccessiveHalvingPruner* and collects trials based on the current *Trial*'s bracket ID, each bracket needs to observe more than 10 *Trial*s for *TPESampler* to adapt its search space.

Thus, for example, if HyperbandPruner has 4 pruners in it, at least 4×10 trials are consumed for startup.

Note: Hyperband has several *SuccessiveHalvingPruner*s. Each *SuccessiveHalvingPruner* is referred to as "bracket" in the original paper. The number of brackets is an important factor to control the early stopping behavior of Hyperband and is automatically determined by `min_resource`, `max_resource` and `reduction_factor` as The number of brackets $= \text{floor}(\log_{\text{reduction\_factor}}(\frac{\text{max\_resource}}{\text{min\_resource}})) + 1$. Please set `reduction_factor` so that the number of brackets is not too large (about $4 - 6$ in most use cases). Please see Section 3.6 of the original paper for the detail.

Example

We minimize an objective function with Hyperband pruning algorithm.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)
n_train_iter = 100


def objective(trial):
    alpha = trial.suggest_float("alpha", 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)
```

(continues on next page)

```
study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.HyperbandPruner(
        min_resource=1, max_resource=n_train_iter, reduction_factor=3
    ),
)
study.optimize(objective, n_trials=20)
```

> **Parameters**
>
> - **min_resource** (*int*) – A parameter for specifying the minimum resource allocated to a trial noted as r in the paper. A smaller r will give a result faster, but a larger r will give a better guarantee of successful judging between configurations. See the details for *SuccessiveHalvingPruner*.
>
> - **max_resource** (*str | int*) – A parameter for specifying the maximum resource allocated to a trial. R in the paper corresponds to `max_resource / min_resource`. This value represents and should match the maximum iteration steps (e.g., the number of epochs for neural networks). When this argument is "auto", the maximum resource is estimated according to the completed trials. The default value of this argument is "auto".
>
>   ---
>
>   **Note:** With "auto", the maximum resource will be the largest step reported by *report()* in the first, or one of the first if trained in parallel, completed trial. No trials will be pruned until the maximum resource is determined.
>
>   ---
>
>   ---
>
>   **Note:** If the step of the last intermediate value may change with each trial, please manually specify the maximum possible step to `max_resource`.
>
>   ---
>
> - **reduction_factor** (*int*) – A parameter for specifying reduction factor of promotable trials noted as η in the paper. See the details for *SuccessiveHalvingPruner*.
>
> - **bootstrap_count** (*int*) – Parameter specifying the number of trials required in a rung before any trial can be promoted. Incompatible with `max_resource` is `"auto"`. See the details for *SuccessiveHalvingPruner*.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

> Judge whether the trial should be pruned based on the reported values.
>
> Note that this method is not supposed to be called by library users. Instead, *optuna.trial.Trial.report()* and *optuna.trial.Trial.should_prune()* provide user interfaces to implement pruning mechanism in an objective function.
>
> > **Parameters**
> >
> > - **study** (*Study*) – Study object of the target study.

- **trial** (FrozenTrial) – FrozenTrial object of the target trial. Take a copy before modifying this object.

> **Returns**
>> A boolean value representing whether the trial should be pruned.
>
> **Return type**
>> bool

optuna.pruners.ThresholdPruner

class optuna.pruners.**ThresholdPruner**(*lower=None*, *upper=None*, *n_warmup_steps=0*, *interval_steps=1*)

> Pruner to detect outlying metrics of the trials.
>
> Prune if a metric exceeds upper threshold, falls behind lower threshold or reaches nan.
>
> ### Example
>
> ```python
> from optuna import create_study
> from optuna.pruners import ThresholdPruner
> from optuna import TrialPruned
>
>
> def objective_for_upper(trial):
>     for step, y in enumerate(ys_for_upper):
>         trial.report(y, step)
>
>         if trial.should_prune():
>             raise TrialPruned()
>     return ys_for_upper[-1]
>
>
> def objective_for_lower(trial):
>     for step, y in enumerate(ys_for_lower):
>         trial.report(y, step)
>
>         if trial.should_prune():
>             raise TrialPruned()
>     return ys_for_lower[-1]
>
>
> ys_for_upper = [0.0, 0.1, 0.2, 0.5, 1.2]
> ys_for_lower = [100.0, 90.0, 0.1, 0.0, -1]
>
> study = create_study(pruner=ThresholdPruner(upper=1.0))
> study.optimize(objective_for_upper, n_trials=10)
>
> study = create_study(pruner=ThresholdPruner(lower=0.0))
> study.optimize(objective_for_lower, n_trials=10)
> ```
>
> **Parameters**
>> - **lower** (*float | None*) – A minimum value which determines whether pruner prunes or not. If an intermediate value is smaller than lower, it prunes.

- **upper** (*float* | *None*) – A maximum value which determines whether pruner prunes or not. If an intermediate value is larger than upper, it prunes.

- **n_warmup_steps** (*int*) – Pruning is disabled if the step is less than the given number of warmup steps.

- **interval_steps** (*int*) – Interval in number of steps between the pruning checks, offset by the warmup steps. If no value has been reported at the time of a pruning check, that particular check will be postponed until a value is reported. Value must be at least 1.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, *optuna.trial.Trial.report()* and *optuna.trial.Trial.should_prune()* provide user interfaces to implement pruning mechanism in an objective function.

> **Parameters**
>
> - **study** (*Study*) – Study object of the target study.
>
> - **trial** (*FrozenTrial*) – FrozenTrial object of the target trial. Take a copy before modifying this object.
>
> **Returns**
> A boolean value representing whether the trial should be pruned.
>
> **Return type**
> bool

optuna.pruners.WilcoxonPruner

class optuna.pruners.**WilcoxonPruner**(***, *p_threshold=0.1*, *n_startup_steps=0*)

Pruner based on the Wilcoxon signed-rank test.

This pruner performs the Wilcoxon signed-rank test between the current trial and the current best trial, and stops whenever the pruner is sure up to a given p-value that the current trial is worse than the best one.

This pruner is effective for optimizing the mean/median of some (costly-to-evaluate) performance scores over a set of problem instances. Example applications include the optimization of: - the mean performance of a heuristic method (simulated annealing, genetic algorithm, SAT solver, etc.) on a set of problem instances, - the k-fold cross-validation score of a machine learning model, and - the accuracy of outputs of a large language model (LLM) on a set of questions.

There can be "easy" or "hard" instances (the pruner handles correspondence of the instances between different trials). In each trial, it is recommended to shuffle the evaluation order, so that the optimization doesn't overfit to the instances in the beginning.

When you use this pruner, you must call *Trial.report(value, step)* function for each step (instance id) with the evaluated value. The instance id may not be in ascending order. This is different from other pruners in that the reported value need not converge to the real value. (To use pruners such as *SuccessiveHalvingPruner* in the same setting, you must provide e.g., the historical average of the evaluated values.)

See also:

Please refer to *report()*.

Example

```python
import optuna
import numpy as np


# We minimize the mean evaluation loss over all the problem instances.
def evaluate(param, instance):
    # A toy loss function for demonstrative purpose.
    return (param - instance) ** 2


problem_instances = np.linspace(-1, 1, 100)


def objective(trial):
    # Sample a parameter.
    param = trial.suggest_float("param", -1, 1)

    # Evaluate performance of the parameter.
    results = []

    # For best results, shuffle the evaluation order in each trial.
    instance_ids = np.random.permutation(len(problem_instances))
    for instance_id in instance_ids:
        loss = evaluate(param, problem_instances[instance_id])
        results.append(loss)

        # Report loss together with the instance id.
        # CAVEAT: You need to pass the same id for the same instance,
        # otherwise WilcoxonPruner cannot correctly pair the losses across trials
→and
        # the pruning performance will degrade.
        trial.report(loss, instance_id)

        if trial.should_prune():
            # Return the current predicted value instead of raising `TrialPruned`.
            # This is a workaround to tell the Optuna about the evaluation
            # results in pruned trials. (See the note below.)
            return sum(results) / len(results)

    return sum(results) / len(results)


study = optuna.create_study(pruner=optuna.pruners.WilcoxonPruner(p_threshold=0.1))
study.optimize(objective, n_trials=100)
```

Note: This pruner cannot handle `infinity` or `nan` values. Trials containing those values are never pruned.

Note: If *trial.should_prune()* returns *True*, you can return an estimation of the final value (e.g., the average of all evaluated values) instead of *raise optuna.TrialPruned()*. This is a workaround for the problem that currently there is no way to tell Optuna the predicted objective value for trials raising *TrialPruned*.

Parameters

- **p_threshold** (`float`) – The p-value threshold for pruning. This value should be between 0 and 1. A trial will be pruned whenever the pruner is sure up to the given p-value that the current trial is worse than the best trial. The larger this value is, the more aggressive pruning will be performed. Defaults to 0.1.

  Note: This pruner repeatedly performs statistical tests between the current trial and the current best trial with increasing samples. The false-positive rate of such a sequential test is different from performing the test only once. To get the nominal false-positive rate, please specify the Pocock-corrected p-value.

- **n_startup_steps** (`int`) – The number of steps before which no trials are pruned. Pruning starts only after you have *n_startup_steps* steps of available observations for comparison between the current trial and the best trial. Defaults to 0 (pruning kicks in from the very first step).

Note: Added in v3.6.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.6.0.

Methods

prune(study, trial)	Judge whether the trial should be pruned based on the reported values.

prune(*study*, *trial*)

Judge whether the trial should be pruned based on the reported values.

Note that this method is not supposed to be called by library users. Instead, `optuna.trial.Trial.report()` and `optuna.trial.Trial.should_prune()` provide user interfaces to implement pruning mechanism in an objective function.

Parameters

- **study** (`Study`) – Study object of the target study.

- **trial** (`FrozenTrial`) – FrozenTrial object of the target trial. Take a copy before modifying this object.

Returns
A boolean value representing whether the trial should be pruned.

Return type
bool

7.3.10 optuna.samplers

The `samplers` module defines a base class for parameter sampling as described extensively in `BaseSampler`. The remaining classes in this module represent child classes, deriving from `BaseSampler`, which implement different sampling strategies.

See also:

pruning tutorial explains the overview of the sampler classes.

See also:

user_defined_sampler tutorial could be helpful if you want to implement your own sampler classes.

	Random-Sampler	Grid-Sampler	TPE-Sampler	CmaEs-Sampler	NSGAI-ISampler	QMC-Sampler	GP-Sampler	BoTorch Sampler	Brute-Force-Sampler
Float parameters	✓	✓	✓	✓	▲	✓	✓	✓	✓ (× for infinite domain)
Integer parameters	✓	✓	✓	✓	▲	✓	✓	▲	✓
Categorical parameters	✓	✓	✓	▲	✓	▲	✓	▲	✓
Pruning	✓	✓	✓	▲	×	✓	▲	▲	✓
Multivariate optimization	▲	▲	✓	✓	▲	▲	✓	✓	▲
Conditional search space	✓	▲	✓	▲	▲	▲	▲	▲	✓
Multi-objective optimization	✓	✓	✓	×	✓ (▲ for single-objective)	✓	×	✓	✓
Batch optimization	✓	✓	✓	✓	✓	✓	▲	✓	✓
Distributed optimization	✓	✓	✓	✓	✓	✓	▲	✓	✓
Constrained optimization	×	×	✓	×	✓	×	×	✓	×
Time complexity (per trial) (*)	$O(d)$	$O(dn)$	$O(dn\,\mathrm{l}\ldots$	$O(d^3)$	$O(mp^2)$ (***)	$O(dn)$	$O(n^3)$	$O(n^3)$	$O(d)$
Recommended budgets (#trials) (**)	as many as one likes	number of combinations	100 – 1000	1000 – 10000	100 – 10000	as many as one likes	– 500	10 – 100	number of combinations

Note: ✓: Supports this feature. ▲: Works, but inefficiently. ×: Causes an error, or has no interface.

(*): We assumes that d is the dimension of the search space, n is the number of finished trials, m is the number of objectives, and p is the population size (algorithm specific parameter). This table shows the time complexity of the sampling algorithms. We may omit other terms that depend on the implementation in Optuna, including $O(d)$ to call the sampling methods and $O(n)$ to collect the completed trials. This means that, for example, the actual time complexity of `RandomSampler` is $O(d + n + d) = O(d + n)$.

From another perspective, with the exception of *NSGAIISampler*, all time complexity is written for single-objective optimization.

(**): (1) The budget depends on the number of parameters and the number of objectives. (2) This budget includes `n_startup_trials` if a sampler has `n_startup_trials` as one of its arguments.

(***): This time complexity assumes that the number of population size p and the number of parallelization are regular. This means that the number of parallelization should not exceed the number of population size p.

Note: Samplers initialize their random number generators by specifying `seed` argument at initialization. However, samplers reseed them when `n_jobs!=1` of *optuna.study.Study.optimize()* to avoid sampling duplicated parameters by using the same generator. Thus we can hardly reproduce the optimization results with `n_jobs!=1`. For the same reason, make sure that use either `seed=None` or different `seed` values among processes with distributed optimization explained in distributed tutorial.

Note: For float, integer, or categorical parameters, see configurations tutorial.

For pruning, see pruning tutorial.

For multivariate optimization, see *BaseSampler*. The multivariate optimization is implemented as *sample_relative()* in Optuna. Please check the concrete documents of samplers for more details.

For conditional search space, see configurations tutorial and *TPESampler*. The `group` option of *TPESampler* allows *TPESampler* to handle the conditional search space.

For multi-objective optimization, see multi_objective tutorial.

For batch optimization, see Batch-Optimization tutorial. Note that the `constant_liar` option of *TPESampler* allows *TPESampler* to handle the batch optimization.

For distributed optimization, see distributed tutorial. Note that the `constant_liar` option of *TPESampler* allows *TPESampler* to handle the distributed optimization.

For constrained optimization, see an example.

---

optuna.samplers.BaseSampler	Base class for samplers.
optuna.samplers.GridSampler	Sampler using grid search.
optuna.samplers.RandomSampler	Sampler using random sampling.
optuna.samplers.TPESampler	Sampler using TPE (Tree-structured Parzen Estimator) algorithm.
optuna.samplers.CmaEsSampler	A sampler using cmaes as the backend.
optuna.samplers.GPSampler	Sampler using Gaussian process-based Bayesian optimization.
optuna.samplers.PartialFixedSampler	Sampler with partially fixed parameters.
optuna.samplers.NSGAIISampler	Multi-objective sampler using the NSGA-II algorithm.
optuna.samplers.NSGAIIISampler	Multi-objective sampler using the NSGA-III algorithm.
optuna.samplers.QMCSampler	A Quasi Monte Carlo Sampler that generates low-discrepancy sequences.
optuna.samplers.BruteForceSampler	Sampler using brute force.
optuna.samplers.IntersectionSearchSpace	A class to calculate the intersection search space of a *Study*.
optuna.samplers.intersection_search_space	Return the intersection search space of the *Study*.

optuna.samplers.BaseSampler

class optuna.samplers.`BaseSampler`

> Base class for samplers.
>
> Optuna combines two types of sampling strategies, which are called *relative sampling* and *independent sampling*.
>
> *The relative sampling* determines values of multiple parameters simultaneously so that sampling algorithms can use relationship between parameters (e.g., correlation). Target parameters of the relative sampling are described in a relative search space, which is determined by `infer_relative_search_space()`.
>
> *The independent sampling* determines a value of a single parameter without considering any relationship between parameters. Target parameters of the independent sampling are the parameters not described in the relative search space.
>
> More specifically, parameters are sampled by the following procedure. At the beginning of a trial, `infer_relative_search_space()` is called to determine the relative search space for the trial. During the execution of the objective function, `sample_relative()` is called only once when sampling the parameters belonging to the relative search space for the first time. `sample_independent()` is used to sample parameters that don't belong to the relative search space.
>
> The following figure depicts the lifetime of a trial and how the above three methods are called in the trial.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

> Trial post-processing.
>
> This method is called after the objective function returns and right before the trial is finished and its state is stored.
>
> ---
>
> **Note:** Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.
>
> ---
>
> **Parameters**
>
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> - **state** (*TrialState*) – Resulting trial state.
> - **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be None if trial succeeded.
>
> **Return type**
>
> > None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> ---
>
> **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> ---
>
> **Parameters**
>
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object.
>
> **Return type**
>
> > None

abstract **infer_relative_search_space**(*study*, *trial*)

Infer the search space that will be used by relative sampling in the target trial.

This method is called right before `sample_relative()` method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.

> **Parameters**
> - **study** (`Study`) – Target study object.
> - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
>
> **Returns**
> A dictionary containing the parameter names and parameter's distributions.
>
> **Return type**
> *Dict*[str, *BaseDistribution*]

See also:

Please refer to `intersection_search_space()` as an implementation of `infer_relative_search_space()`.

reseed_rng()

Reseed sampler's random number generator.

This method is called by the `Study` instance if trials are executed in parallel with the option `n_jobs>1`. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

> **Return type**
> None

abstract **sample_independent**(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by `sample_relative()` method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
> - **study** (`Study`) – Target study object.
> - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
> - **param_name** (`str`) – Name of the sampled parameter.
> - **param_distribution** (`BaseDistribution`) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
> A parameter value.
>
> **Return type**
> *Any*

---

abstract **sample_relative**(*study*, *trial*, *search_space*)

Sample parameters in a given search space.

This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

Parameters

- **study** (Study) – Target study object.

- **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.

- **search_space** (*Dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.

Returns

A dictionary containing the parameter names and the values.

Return type

Dict[str, *Any*]

optuna.samplers.GridSampler

class optuna.samplers.**GridSampler**(*search_space*, *seed=None*)

Sampler using grid search.

With *GridSampler*, the trials suggest all combinations of parameters in the given search space during the study.

Example

```
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_int("y", -100, 100)
    return x**2 + y**2


search_space = {"x": [-50, 0, 50], "y": [-99, 0, 99]}
study = optuna.create_study(sampler=optuna.samplers.GridSampler(search_space))
study.optimize(objective)
```

> **Note:** This sampler with ask_and_tell raises RuntimeError just after evaluating the final grid. This is because *GridSampler* automatically stops the optimization if all combinations in the passed search_space have already been evaluated, internally invoking the *stop()* method. As a workaround, we need to handle the error manually as in https://github.com/optuna/optuna/issues/4121#issuecomment-1305289910.

Note: *GridSampler* does not take care of a parameter's quantization specified by discrete suggest methods but just samples one of values specified in the search space. E.g., in the following code snippet, either of -0.5 or 0.5 is sampled as x instead of an integer point.

```python
import optuna


def objective(trial):
    # The following suggest method specifies integer points between -5 and 5.
    x = trial.suggest_float("x", -5, 5, step=1)
    return x**2


# Non-int points are specified in the grid.
search_space = {"x": [-0.5, 0.5]}
study = optuna.create_study(sampler=optuna.samplers.GridSampler(search_space))
study.optimize(objective, n_trials=2)
```

Note: A parameter configuration in the grid is not considered finished until its trial is finished. Therefore, during distributed optimization where trials run concurrently, different workers will occasionally suggest the same parameter configuration. The total number of actual trials may therefore exceed the size of the grid.

Note: All parameters must be specified when using *GridSampler* with *enqueue_trial()*.

Parameters

- **search_space** (*Mapping[str, Sequence[None | bool | int | float | str]]*) – A dictionary whose key and value are a parameter name and the corresponding candidates of values, respectively.

- **seed** (*int | None*) – A seed to fix the order of trials as the grid is randomly shuffled. Please note that it is not recommended using this option in distributed optimization settings since this option cannot ensure the order of trials and may increase the number of duplicate suggestions during distributed optimization.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

> Trial post-processing.
>
> This method is called after the objective function returns and right before the trial is finished and its state is stored.
>
> ---
>
> **Note:** Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.
>
> ---
>
> > **Parameters**
> >
> > - **study** (Study) – Target study object.
> >
> > - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
> >
> > - **state** (TrialState) – Resulting trial state.
> >
> > - **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be None if trial succeeded.
> >
> > **Return type**
> > None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> ---
>
> **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> ---
>
> > **Parameters**
> >
> > - **study** (Study) – Target study object.
> >
> > - **trial** (FrozenTrial) – Target trial object.
> >
> > **Return type**
> > None

infer_relative_search_space(*study*, *trial*)

> Infer the search space that will be used by relative sampling in the target trial.
>
> This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.
>
> > **Parameters**
> >
> > - **study** (Study) – Target study object.
> >
> > - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
> >
> > **Returns**
> > A dictionary containing the parameter names and parameter's distributions.

> **Return type**
> *Dict*[str, *BaseDistribution*]

See also:

Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

Reseed sampler's random number generator.

This method is called by the *Study* instance if trials are executed in parallel with the option `n_jobs`>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

> **Return type**
> None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> - **param_name** (*str*) – Name of the sampled parameter.
> - **param_distribution** (*BaseDistribution*) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
> A parameter value.
>
> **Return type**
> *Any*

sample_relative(*study*, *trial*, *search_space*)

Sample parameters in a given search space.

This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **search_space** (*Dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.

> **Returns**
> A dictionary containing the parameter names and the values.

> **Return type**
> *Dict*[str, *Any*]

optuna.samplers.RandomSampler

class optuna.samplers.**RandomSampler**(*seed=None*)

> Sampler using random sampling.
>
> This sampler is based on *independent sampling*. See also *BaseSampler* for more details of 'independent sampling'.

Example

```python
import optuna
from optuna.samplers import RandomSampler


def objective(trial):
    x = trial.suggest_float("x", -5, 5)
    return x**2


study = optuna.create_study(sampler=RandomSampler())
study.optimize(objective, n_trials=10)
```

> **Parameters**
> **seed** (*int | None*) – Seed for random number generator.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

> Trial post-processing.
>
> This method is called after the objective function returns and right before the trial is finished and its state is stored.
>
> ---
>
> **Note:** Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.
>
> ---
>
> > **Parameters**
> >
> > - **study** (`Study`) – Target study object.
> >
> > - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
> >
> > - **state** (`TrialState`) – Resulting trial state.
> >
> > - **values** (`Sequence[float] | None`) – Resulting trial values. Guaranteed to not be `None` if trial succeeded.
> >
> > **Return type**
> > > None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the `infer_relative_search_space()` call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> ---
>
> **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> ---
>
> > **Parameters**
> >
> > - **study** (`Study`) – Target study object.
> >
> > - **trial** (`FrozenTrial`) – Target trial object.
> >
> > **Return type**
> > > None

infer_relative_search_space(*study*, *trial*)

> Infer the search space that will be used by relative sampling in the target trial.
>
> This method is called right before `sample_relative()` method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.
>
> > **Parameters**
> >
> > - **study** (`Study`) – Target study object.
> >
> > - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
> >
> > **Returns**
> > > A dictionary containing the parameter names and parameter's distributions.

Return type
    Dict[str, *BaseDistribution*]

See also:

Please refer to `intersection_search_space()` as an implementation of `infer_relative_search_space()`.

reseed_rng()

Reseed sampler's random number generator.

This method is called by the *Study* instance if trials are executed in parallel with the option `n_jobs>1`. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

Return type
    None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by `sample_relative()` method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **param_name** (*str*) – Name of the sampled parameter.

- **param_distribution** (*BaseDistribution*) – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns
    A parameter value.

Return type
    Any

sample_relative(*study*, *trial*, *search_space*)

Sample parameters in a given search space.

This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **search_space** (*Dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.

Returns
> A dictionary containing the parameter names and the values.

Return type
> *Dict*[str, *Any*]

optuna.samplers.TPESampler

class optuna.samplers.**TPESampler**(*consider_prior=True*, *prior_weight=1.0*, *consider_magic_clip=True*, *consider_endpoints=False*, *n_startup_trials=10*, *n_ei_candidates=24*, *gamma=<function default_gamma>*, *weights=<function default_weights>*, *seed=None*, ***, *multivariate=False*, *group=False*, *warn_independent_sampling=True*, *constant_liar=False*, *constraints_func=None*, *categorical_distance_func=None*)

Sampler using TPE (Tree-structured Parzen Estimator) algorithm.

This sampler is based on *independent sampling*. See also *BaseSampler* for more details of 'independent sampling'.

On each trial, for each parameter, TPE fits one Gaussian Mixture Model (GMM) l(x) to the set of parameter values associated with the best objective values, and another GMM g(x) to the remaining parameter values. It chooses the parameter value x that maximizes the ratio l(x)/g(x).

For further information about TPE algorithm, please refer to the following papers:

- Algorithms for Hyper-Parameter Optimization

- Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures

- Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance

For multi-objective TPE (MOTPE), please refer to the following papers:

- Multiobjective Tree-Structured Parzen Estimator for Computationally Expensive Optimization Problems

- Multiobjective Tree-Structured Parzen Estimator

Example

An example of a single-objective optimization is as follows:

```python
import optuna
from optuna.samplers import TPESampler


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return x**2
```

(continues on next page)

```python
study = optuna.create_study(sampler=TPESampler())
study.optimize(objective, n_trials=10)
```

Note: *TPESampler* can handle a multi-objective task as well and the following shows an example:

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    f1 = x**2 + y
    f2 = -((x - 2) ** 2 + y)
    return f1, f2


# We minimize the first objective and maximize the second objective.
sampler = optuna.samplers.TPESampler()
study = optuna.create_study(directions=["minimize", "maximize"], sampler=sampler)
study.optimize(objective, n_trials=100)
```

Parameters

- **consider_prior** (*bool*) – Enhance the stability of Parzen estimator by imposing a Gaussian prior when True. The prior is only effective if the sampling distribution is either *FloatDistribution*, or *IntDistribution*.

- **prior_weight** (*float*) – The weight of the prior. This argument is used in *FloatDistribution*, *IntDistribution*, and *CategoricalDistribution*.

- **consider_magic_clip** (*bool*) – Enable a heuristic to limit the smallest variances of Gaussians used in the Parzen estimator.

- **consider_endpoints** (*bool*) – Take endpoints of domains into account when calculating variances of Gaussians in Parzen estimator. See the original paper for details on the heuristics to calculate the variances.

- **n_startup_trials** (*int*) – The random sampling is used instead of the TPE algorithm until the given number of trials finish in the same study.

- **n_ei_candidates** (*int*) – Number of candidate samples used to calculate the expected improvement.

- **gamma** (*Callable[[int], int]*) – A function that takes the number of finished trials and returns the number of trials to form a density function for samples with low grains. See the original paper for more details.

- **weights** (*Callable[[int], np.ndarray]*) – A function that takes the number of finished trials and returns a weight for them. See Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures for more details.

Note: In the multi-objective case, this argument is only used to compute the weights of bad

---

trials, i.e., trials to construct *g(x)* in the paper). The weights of good trials, i.e., trials to construct *l(x)*, are computed by a rule based on the hypervolume contribution proposed in the paper of MOTPE.

- **seed** (*Optional[int]*) – Seed for random number generator.

- **multivariate** (*bool*) – If this is True, the multivariate TPE is used when suggesting parameters. The multivariate TPE is reported to outperform the independent TPE. See BOHB: Robust and Efficient Hyperparameter Optimization at Scale for more details.

  Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

- **group** (*bool*) – If this and multivariate are True, the multivariate TPE with the group decomposed search space is used when suggesting parameters. The sampling algorithm decomposes the search space based on past trials and samples from the joint distribution in each decomposed subspace. The decomposed subspaces are a partition of the whole search space. Each subspace is a maximal subset of the whole search space, which satisfies the following: for a trial in completed trials, the intersection of the subspace and the search space of the trial becomes subspace itself or an empty set. Sampling from the joint distribution on the subspace is realized by multivariate TPE. If group is True, multivariate must be True as well.

  Note: Added in v2.8.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.8.0.

  Example:

  ```python
  import optuna


  def objective(trial):
      x = trial.suggest_categorical("x", ["A", "B"])
      if x == "A":
          return trial.suggest_float("y", -10, 10)
      else:
          return trial.suggest_int("z", -10, 10)


  sampler = optuna.samplers.TPESampler(multivariate=True, group=True)
  study = optuna.create_study(sampler=sampler)
  study.optimize(objective, n_trials=10)
  ```

- **warn_independent_sampling** (*bool*) – If this is True and multivariate=True, a warning message is emitted when the value of a parameter is sampled by using an independent sampler. If multivariate=False, this flag has no effect.

- **constant_liar** (*bool*) – If True, penalize running trials to avoid suggesting parameter configurations nearby.

  Note: Abnormally terminated trials often leave behind a record with a state of RUNNING in the storage. Such "zombie" trial parameters will be avoided by the constant liar algo-

---

rithm during subsequent sampling. When using an *RDBStorage*, it is possible to enable the
`heartbeat_interval` to change the records for abnormally terminated trials to `FAIL`.

Note: It is recommended to set this value to `True` during distributed optimization to avoid
having multiple workers evaluating similar parameter configurations. In particular, if each
objective function evaluation is costly and the durations of the running states are significant,
and/or the number of workers is high.

Note: Added in v2.8.0 as an experimental feature. The interface may change in newer
versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.8.0.

- `constraints_func` (*Optional[Callable[[FrozenTrial], Sequence[float]]]*) –
  An optional function that computes the objective constraints. It must take a *FrozenTrial*
  and return the constraints. The return value must be a sequence of `float` s. A value strictly
  larger than 0 means that a constraints is violated. A value equal to or smaller than 0 is
  considered feasible. If `constraints_func` returns more than one value for a trial, that trial
  is considered feasible if and only if all values are equal to 0 or smaller.

  The `constraints_func` will be evaluated after each successful trial. The function won't be
  called when trials fail or they are pruned, but this behavior is subject to change in the future
  releases.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer
versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

- `categorical_distance_func` (*Optional[dict[str,*
  Callable[[CategoricalChoiceType, CategoricalChoiceType], float]]])
  – A dictionary of distance functions for categorical parameters. The key is the
  name of the categorical parameter and the value is a distance function that takes two
  `CategoricalChoiceType` s and returns a `float` value. The distance function must return
  a non-negative value.

  While categorical choices are handled equally by default, this option allows users to specify
  prior knowledge on the structure of categorical parameters. When specified, categorical
  choices closer to current best choices are more likely to be sampled.

Note: Added in v3.4.0 as an experimental feature. The interface may change in newer
versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.4.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
hyperopt_parameters()	Return the the default parameters of hyperopt (v0.1.2).
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

> Trial post-processing.
>
> This method is called after the objective function returns and right before the trial is finished and its state is stored.
>
> ---
>
> **Note:** Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.
>
> ---
>
> **Parameters**
>
> - **study** (Study) – Target study object.
>
> - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>
> - **state** (TrialState) – Resulting trial state.
>
> - **values** (*Sequence[float]* | *None*) – Resulting trial values. Guaranteed to not be None if trial succeeded.
>
> **Return type**
> > None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> ---
>
> **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> ---
>
> **Parameters**
>
> - **study** (Study) – Target study object.
>
> - **trial** (FrozenTrial) – Target trial object.

> > **Return type**
> > None

> **static hyperopt_parameters()**

> > Return the the default parameters of hyperopt (v0.1.2).

> > *TPESampler* can be instantiated with the parameters returned by this method.

> > ### Example

> > Create a *TPESampler* instance with the default parameters of hyperopt.

> > ```python
> > import optuna
> > from optuna.samplers import TPESampler
> >
> >
> > def objective(trial):
> >   x = trial.suggest_float("x", -10, 10)
> >   return x**2
> >
> >
> > sampler = TPESampler(**TPESampler.hyperopt_parameters())
> > study = optuna.create_study(sampler=sampler)
> > study.optimize(objective, n_trials=10)
> > ```

> > **Returns**
> > A dictionary containing the default parameters of hyperopt.

> > **Return type**
> > *Dict*[str, *Any*]

> **infer_relative_search_space**(*study*, *trial*)

> > Infer the search space that will be used by relative sampling in the target trial.

> > This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

> > **Parameters**

> > - **study** (*Study*) – Target study object.

> > - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

> > **Returns**
> > A dictionary containing the parameter names and parameter's distributions.

> > **Return type**
> > *Dict*[str, *BaseDistribution*]

> **See also:**

> Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

> Reseed sampler's random number generator.
>
> This method is called by the *Study* instance if trials are executed in parallel with the option n_jobs>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.
>
> > **Return type**
> >> None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

> Sample a parameter for a given distribution.
>
> This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.
>
> ---
>
> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.
>
> ---
>
> > **Parameters**
> >
> > - **study** (*Study*) – Target study object.
> >
> > - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> >
> > - **param_name** (*str*) – Name of the sampled parameter.
> >
> > - **param_distribution** (*BaseDistribution*) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
> >
> > **Returns**
> >> A parameter value.
> >
> > **Return type**
> >> *Any*

sample_relative(*study*, *trial*, *search_space*)

> Sample parameters in a given search space.
>
> This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.
>
> ---
>
> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.
>
> ---
>
> > **Parameters**
> >
> > - **study** (*Study*) – Target study object.
> >
> > - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> >
> > - **search_space** (*Dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.

> **Returns**
>> A dictionary containing the parameter names and the values.
>
> **Return type**
>> *Dict*[str, *Any*]

optuna.samplers.CmaEsSampler

class optuna.samplers.**CmaEsSampler**(*x0=None*, *sigma0=None*, *n_startup_trials=1*,
independent_sampler=None, *warn_independent_sampling=True*,
seed=None, ***, *consider_pruned_trials=False*, *restart_strategy=None*,
popsize=None, *inc_popsize=2*, *use_separable_cma=False*,
with_margin=False, *lr_adapt=False*, *source_trials=None*)

A sampler using cmaes as the backend.

Example

Optimize a simple quadratic function by using *CmaEsSampler*.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    y = trial.suggest_int("y", -1, 1)
    return x**2 + y


sampler = optuna.samplers.CmaEsSampler()
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=20)
```

Please note that this sampler does not support CategoricalDistribution. However, *FloatDistribution* with step, (*suggest_float()*) and *IntDistribution* (*suggest_int()*) are supported.

If your search space contains categorical parameters, I recommend you to use *TPESampler* instead. Furthermore, there is room for performance improvements in parallel optimization settings. This sampler cannot use some trials for updating the parameters of multivariate normal distribution.

For further information about CMA-ES algorithm, please refer to the following papers:

- N. Hansen, The CMA Evolution Strategy: A Tutorial. arXiv:1604.00772, 2016.

- A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2005), pages 1769–1776. IEEE Press, 2005.

- N. Hansen. Benchmarking a BI-Population CMA-ES on the BBOB-2009 Function Testbed. GECCO Workshop, 2009.

- Raymond Ros, Nikolaus Hansen. A Simple Modification in CMA-ES Achieving Linear Time and Space Complexity. 10th International Conference on Parallel Problem Solving From Nature, Sep 2008, Dortmund, Germany. inria-00287367.

- Masahiro Nomura, Shuhei Watanabe, Youhei Akimoto, Yoshihiko Ozaki, Masaki Onishi. Warm Starting CMA-ES for Hyperparameter Optimization, AAAI. 2021.

- R. Hamano, S. Saito, M. Nomura, S. Shirakawa. CMA-ES with Margin: Lower-Bounding Marginal Probability for Mixed-Integer Black-Box Optimization, GECCO. 2022.

- M. Nomura, Y. Akimoto, I. Ono. CMA-ES with Learning Rate Adaptation: Can CMA-ES with Default Population Size Solve Multimodal and Noisy Problems?, GECCO. 2023.

See also:

You can also use optuna_integration.PyCmaSampler which is a sampler using cma library as the backend.

Parameters

- **x0** (`Optional[Dict[str, Any]]`) – A dictionary of an initial parameter values for CMA-ES. By default, the mean of `low` and `high` for each distribution is used. Note that `x0` is sampled uniformly within the search space domain for each restart if you specify `restart_strategy` argument.

- **sigma0** (`Optional[float]`) – Initial standard deviation of CMA-ES. By default, `sigma0` is set to `min_range / 6`, where `min_range` denotes the minimum range of the distributions in the search space.

- **seed** (`Optional[int]`) – A random seed for CMA-ES.

- **n_startup_trials** (`int`) – The independent sampling is used instead of the CMA-ES algorithm until the given number of trials finish in the same study.

- **independent_sampler** (`Optional[BaseSampler]`) – A *BaseSampler* instance that is used for independent sampling. The parameters not contained in the relative search space are sampled by this sampler. The search space for *CmaEsSampler* is determined by *intersection_search_space()*.

  If `None` is specified, *RandomSampler* is used as the default.

  See also:

  optuna.samplers module provides built-in independent samplers such as *RandomSampler* and *TPESampler*.

- **warn_independent_sampling** (`bool`) – If this is `True`, a warning message is emitted when the value of a parameter is sampled by using an independent sampler.

  Note that the parameters of the first trial in a study are always sampled via an independent sampler, so no warning messages are emitted in this case.

- **restart_strategy** (`Optional[str]`) – Strategy for restarting CMA-ES optimization when converges to a local minimum. If `None` is given, CMA-ES will not restart (default). If 'ipop' is given, CMA-ES will restart with increasing population size. if 'bipop' is given, CMA-ES will restart with the population size increased or decreased. Please see also `inc_popsize` parameter.

  Note: Added in v2.1.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.1.0.

- **popsize** (`Optional[int]`) – A population size of CMA-ES. When `restart_strategy = 'ipop'` or `restart_strategy = 'bipop'` is specified, this is used as the initial population size.

- **inc_popsize** (`int`) – Multiplier for increasing population size before each restart. This argument will be used when `restart_strategy = 'ipop'` or `restart_strategy = 'bipop'` is specified.

- **consider_pruned_trials** (*bool*) – If this is True, the PRUNED trials are considered for sampling.

  Note: Added in v2.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.0.0.

  Note: It is suggested to set this flag False when the *MedianPruner* is used. On the other hand, it is suggested to set this flag True when the *HyperbandPruner* is used. Please see the benchmark result for the details.

- **use_separable_cma** (*bool*) – If this is True, the covariance matrix is constrained to be diagonal. Due to reduce the model complexity, the learning rate for the covariance matrix is increased. Consequently, this algorithm outperforms CMA-ES on separable functions.

  Note: Added in v2.6.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.6.0.

- **with_margin** (*bool*) – If this is True, CMA-ES with margin is used. This algorithm prevents samples in each discrete distribution (*FloatDistribution* with *step* and *IntDistribution*) from being fixed to a single point. Currently, this option cannot be used with use_separable_cma=True.

  Note: Added in v3.1.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

- **lr_adapt** (*bool*) – If this is True, CMA-ES with learning rate adaptation is used. This algorithm focuses on working well on multimodal and/or noisy problems with default settings. Currently, this option cannot be used with use_separable_cma=True or with_margin=True.

  Note: Added in v3.3.0 or later, as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

- **source_trials** (*Optional[List[FrozenTrial]]*) – This option is for Warm Starting CMA-ES, a method to transfer prior knowledge on similar HPO tasks through the initialization of CMA-ES. This method estimates a promising distribution from source_trials and generates the parameter of multivariate gaussian distribution. Please note that it is prohibited to use x0, sigma0, or use_separable_cma argument together.

  Note: Added in v2.6.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.6.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

> Trial post-processing.
>
> This method is called after the objective function returns and right before the trial is finished and its state is stored.
>
> ---
> **Note:** Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.
>
> ---
>
> **Parameters**
>
> - **study** (*Study*) – Target study object.
>
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
>
> - **state** (*TrialState*) – Resulting trial state.
>
> - **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be *None* if trial succeeded.
>
> **Return type**
> > None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> ---
> **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> ---
>
> **Parameters**
>
> - **study** (*Study*) – Target study object.
>
> - **trial** (*FrozenTrial*) – Target trial object.
>
> **Return type**
> > None

infer_relative_search_space(*study*, *trial*)

Infer the search space that will be used by relative sampling in the target trial.

This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

> **Parameters**
>> • **study** (Study) – Target study object.
>>
>> • **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>
> **Returns**
>> A dictionary containing the parameter names and parameter's distributions.
>
> **Return type**
>> *Dict*[str, *BaseDistribution*]

> **See also:**
>
> Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

Reseed sampler's random number generator.

This method is called by the *Study* instance if trials are executed in parallel with the option `n_jobs`>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

> **Return type**
>> None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
>> • **study** (Study) – Target study object.
>>
>> • **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>>
>> • **param_name** (str) – Name of the sampled parameter.
>>
>> • **param_distribution** (BaseDistribution) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
>> A parameter value.
>
> **Return type**
>> *Any*

---

sample_relative(*study*, *trial*, *search_space*)

> Sample parameters in a given search space.
>
> This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
>
> - **study** (`Study`) – Target study object.
>
> - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
>
> - **search_space** (`Dict[str, BaseDistribution]`) – The search space returned by `infer_relative_search_space()`.
>
> **Returns**
>
> > A dictionary containing the parameter names and the values.
>
> **Return type**
>
> > *Dict*[str, *Any*]

optuna.samplers.GPSampler

class optuna.samplers.**GPSampler**(***, *seed=None*, *independent_sampler=None*, *n_startup_trials=10*, *deterministic_objective=False*)

> Sampler using Gaussian process-based Bayesian optimization.
>
> This sampler fits a Gaussian process (GP) to the objective function and optimizes the acquisition function to suggest the next parameters.
>
> **The current implementation uses:**
>
> - Matern kernel with nu=2.5 (twice differentiable),
>
> - Automatic relevance determination (ARD) for the length scale of each parameter,
>
> - Gamma prior for inverse squared lengthscales, kernel scale, and noise variance,
>
> - Log Expected Improvement (logEI) as the acquisition function, and
>
> - Quasi-Monte Carlo (QMC) sampling to optimize the acquisition function.

> **Note:** This sampler requires *scipy* and *pytorch*. You can install these dependencies with *pip install scipy pytorch*.

> **Parameters**
>
> - **seed** (`int` | *None*) – Random seed to initialize internal random number generator. Defaults to None (a seed is picked randomly).
>
> - **independent_sampler** (`BaseSampler` | *None*) – Sampler used for initial sampling (for the first *n_startup_trials* trials) and for conditional parameters. Defaults to *None* (a random sampler with the same *seed* is used).
>
> - **n_startup_trials** (`int`) – Number of initial trials. Defaults to 10.

- **deterministic_objective** (*bool*) – Whether the objective function is deterministic or not. If *True*, the sampler will fix the noise variance of the surrogate model to the minimum value (slightly above 0 to ensure numerical stability). Defaults to *False*.

Note: Added in v3.6.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.6.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

Trial post-processing.

This method is called after the objective function returns and right before the trial is finished and its state is stored.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **state** (*TrialState*) – Resulting trial state.

- **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be *None* if trial succeeded.

Return type

None

before_trial(*study*, *trial*)

Trial pre-processing.

This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

> **Parameters**
>
> - **study** (Study) – Target study object.
>
> - **trial** (FrozenTrial) – Target trial object.
>
> **Return type**
> None

infer_relative_search_space(*study*, *trial*)

> Infer the search space that will be used by relative sampling in the target trial.
>
> This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.
>
> > **Parameters**
> >
> > - **study** (Study) – Target study object.
> >
> > - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
> >
> > **Returns**
> > A dictionary containing the parameter names and parameter's distributions.
> >
> > **Return type**
> > dict[str, *BaseDistribution*]
>
> **See also:**
>
> Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

> Reseed sampler's random number generator.
>
> This method is called by the *Study* instance if trials are executed in parallel with the option n_jobs>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.
>
> > **Return type**
> > None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

> Sample a parameter for a given distribution.
>
> This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**

- **study** (Study) – Target study object.

- **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.

- **param_name** (*str*) – Name of the sampled parameter.

- **param_distribution** (*BaseDistribution*) – Distribution object that specifies a prior and/or scale of the sampling algorithm.

> **Returns**
> A parameter value.

> **Return type**
> *Any*

sample_relative(*study*, *trial*, *search_space*)

> Sample parameters in a given search space.

> This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**

- **study** (Study) – Target study object.

- **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.

- **search_space** (*dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.

> **Returns**
> A dictionary containing the parameter names and the values.

> **Return type**
> dict[str, *Any*]

optuna.samplers.PartialFixedSampler

class optuna.samplers.**PartialFixedSampler**(*fixed_params*, *base_sampler*)

> Sampler with partially fixed parameters.

Example

After several steps of optimization, you can fix the value of y and re-optimize it.

```
import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    y = trial.suggest_int("y", -1, 1)
```

```
    return x**2 + y


study = optuna.create_study()
study.optimize(objective, n_trials=10)

best_params = study.best_params
fixed_params = {"y": best_params["y"]}
partial_sampler = optuna.samplers.PartialFixedSampler(fixed_params, study.sampler)

study.sampler = partial_sampler
study.optimize(objective, n_trials=10)
```

Parameters

- **fixed_params** (`Dict[str, Any]`) – A dictionary of parameters to be fixed.
- **base_sampler** (`BaseSampler`) – A sampler which samples unfixed parameters.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

Trial post-processing.

This method is called after the objective function returns and right before the trial is finished and its state is stored.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Parameters

- **study** (`Study`) – Target study object.
- **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
- **state** (`TrialState`) – Resulting trial state.

- **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be None if trial succeeded.

> **Return type**
> None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> ---
>
> **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> ---
>
> **Parameters**
>
> - **study** (Study) – Target study object.
> - **trial** (FrozenTrial) – Target trial object.
>
> **Return type**
> None

infer_relative_search_space(*study*, *trial*)

> Infer the search space that will be used by relative sampling in the target trial.
>
> This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.
>
> **Parameters**
>
> - **study** (Study) – Target study object.
> - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>
> **Returns**
> A dictionary containing the parameter names and parameter's distributions.
>
> **Return type**
> *Dict*[str, *BaseDistribution*]

> **See also:**
>
> Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

> Reseed sampler's random number generator.
>
> This method is called by the *Study* instance if trials are executed in parallel with the option n_jobs>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.
>
> **Return type**
> None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

> Sample a parameter for a given distribution.
>
> This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.
>
> ---
>
> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.
>
> ---
>
> **Parameters**
>
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> - **param_name** (*str*) – Name of the sampled parameter.
> - **param_distribution** (*BaseDistribution*) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
>
> > A parameter value.
>
> **Return type**
>
> > *Any*

sample_relative(*study*, *trial*, *search_space*)

> Sample parameters in a given search space.
>
> This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.
>
> ---
>
> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.
>
> ---
>
> **Parameters**
>
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> - **search_space** (*Dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.
>
> **Returns**
>
> > A dictionary containing the parameter names and the values.
>
> **Return type**
>
> > *Dict*[str, *Any*]

optuna.samplers.NSGAIISampler

class optuna.samplers.**NSGAIISampler**(**, population_size=50, mutation_prob=None, crossover=None, crossover_prob=0.9, swapping_prob=0.5, seed=None, constraints_func=None, elite_population_selection_strategy=None, child_generation_strategy=None, after_trial_strategy=None*)

Multi-objective sampler using the NSGA-II algorithm.

NSGA-II stands for "Nondominated Sorting Genetic Algorithm II", which is a well known, fast and elitist multi-objective genetic algorithm.

For further information about NSGA-II, please refer to the following paper:

- A fast and elitist multiobjective genetic algorithm: NSGA-II

Parameters

- **population_size** (*int*) – Number of individuals (trials) in a generation. population_size must be greater than or equal to crossover.n_parents. For *UNDXCrossover* and *SPXCrossover*, n_parents=3, and for the other algorithms, n_parents=2.

- **mutation_prob** (*float | None*) – Probability of mutating each parameter when creating a new individual. If None is specified, the value 1.0 / len(parent_trial.params) is used where parent_trial is the parent trial of the target individual.

- **crossover** (*BaseCrossover | None*) – Crossover to be applied when creating child individuals. The available crossovers are listed here: https://optuna.readthedocs.io/en/stable/reference/samplers/nsgaii.html.

  UniformCrossover is always applied to parameters sampled from *CategoricalDistribution*, and by default for parameters sampled from other distributions unless this argument is specified.

  For more information on each of the crossover method, please refer to specific crossover documentation.

- **crossover_prob** (*float*) – Probability that a crossover (parameters swapping between parents) will occur when creating a new individual.

- **swapping_prob** (*float*) – Probability of swapping each parameter of the parents during crossover.

- **seed** (*int | None*) – Seed for random number generator.

- **constraints_func** (*Callable[[FrozenTrial], Sequence[float]] | None*) – An optional function that computes the objective constraints. It must take a *FrozenTrial* and return the constraints. The return value must be a sequence of float s. A value strictly larger than 0 means that a constraints is violated. A value equal to or smaller than 0 is considered feasible. If constraints_func returns more than one value for a trial, that trial is considered feasible if and only if all values are equal to 0 or smaller.

  The constraints_func will be evaluated after each successful trial. The function won't be called when trials fail or they are pruned, but this behavior is subject to change in the future releases.

  The constraints are handled by the constrained domination. A trial x is said to constrained-dominate a trial y, if any of the following conditions is true:

  1. Trial x is feasible and trial y is not.

  2. Trial x and y are both infeasible, but trial x has a smaller overall violation.

3. Trial x and y are feasible and trial x dominates trial y.

Note: Added in v2.5.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.5.0.

- **elite_population_selection_strategy**   (*Callable[[*Study, *list[FrozenTrial]]*, *list[FrozenTrial]] | None*) – The selection strategy for determining the individuals to survive from the current population pool. Default to None.

Note: The arguments `elite_population_selection_strategy` was added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

- **child_generation_strategy**   (*Callable[[*Study, *dict[str, BaseDistribution]*, *list[FrozenTrial]]*, *dict[str, Any]] | None*) – The strategy for generating child parameters from parent trials. Defaults to None.

Note: The arguments `child_generation_strategy` was added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

- **after_trial_strategy**   (*Callable[[*Study, FrozenTrial, TrialState, Sequence[float] | None]*, *None] | None*) – A set of procedure to be conducted after each trial. Defaults to None.

Note: The arguments `after_trial_strategy` was added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

　　Trial post-processing.

　　This method is called after the objective function returns and right before the trial is finished and its state is stored.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **state** (*TrialState*) – Resulting trial state.

- **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be None if trial succeeded.

Return type
    None

before_trial(*study*, *trial*)

Trial pre-processing.

This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object.

Return type
    None

infer_relative_search_space(*study*, *trial*)

Infer the search space that will be used by relative sampling in the target trial.

This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

Returns
    A dictionary containing the parameter names and parameter's distributions.

Return type
    dict[str, *BaseDistribution*]

See also:

Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

Reseed sampler's random number generator.

This method is called by the *Study* instance if trials are executed in parallel with the option `n_jobs>1`. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

> **Return type**
> None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
> - **param_name** (*str*) – Name of the sampled parameter.
> - **param_distribution** (*BaseDistribution*) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
> A parameter value.
>
> **Return type**
> *Any*

sample_relative(*study*, *trial*, *search_space*)

Sample parameters in a given search space.

This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
> - **study** (*Study*) – Target study object.
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **search_space** (`dict[str, BaseDistribution]`) – The search space returned by `infer_relative_search_space()`.

Returns
A dictionary containing the parameter names and the values.

Return type
dict[str, *Any*]

optuna.samplers.NSGAIIISampler

class optuna.samplers.**NSGAIIISampler**(*, *population_size=50*, *mutation_prob=None*, *crossover=None*, *crossover_prob=0.9*, *swapping_prob=0.5*, *seed=None*, *constraints_func=None*, *reference_points=None*, *dividing_parameter=3*, *elite_population_selection_strategy=None*, *child_generation_strategy=None*, *after_trial_strategy=None*)

Multi-objective sampler using the NSGA-III algorithm.

NSGA-III stands for "Nondominated Sorting Genetic Algorithm III", which is a modified version of NSGA-II for many objective optimization problem.

For further information about NSGA-III, please refer to the following papers:

- An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints

- An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach

Parameters

- **reference_points** (`np.ndarray | None`) – A 2 dimension `numpy.ndarray` with objective dimension columns. Represents a list of reference points which is used to determine who to survive. After non-dominated sort, who out of borderline front are going to survived is determined according to how sparse the closest reference point of each individual is. In the default setting the algorithm uses *uniformly* spread points to diversify the result. It is also possible to reflect your *preferences* by giving an arbitrary set of *target* points since the algorithm prioritizes individuals around reference points.

- **dividing_parameter** (`int`) – A parameter to determine the density of default reference points. This parameter determines how many divisions are made between reference points on each axis. The smaller this value is, the less reference points you have. The default value is 3. Note that this parameter is not used when `reference_points` is not `None`.

- **population_size** (`int`) –

- **mutation_prob** (`float | None`) –

- **crossover** (`BaseCrossover | None`) –

- **crossover_prob** (`float`) –

- **swapping_prob** (`float`) –

- **seed** (`int | None`) –

- **constraints_func** (`Callable[[FrozenTrial], Sequence[float]] | None`) –

- **elite_population_selection_strategy** (`Callable[[Study, list[FrozenTrial]], list[FrozenTrial]] | None`) –

- **child_generation_strategy** (*Callable[[Study*, *dict[str*, *BaseDistribution]*, *list[FrozenTrial]]*, *dict[str*, *Any]] | None*) –

- **after_trial_strategy** (*Callable[[Study*, *FrozenTrial*, *TrialState*, *Sequence[float] | None]*, *None] | None*) –

Note: Other parameters than `reference_points` and `dividing_parameter` are the same as *NSGAIISampler*.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

Trial post-processing.

This method is called after the objective function returns and right before the trial is finished and its state is stored.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **state** (*TrialState*) – Resulting trial state.

- **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be *None* if trial succeeded.

Return type

None

before_trial(*study*, *trial*)

Trial pre-processing.

This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

>   **Parameters**
>
>   - **study** (Study) – Target study object.
>
>   - **trial** (FrozenTrial) – Target trial object.
>
>   **Return type**
>   None

infer_relative_search_space(*study*, *trial*)

Infer the search space that will be used by relative sampling in the target trial.

This method is called right before *sample_relative()* method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

>   **Parameters**
>
>   - **study** (Study) – Target study object.
>
>   - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>
>   **Returns**
>   A dictionary containing the parameter names and parameter's distributions.
>
>   **Return type**
>   dict[str, *BaseDistribution*]

See also:

Please refer to *intersection_search_space()* as an implementation of *infer_relative_search_space()*.

reseed_rng()

Reseed sampler's random number generator.

This method is called by the *Study* instance if trials are executed in parallel with the option n_jobs>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

>   **Return type**
>   None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by *sample_relative()* method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
>
> - **study** (Study) – Target study object.
>
> - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>
> - **param_name** (str) – Name of the sampled parameter.
>
> - **param_distribution** (BaseDistribution) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
> > A parameter value.
>
> **Return type**
> > *Any*

sample_relative(*study*, *trial*, *search_space*)

> Sample parameters in a given search space.
>
> This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
>
> - **study** (Study) – Target study object.
>
> - **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.
>
> - **search_space** (dict[str, BaseDistribution]) – The search space returned by infer_relative_search_space().
>
> **Returns**
> > A dictionary containing the parameter names and the values.
>
> **Return type**
> > dict[str, *Any*]

optuna.samplers.QMCSampler

class optuna.samplers.**QMCSampler**(***, *qmc_type='sobol'*, *scramble=False*, *seed=None*, *independent_sampler=None*, *warn_asynchronous_seeding=True*, *warn_independent_sampling=True*)

> A Quasi Monte Carlo Sampler that generates low-discrepancy sequences.
>
> Quasi Monte Carlo (QMC) sequences are designed to have lower discrepancies than standard random sequences. They are known to perform better than the standard random sequences in hyperparameter optimization.

For further information about the use of QMC sequences for hyperparameter optimization, please refer to the following paper:

- Bergstra, James, and Yoshua Bengio. Random search for hyper-parameter optimization. Journal of machine learning research 13.2, 2012.

We use the QMC implementations in Scipy. For the details of the QMC algorithm, see the Scipy API references on scipy.stats.qmc.

Note: The search space of the sampler is determined by either previous trials in the study or the first trial that this sampler samples.

If there are previous trials in the study, *QMCSampler* infers its search space using the trial which was created first in the study.

Otherwise (if the study has no previous trials), *QMCSampler* samples the first trial using its *independent_sampler* and then infers the search space in the second trial.

As mentioned above, the search space of the *QMCSampler* is determined by the first trial of the study. Once the search space is determined, it cannot be changed afterwards.

Parameters

- **qmc_type** (*str*) – The type of QMC sequence to be sampled. This must be one of *"halton"* and *"sobol"*. Default is *"sobol"*.

    Note: Sobol' sequence is designed to have low-discrepancy property when the number of samples is $n = 2^m$ for each positive integer m. When it is possible to pre-specify the number of trials suggested by *QMCSampler*, it is recommended that the number of trials should be set as power of two.

- **scramble** (*bool*) – If this option is True, scrambling (randomization) is applied to the QMC sequences.

- **seed** (*int | None*) – A seed for QMCSampler. This argument is used only when scramble is True. If this is None, the seed is initialized randomly. Default is None.

    Note: When using multiple *QMCSampler*'s in parallel and/or distributed optimization, all the samplers must share the same seed when the *scrambling* is enabled. Otherwise, the low-discrepancy property of the samples will be degraded.

- **independent_sampler** (*BaseSampler | None*) – A *BaseSampler* instance that is used for independent sampling. The first trial of the study and the parameters not contained in the relative search space are sampled by this sampler.

    If None is specified, *RandomSampler* is used as the default.

    See also:

    samplers module provides built-in independent samplers such as *RandomSampler* and *TPESampler*.

- **warn_independent_sampling** (*bool*) – If this is True, a warning message is emitted when the value of a parameter is sampled by using an independent sampler.

Note that the parameters of the first trial in a study are sampled via an independent sampler in most cases, so no warning messages are emitted in such cases.

- **warn_asynchronous_seeding** (*bool*) – If this is `True`, a warning message is emitted when the scrambling (randomization) is applied to the QMC sequence and the random seed of the sampler is not set manually.

Note: When using parallel and/or distributed optimization without manually setting the seed, the seed is set randomly for each instances of *QMCSampler* for different workers, which ends up asynchronous seeding for multiple samplers used in the optimization.

See also:

See parameter `seed` in *QMCSampler*.

Example

Optimize a simple quadratic function by using *QMCSampler*.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    y = trial.suggest_int("y", -1, 1)
    return x**2 + y


sampler = optuna.samplers.QMCSampler()
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=8)
```

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

Trial post-processing.

This method is called after the objective function returns and right before the trial is finished and its state is stored.

> **Note:** Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

> **Parameters**
>
> - **study** (`Study`) – Target study object.
> - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
> - **state** (`TrialState`) – Resulting trial state.
> - **values** (`Sequence[float] | None`) – Resulting trial values. Guaranteed to not be `None` if trial succeeded.
>
> **Return type**
> > None

before_trial(*study*, *trial*)

> Trial pre-processing.
>
> This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the `infer_relative_search_space()` call. In other words, it is responsible for pre-processing that should be done before inferring the search space.
>
> > **Note:** Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.
>
> > **Parameters**
> >
> > - **study** (`Study`) – Target study object.
> > - **trial** (`FrozenTrial`) – Target trial object.
> >
> > **Return type**
> > > None

infer_relative_search_space(*study*, *trial*)

> Infer the search space that will be used by relative sampling in the target trial.
>
> This method is called right before `sample_relative()` method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.
>
> > **Parameters**
> >
> > - **study** (`Study`) – Target study object.
> > - **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
> >
> > **Returns**
> > > A dictionary containing the parameter names and parameter's distributions.
> >
> > **Return type**
> > > *Dict*[str, *BaseDistribution*]

See also:

Please refer to `intersection_search_space()` as an implementation of `infer_relative_search_space()`.

`reseed_rng()`

Reseed sampler's random number generator.

This method is called by the `Study` instance if trials are executed in parallel with the option `n_jobs>1`. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

> **Return type**
> None

`sample_independent`(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

This method is called only for the parameters not contained in the search space returned by `sample_relative()` method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
> * **study** (`Study`) – Target study object.
> * **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.
> * **param_name** (`str`) – Name of the sampled parameter.
> * **param_distribution** (`BaseDistribution`) – Distribution object that specifies a prior and/or scale of the sampling algorithm.
>
> **Returns**
> A parameter value.
>
> **Return type**
> *Any*

`sample_relative`(*study*, *trial*, *search_space*)

Sample parameters in a given search space.

This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
> * **study** (`Study`) – Target study object.
> * **trial** (`FrozenTrial`) – Target trial object. Take a copy before modifying this object.

- **search_space** (`Dict[str, BaseDistribution]`) – The search space returned by *infer_relative_search_space()*.

Returns
> A dictionary containing the parameter names and the values.

Return type
> *Dict*[str, *Any*]

optuna.samplers.BruteForceSampler

class optuna.samplers.**BruteForceSampler**(*seed=None*)

> Sampler using brute force.
>
> This sampler performs exhaustive search on the defined search space.

Example

```python
import optuna


def objective(trial):
    c = trial.suggest_categorical("c", ["float", "int"])
    if c == "float":
        return trial.suggest_float("x", 1, 3, step=0.5)
    elif c == "int":
        a = trial.suggest_int("a", 1, 3)
        b = trial.suggest_int("b", a, 3)
        return a + b


study = optuna.create_study(sampler=optuna.samplers.BruteForceSampler())
study.optimize(objective)
```

> **Note:** The defined search space must be finite. Therefore, when using *FloatDistribution* or *suggest_float()*, step=None is not allowed.

> **Note:** The sampler may fail to try the entire search space in when the suggestion ranges or parameters are changed in the same *Study*.

> **Parameters**
> > **seed** (`int | None`) – A seed to fix the order of trials as the search order randomly shuffled. Please note that it is not recommended using this option in distributed optimization settings since this option cannot ensure the order of trials and may increase the number of duplicate suggestions during distributed optimization.

> **Note:** Added in v3.1.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

after_trial(*study*, *trial*, *state*, *values*)

Trial post-processing.

This method is called after the objective function returns and right before the trial is finished and its state is stored.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **state** (*TrialState*) – Resulting trial state.

- **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be None if trial succeeded.

Return type

None

before_trial(*study*, *trial*)

Trial pre-processing.

This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the *infer_relative_search_space()* call. In other words, it is responsible for pre-processing that should be done before inferring the search space.

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

Parameters

- **study** (*Study*) – Target study object.

- **trial** (*FrozenTrial*) – Target trial object.

Return type

None

infer_relative_search_space(*study*, *trial*)

    Infer the search space that will be used by relative sampling in the target trial.

    This method is called right before `sample_relative()` method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.

        Parameters

- **study** (Study) – Target study object.

- **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.

        Returns

        A dictionary containing the parameter names and parameter's distributions.

        Return type

        Dict[str, *BaseDistribution*]

    See also:

    Please refer to `intersection_search_space()` as an implementation of `infer_relative_search_space()`.

reseed_rng()

    Reseed sampler's random number generator.

    This method is called by the `Study` instance if trials are executed in parallel with the option `n_jobs>1`. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

        Return type

        None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

    Sample a parameter for a given distribution.

    This method is called only for the parameters not contained in the search space returned by `sample_relative()` method. This method is suitable for sampling algorithms that do not use relationship between parameters such as random sampling and TPE.

    Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

        Parameters

- **study** (Study) – Target study object.

- **trial** (FrozenTrial) – Target trial object. Take a copy before modifying this object.

- **param_name** (str) – Name of the sampled parameter.

- **param_distribution** (BaseDistribution) – Distribution object that specifies a prior and/or scale of the sampling algorithm.

        Returns

        A parameter value.

        Return type

        Any

sample_relative(*study*, *trial*, *search_space*)

> Sample parameters in a given search space.
>
> This method is called once at the beginning of each trial, i.e., right before the evaluation of the objective function. This method is suitable for sampling algorithms that use relationship between parameters such as Gaussian Process and CMA-ES.

> **Note:** The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

> **Parameters**
>
> - **study** (*Study*) – Target study object.
>
> - **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
>
> - **search_space** (*Dict[str, BaseDistribution]*) – The search space returned by *infer_relative_search_space()*.
>
> **Returns**
>
> A dictionary containing the parameter names and the values.
>
> **Return type**
>
> *Dict*[str, *Any*]

optuna.samplers.IntersectionSearchSpace

class optuna.samplers.**IntersectionSearchSpace**(*include_pruned=False*)

> A class to calculate the intersection search space of a *Study*.
>
> Intersection search space contains the intersection of parameter distributions that have been suggested in the completed trials of the study so far. If there are multiple parameters that have the same name but different distributions, neither is included in the resulting search space (i.e., the parameters with dynamic value ranges are excluded).
>
> Note that an instance of this class is supposed to be used for only one study. If different studies are passed to *calculate()*, a ValueError is raised.
>
> > **Parameters**
> >
> > **include_pruned** (*bool*) – Whether pruned trials should be included in the search space.

> **Warning:** Deprecated in v3.2.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v4.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.2.0.
>
> Please use optuna.search_space.IntersectionSearchSpace instead.

Methods

calculate(study[, ordered_dict])	Returns the intersection search space of the *Study*.

calculate(*study*, *ordered_dict=False*)

> Returns the intersection search space of the *Study*.

> **Parameters**
>> - **study** (*Study*) – A study with completed trials. The same study must be passed for one instance of this class through its lifetime.
>> - **ordered_dict** (*bool*) – A boolean flag determining the return type. If `False`, the returned object will be a `dict`. If `True`, the returned object will be a `dict` sorted by keys, i.e. parameter names.

> **Returns**
>> A dictionary containing the parameter names and parameter's distributions.

> **Return type**
>> *Dict*[str, *BaseDistribution*]

optuna.samplers.intersection_search_space

optuna.samplers.**intersection_search_space**(*study*, *ordered_dict=False*, *include_pruned=False*)

> Return the intersection search space of the *Study*.

> Intersection search space contains the intersection of parameter distributions that have been suggested in the completed trials of the study so far. If there are multiple parameters that have the same name but different distributions, neither is included in the resulting search space (i.e., the parameters with dynamic value ranges are excluded).

> ---
> **Note:** *IntersectionSearchSpace* provides the same functionality with a much faster way. Please consider using it if you want to reduce execution time as much as possible.
> ---

> **Parameters**
>> - **study** (*Study*) – A study with completed trials.
>> - **ordered_dict** (*bool*) – A boolean flag determining the return type. If `False`, the returned object will be a `dict`. If `True`, the returned object will be a `dict` sorted by keys, i.e. parameter names.
>> - **include_pruned** (*bool*) – Whether pruned trials should be included in the search space.

> **Returns**
>> A dictionary containing the parameter names and parameter's distributions.

> **Return type**
>> *Dict*[str, *BaseDistribution*]

> **Warning:** Deprecated in v3.2.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v4.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.2.0.
>
> Please use optuna.search_space.intersection_search_space instead.

Note: The following `optuna.samplers.nsgaii` module defines crossover operations used by *NSGAIISampler*.

---

optuna.samplers.nsgaii

The `nsgaii` module defines crossover operations used by *NSGAIISampler*.

`optuna.samplers.nsgaii.BaseCrossover`	Base class for crossovers.
`optuna.samplers.nsgaii.UniformCrossover`	Uniform Crossover operation used by *NSGAIISampler*.
`optuna.samplers.nsgaii.BLXAlphaCrossover`	Blend Crossover operation used by *NSGAIISampler*.
`optuna.samplers.nsgaii.SPXCrossover`	Simplex Crossover operation used by *NSGAIISampler*.
`optuna.samplers.nsgaii.SBXCrossover`	Simulated Binary Crossover operation used by *NSGAIISampler*.
`optuna.samplers.nsgaii.VSBXCrossover`	Modified Simulated Binary Crossover operation used by *NSGAIISampler*.
`optuna.samplers.nsgaii.UNDXCrossover`	Unimodal Normal Distribution Crossover used by *NSGAIISampler*.

optuna.samplers.nsgaii.BaseCrossover

class optuna.samplers.nsgaii.**BaseCrossover**

Base class for crossovers.

A crossover operation is used by *NSGAIISampler* to create new parameter combination from parameters of n parent individuals.

Note: Concrete implementations of this class are expected to only accept parameters from numerical distributions. At the moment, only crossover operation for categorical parameters (uniform crossover) is built-in into *NSGAIISampler*.

Methods

crossover(parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents	Number of parent individuals required to perform crossover.

abstract crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

Perform crossover of selected parent individuals.

This method is called in *sample_relative()*.

> **Parameters**
>
> - **parents_params** (*ndarray*) – A `numpy.ndarray` with dimensions `num_parents x num_parameters`. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.
>
> - **rng** (*RandomState*) – An instance of `numpy.random.RandomState`.
>
> - **study** (*Study*) – Target study object.
>
> - **search_space_bounds** (*ndarray*) – A `numpy.ndarray` with dimensions `len_search_space x 2` representing numerical distribution bounds constructed from transformed search space.
>
> **Returns**
>
> A 1-dimensional `numpy.ndarray` containing new parameter combination.
>
> **Return type**
>
> *ndarray*

abstract property n_parents: int

Number of parent individuals required to perform crossover.

optuna.samplers.nsgaii.UniformCrossover

class optuna.samplers.nsgaii.**UniformCrossover**(*swapping_prob=0.5*)

Uniform Crossover operation used by *NSGAIISampler*.

Select each parameter with equal probability from the two parent individuals. For further information about uniform crossover, please refer to the following paper:

- Gilbert Syswerda. 1989. Uniform Crossover in Genetic Algorithms. In Proceedings of the 3rd International Conference on Genetic Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2-9.

> **Parameters**
>
> **swapping_prob** (*float*) – Probability of swapping each parameter of the parents during crossover.

Methods

[crossover](parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents

crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

> Perform crossover of selected parent individuals.
>
> This method is called in [sample_relative()](#).
>
> > **Parameters**
> >
> > - **parents_params** ([*ndarray*](#)) – A numpy.ndarray with dimensions num_parents x num_parameters. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.
> > - **rng** ([*RandomState*](#)) – An instance of numpy.random.RandomState.
> > - **study** ([*Study*](#)) – Target study object.
> > - **search_space_bounds** ([*ndarray*](#)) – A numpy.ndarray with dimensions len_search_space x 2 representing numerical distribution bounds constructed from transformed search space.
> >
> > **Returns**
> > A 1-dimensional numpy.ndarray containing new parameter combination.
> >
> > **Return type**
> > [*ndarray*](#)

optuna.samplers.nsgaii.BLXAlphaCrossover

class optuna.samplers.nsgaii.**BLXAlphaCrossover**(*alpha=0.5*)

> Blend Crossover operation used by [*NSGAIISampler*](#).
>
> Uniformly samples child individuals from the hyper-rectangles created by the two parent individuals. For further information about BLX-alpha crossover, please refer to the following paper:
>
> - [Eshelman, L. and J. D. Schaffer. Real-Coded Genetic Algorithms and Interval-Schemata. FOGA (1992).](#)
>
> > **Parameters**
> > **alpha** ([*float*](#)) – Parametrizes blend operation.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Methods

[crossover](parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents

crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

Perform crossover of selected parent individuals.

This method is called in [sample_relative()](.).

> **Parameters**
> - **parents_params** ([ndarray](.)) – A numpy.ndarray with dimensions num_parents x num_parameters. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.
> - **rng** ([RandomState](.)) – An instance of numpy.random.RandomState.
> - **study** ([Study](.)) – Target study object.
> - **search_space_bounds** ([ndarray](.)) – A numpy.ndarray with dimensions len_search_space x 2 representing numerical distribution bounds constructed from transformed search space.
>
> **Returns**
> A 1-dimensional numpy.ndarray containing new parameter combination.
>
> **Return type**
> [ndarray](.)

optuna.samplers.nsgaii.SPXCrossover

class optuna.samplers.nsgaii.**SPXCrossover**(*epsilon=None*)

Simplex Crossover operation used by [NSGAIISampler](.).

Uniformly samples child individuals from within a single simplex that is similar to the simplex produced by the parent individual. For further information about SPX crossover, please refer to the following paper:

- [Shigeyoshi Tsutsui and Shigeyoshi Tsutsui and David E. Goldberg and David E. Goldberg and Kumara Sastry and Kumara Sastry Progress Toward Linkage Learning in Real-Coded GAs with Simplex Crossover. IlliGAL Report. 2000.](.)

> **Parameters**
> **epsilon** ([float](.) | None) – Expansion rate. If not specified, defaults to sqrt(len(search_space) + 2).

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Methods

[crossover](parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents

crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

 Perform crossover of selected parent individuals.

 This method is called in [sample_relative()](#).

  Parameters

   • **parents_params** ([ndarray](#)) – A numpy.ndarray with dimensions num_parents x num_parameters. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.

   • **rng** ([RandomState](#)) – An instance of numpy.random.RandomState.

   • **study** ([Study](#)) – Target study object.

   • **search_space_bounds** ([ndarray](#)) – A numpy.ndarray with dimensions len_search_space x 2 representing numerical distribution bounds constructed from transformed search space.

  Returns

   A 1-dimensional numpy.ndarray containing new parameter combination.

  Return type

   ndarray

optuna.samplers.nsgaii.SBXCrossover

class optuna.samplers.nsgaii.**SBXCrossover**(*eta=None*)

 Simulated Binary Crossover operation used by [NSGAIISampler](#).

 Generates a child from two parent individuals according to the polynomial probability distribution.

  • Deb, K. and R. Agrawal. "Simulated Binary Crossover for Continuous Search Space." Complex Syst. 9 (1995): n. pag.

  Parameters

   eta ([float](#) | *None*) – Distribution index. A small value of eta allows distant solutions to be selected as children solutions. If not specified, takes default value of 2 for single objective functions and 20 for multi objective.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Methods

[crossover](#)(parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents

crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

>   Perform crossover of selected parent individuals.
>
>   This method is called in [sample_relative()](#).
>
> >   **Parameters**
> >
> >   - **parents_params** (*ndarray*) – A `numpy.ndarray` with dimensions `num_parents x num_parameters`. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.
> >
> >   - **rng** (*RandomState*) – An instance of `numpy.random.RandomState`.
> >
> >   - **study** (*Study*) – Target study object.
> >
> >   - **search_space_bounds** (*ndarray*) – A `numpy.ndarray` with dimensions `len_search_space x 2` representing numerical distribution bounds constructed from transformed search space.
> >
> >   **Returns**
> >
> >   A 1-dimensional `numpy.ndarray` containing new parameter combination.
> >
> >   **Return type**
> >
> >   *ndarray*

optuna.samplers.nsgaii.VSBXCrossover

class optuna.samplers.nsgaii.**VSBXCrossover**(*eta=None*)

>   Modified Simulated Binary Crossover operation used by [NSGAIISampler](#).
>
>   vSBX generates child individuals without excluding any region of the parameter space, while maintaining the excellent properties of SBX.
>
>   - [Pedro J. Ballester, Jonathan N. Carter. Real-Parameter Genetic Algorithms for Finding Multiple Optimal Solutions in Multi-modal Optimization. GECCO 2003: 706-717](#)
>
> >   **Parameters**
> >
> >   **eta** (*float | None*) – Distribution index. A small value of `eta` allows distant solutions to be selected as children solutions. If not specified, takes default value of `2` for single objective functions and `20` for multi objective.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Methods

[crossover](parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents

crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

> Perform crossover of selected parent individuals.
>
> This method is called in [sample_relative()](#).
>
> > **Parameters**
> >
> > - **parents_params** ([ndarray](#)) – A `numpy.ndarray` with dimensions `num_parents x num_parameters`. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.
> >
> > - **rng** ([RandomState](#)) – An instance of `numpy.random.RandomState`.
> >
> > - **study** ([Study](#)) – Target study object.
> >
> > - **search_space_bounds** ([ndarray](#)) – A `numpy.ndarray` with dimensions `len_search_space x 2` representing numerical distribution bounds constructed from transformed search space.
> >
> > **Returns**
> >   A 1-dimensional `numpy.ndarray` containing new parameter combination.
> >
> > **Return type**
> >   *[ndarray](#)*

optuna.samplers.nsgaii.UNDXCrossover

class optuna.samplers.nsgaii.**UNDXCrossover**(*sigma_xi=0.5*, *sigma_eta=None*)

> Unimodal Normal Distribution Crossover used by [NSGAIISampler](#).
>
> Generates child individuals from the three parents using a multivariate normal distribution.
>
> - [H. Kita, I. Ono and S. Kobayashi, Multi-parental extension of the unimodal normal distribution crossover for real-coded genetic algorithms, Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406), 1999, pp. 1581-1588 Vol. 2](#)
>
> > **Parameters**
> >
> > - **sigma_xi** ([float](#)) – Parametrizes normal distribution from which `xi` is drawn.
> >
> > - **sigma_eta** ([float](#) | None) – Parametrizes normal distribution from which `etas` are drawn. If not specified, defaults to `0.35 / sqrt(len(search_space))`.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Methods

crossover(parents_params, rng, study, ...)	Perform crossover of selected parent individuals.

Attributes

n_parents

crossover(*parents_params*, *rng*, *study*, *search_space_bounds*)

> Perform crossover of selected parent individuals.
>
> This method is called in *sample_relative()*.
>
> > **Parameters**
> >
> > - **parents_params** (*ndarray*) – A numpy.ndarray with dimensions num_parents x num_parameters. Represents a parameter space for each parent individual. This space is continuous for numerical parameters.
> >
> > - **rng** (*RandomState*) – An instance of numpy.random.RandomState.
> >
> > - **study** (*Study*) – Target study object.
> >
> > - **search_space_bounds** (*ndarray*) – A numpy.ndarray with dimensions len_search_space x 2 representing numerical distribution bounds constructed from transformed search space.
> >
> > **Returns**
> > A 1-dimensional numpy.ndarray containing new parameter combination.
> >
> > **Return type**
> > *ndarray*

7.3.11 optuna.search_space

The *search_space* module provides functionality for controlling search space of parameters.

optuna.search_space. *IntersectionSearchSpace*	A class to calculate the intersection search space of a *Study*.
optuna.search_space. *intersection_search_space*	Return the intersection search space of the given trials.

---

optuna.search_space.IntersectionSearchSpace

class optuna.search_space.**IntersectionSearchSpace**(*include_pruned=False*)

A class to calculate the intersection search space of a *Study*.

Intersection search space contains the intersection of parameter distributions that have been suggested in the completed trials of the study so far. If there are multiple parameters that have the same name but different distributions, neither is included in the resulting search space (i.e., the parameters with dynamic value ranges are excluded).

Note that an instance of this class is supposed to be used for only one study. If different studies are passed to *calculate()*, a `ValueError` is raised.

> **Parameters**
> **include_pruned** (*bool*) – Whether pruned trials should be included in the search space.

Methods

calculate(study)	Returns the intersection search space of the *Study*.

calculate(*study*)

Returns the intersection search space of the *Study*.

> **Parameters**
> **study** (*Study*) – A study with completed trials. The same study must be passed for one instance of this class through its lifetime.
>
> **Returns**
> A dictionary containing the parameter names and parameter's distributions sorted by parameter names.
>
> **Return type**
> *Dict*[str, *BaseDistribution*]

optuna.search_space.intersection_search_space

optuna.search_space.**intersection_search_space**(*trials*, *include_pruned=False*)

Return the intersection search space of the given trials.

Intersection search space contains the intersection of parameter distributions that have been suggested in the completed trials of the study so far. If there are multiple parameters that have the same name but different distributions, neither is included in the resulting search space (i.e., the parameters with dynamic value ranges are excluded).

Note: *IntersectionSearchSpace* provides the same functionality with a much faster way. Please consider using it if you want to reduce execution time as much as possible.

> **Parameters**
>
> - **trials** (*list[FrozenTrial]*) – A list of trials.
>
> - **include_pruned** (*bool*) – Whether pruned trials should be included in the search space.

Returns
> A dictionary containing the parameter names and parameter's distributions sorted by parameter names.

Return type
> *Dict*[str, *BaseDistribution*]

7.3.12 optuna.storages

The `storages` module defines a `BaseStorage` class which abstracts a backend database and provides library-internal interfaces to the read/write histories of the studies and trials. Library users who wish to use storage solutions other than the default in-memory storage should use one of the child classes of `BaseStorage` documented below.

`optuna.storages.RDBStorage`	Storage class for RDB backend.
`optuna.storages.RetryFailedTrialCallback`	Retry a failed trial up to a maximum number of times.
`optuna.storages.fail_stale_trials`	Fail stale trials and run their failure callbacks.
`optuna.storages.JournalStorage`	Storage class for Journal storage backend.
`optuna.storages.JournalFileStorage`	File storage class for Journal log backend.
`optuna.storages.JournalFileSymlinkLock`	Lock class for synchronizing processes for NFSv2 or later.
`optuna.storages.JournalFileOpenLock`	Lock class for synchronizing processes for NFSv3 or later.
`optuna.storages.JournalRedisStorage`	Redis storage class for Journal log backend.

optuna.storages.RDBStorage

class optuna.storages.**RDBStorage**(*url*, *engine_kwargs=None*, *skip_compatibility_check=False*, *,
> *heartbeat_interval=None*, *grace_period=None*,
> *failed_trial_callback=None*, *skip_table_creation=False*)

Storage class for RDB backend.

Note that library users can instantiate this class, but the attributes provided by this class are not supposed to be directly accessed by them.

Example

Create an *RDBStorage* instance with customized `pool_size` and `timeout` settings.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    return x**2


storage = optuna.storages.RDBStorage(
    url="sqlite:///:memory:",
    engine_kwargs={"pool_size": 20, "connect_args": {"timeout": 10}},
)
```

```
study = optuna.create_study(storage=storage)
study.optimize(objective, n_trials=10)
```

> **Parameters**
>
> - **url** (`str`) – URL of the storage.
>
> - **engine_kwargs** (`Optional[Dict[str, Any]]`) – A dictionary of keyword arguments that is passed to sqlalchemy.engine.create_engine function.
>
> - **skip_compatibility_check** (`bool`) – Flag to skip schema compatibility check if set to `True`.
>
> - **heartbeat_interval** (`Optional[int]`) – Interval to record the heartbeat. It is recorded every `interval` seconds. `heartbeat_interval` must be `None` or a positive integer.
>
>   ---
>
>   **Note:** The heartbeat is supposed to be used with *optimize()*. If you use *ask()* and *tell()* instead, it will not work.
>
>   ---
>
> - **grace_period** (`Optional[int]`) – Grace period before a running trial is failed from the last heartbeat. `grace_period` must be `None` or a positive integer. If it is `None`, the grace period will be *2 * heartbeat_interval*.
>
> - **failed_trial_callback** (`Optional[Callable[['optuna.study.Study', FrozenTrial], None]]`) – A callback function that is invoked after failing each stale trial. The function must accept two parameters with the following types in this order: *Study* and *FrozenTrial*.
>
>   ---
>
>   **Note:** The procedure to fail existing stale trials is called just before asking the study for a new trial.
>
>   ---
>
> - **skip_table_creation** (`bool`) – Flag to skip table creation if set to `True`.

Note: If you use MySQL, pool_pre_ping will be set to `True` by default to prevent connection timeout. You can turn it off with `engine_kwargs['pool_pre_ping']=False`, but it is recommended to keep the setting if execution time of your objective function is longer than the *wait_timeout* of your MySQL configuration.

Note: We would never recommend SQLite3 for parallel optimization. Please see the FAQ *How can I solve the error that occurs when performing parallel optimization with SQLite3?* for details.

Note: Mainly in a cluster environment, running trials are often killed unexpectedly. If you want to detect a failure of trials, please use the heartbeat mechanism. Set `heartbeat_interval`, `grace_period`, and `failed_trial_callback` appropriately according to your use case. For more details, please refer to the *tutorial* and Example page.

See also:

You can use *RetryFailedTrialCallback* to automatically retry failed trials detected by heartbeat.

Methods

check_trial_is_updatable(trial_id, trial_state)	Check whether a trial state is updatable.
create_new_study(directions[, study_name])	Create a new study from a name.
create_new_trial(study_id[, template_trial])	Create and add a new trial to a study.
delete_study(study_id)	Delete a study.
get_all_studies()	Read a list of `FrozenStudy` objects.
get_all_trials(study_id[, deepcopy, states])	Read all trials in a study.
get_all_versions()	Return the schema version list.
get_best_trial(study_id)	Return the trial with the best value in a study.
get_current_version()	Return the schema version currently used by this storage.
get_failed_trial_callback()	Get the failed trial callback function.
get_head_version()	Return the latest schema version.
get_heartbeat_interval()	Get the heartbeat interval if it is set.
get_n_trials(study_id[, state])	Count the number of trials in a study.
get_study_directions(study_id)	Read whether a study maximizes or minimizes an objective.
get_study_id_from_name(study_name)	Read the ID of a study.
get_study_name_from_id(study_id)	Read the study name of a study.
get_study_system_attrs(study_id)	Read the optuna-internal attributes of a study.
get_study_user_attrs(study_id)	Read the user-defined attributes of a study.
get_trial(trial_id)	Read a trial.
get_trial_id_from_study_id_trial_number(...	Read the trial ID of a trial.
get_trial_number_from_id(trial_id)	Read the trial number of a trial.
get_trial_param(trial_id, param_name)	Read the parameter of a trial.
get_trial_params(trial_id)	Read the parameter dictionary of a trial.
get_trial_system_attrs(trial_id)	Read the optuna-internal attributes of a trial.
get_trial_user_attrs(trial_id)	Read the user-defined attributes of a trial.
record_heartbeat(trial_id)	Record the heartbeat of the trial.
remove_session()	Removes the current session.
set_study_system_attr(study_id, key, value)	Register an optuna-internal attribute to a study.
set_study_user_attr(study_id, key, value)	Register a user-defined attribute to a study.
set_trial_intermediate_value(trial_id, step, ...)	Report an intermediate value of an objective function.
set_trial_param(trial_id, param_name, ...)	Set a parameter to a trial.
set_trial_state_values(trial_id, state[, values])	Update the state and values of a trial.
set_trial_system_attr(trial_id, key, value)	Set an optuna-internal attribute to a trial.
set_trial_user_attr(trial_id, key, value)	Set a user-defined attribute to a trial.
upgrade()	Upgrade the storage schema.

check_trial_is_updatable(*trial_id*, *trial_state*)

> Check whether a trial state is updatable.

> > **Parameters**
> >
> > - **trial_id** (*int*) – ID of the trial. Only used for an error message.
> >
> > - **trial_state** (`TrialState`) – Trial state to check.
> >
> > **Raises**
> > `RuntimeError` – If the trial is already finished.
> >
> > **Return type**
> > None

create_new_study(*directions*, *study_name=None*)

> Create a new study from a name.
>
> If no name is specified, the storage class generates a name. The returned study ID is unique among all current and deleted studies.
>
> > **Parameters**
> >
> > - **directions** (*Sequence[*StudyDirection*]*) – A sequence of direction whose element is either *MAXIMIZE* or *MINIMIZE*.
> >
> > - **study_name** (*str | None*) – Name of the new study to create.
> >
> > **Returns**
> > > ID of the created study.
> >
> > **Raises**
> > > *optuna.exceptions.DuplicatedStudyError* – If a study with the same study_name already exists.
> >
> > **Return type**
> > > int

create_new_trial(*study_id*, *template_trial=None*)

> Create and add a new trial to a study.
>
> The returned trial ID is unique among all current and deleted trials.
>
> > **Parameters**
> >
> > - **study_id** (*int*) – ID of the study.
> >
> > - **template_trial** (*FrozenTrial | None*) – Template *FrozenTrial* with default user-attributes, system-attributes, intermediate-values, and a state.
> >
> > **Returns**
> > > ID of the created trial.
> >
> > **Raises**
> > > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> > > int

delete_study(*study_id*)

> Delete a study.
>
> > **Parameters**
> > > **study_id** (*int*) – ID of the study.
> >
> > **Raises**
> > > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> > > None

get_all_studies()

> Read a list of FrozenStudy objects.
>
> > **Returns**
> > > A list of FrozenStudy objects, sorted by study_id.
> >
> > **Return type**
> > > *List*[*FrozenStudy*]

get_all_trials(*study_id*, *deepcopy=True*, *states=None*)

Read all trials in a study.

> **Parameters**
>
> - **study_id** (*int*) – ID of the study.
>
> - **deepcopy** (*bool*) – Whether to copy the list of trials before returning. Set to True if you intend to update the list or elements of the list.
>
> - **states** (*Container[TrialState] | None*) – Trial states to filter on. If None, include all states.
>
> **Returns**
>
> List of trials in the study, sorted by `trial_id`.
>
> **Raises**
>
> **KeyError** – If no study with the matching `study_id` exists.
>
> **Return type**
>
> *List*[FrozenTrial]

get_all_versions()

Return the schema version list.

> **Return type**
>
> *List*[str]

get_best_trial(*study_id*)

Return the trial with the best value in a study.

This method is valid only during single-objective optimization.

> **Parameters**
>
> **study_id** (*int*) – ID of the study.
>
> **Returns**
>
> The trial with the best objective value among all finished trials in the study.
>
> **Raises**
>
> - **KeyError** – If no study with the matching `study_id` exists.
>
> - **RuntimeError** – If the study has more than one direction.
>
> - **ValueError** – If no trials have been completed.
>
> **Return type**
>
> FrozenTrial

get_current_version()

Return the schema version currently used by this storage.

> **Return type**
>
> str

get_failed_trial_callback()

Get the failed trial callback function.

> **Returns**
>
> The failed trial callback function if it is set, otherwise None.
>
> **Return type**
>
> *Callable*[[Study, FrozenTrial], None] | None

get_head_version()

Return the latest schema version.

> **Return type**
> > [str](#)

get_heartbeat_interval()

Get the heartbeat interval if it is set.

> **Returns**
> > The heartbeat interval if it is set, otherwise [None](#).
>
> **Return type**
> > [int](#) | None

get_n_trials(*study_id*, *state=None*)

Count the number of trials in a study.

> **Parameters**
> > - **study_id** ([int](#)) – ID of the study.
> > - **state** (*Tuple[*[TrialState](#)*, ...]* | [TrialState](#) | *None*) – Trial states to filter on. If [None](#), include all states.
>
> **Returns**
> > Number of trials in the study.
>
> **Raises**
> > [KeyError](#) – If no study with the matching study_id exists.
>
> **Return type**
> > [int](#)

get_study_directions(*study_id*)

Read whether a study maximizes or minimizes an objective.

> **Parameters**
> > **study_id** ([int](#)) – ID of a study.
>
> **Returns**
> > Optimization directions list of the study.
>
> **Raises**
> > [KeyError](#) – If no study with the matching study_id exists.
>
> **Return type**
> > [List](#)[StudyDirection]

get_study_id_from_name(*study_name*)

Read the ID of a study.

> **Parameters**
> > **study_name** ([str](#)) – Name of the study.
>
> **Returns**
> > ID of the study.
>
> **Raises**
> > [KeyError](#) – If no study with the matching study_name exists.
>
> **Return type**
> > [int](#)

get_study_name_from_id(*study_id*)

Read the study name of a study.

> **Parameters**
> **study_id** (*int*) – ID of the study.
>
> **Returns**
> Name of the study.
>
> **Raises**
> [KeyError](#) – If no study with the matching study_id exists.
>
> **Return type**
> [str](#)

get_study_system_attrs(*study_id*)

Read the optuna-internal attributes of a study.

> **Parameters**
> **study_id** (*int*) – ID of the study.
>
> **Returns**
> Dictionary with the optuna-internal attributes of the study.
>
> **Raises**
> [KeyError](#) – If no study with the matching study_id exists.
>
> **Return type**
> *[Dict](#)*[str, *Any*]

get_study_user_attrs(*study_id*)

Read the user-defined attributes of a study.

> **Parameters**
> **study_id** (*int*) – ID of the study.
>
> **Returns**
> Dictionary with the user attributes of the study.
>
> **Raises**
> [KeyError](#) – If no study with the matching study_id exists.
>
> **Return type**
> *[Dict](#)*[str, *Any*]

get_trial(*trial_id*)

Read a trial.

> **Parameters**
> **trial_id** (*int*) – ID of the trial.
>
> **Returns**
> Trial with a matching trial ID.
>
> **Raises**
> [KeyError](#) – If no trial with the matching trial_id exists.
>
> **Return type**
> [FrozenTrial](#)

get_trial_id_from_study_id_trial_number(*study_id*, *trial_number*)

Read the trial ID of a trial.

> **Parameters**
>
> - **study_id** (*int*) – ID of the study.
>
> - **trial_number** (*int*) – Number of the trial.
>
> **Returns**
>   ID of the trial.
>
> **Raises**
>   **KeyError** – If no trial with the matching study_id and trial_number exists.
>
> **Return type**
>   int

get_trial_number_from_id(*trial_id*)

> Read the trial number of a trial.
>
> ---
>
> **Note:** The trial number is only unique within a study, and is sequential.
>
> ---
>
> **Parameters**
>   **trial_id** (*int*) – ID of the trial.
>
> **Returns**
>   Number of the trial.
>
> **Raises**
>   **KeyError** – If no trial with the matching trial_id exists.
>
> **Return type**
>   int

get_trial_param(*trial_id*, *param_name*)

> Read the parameter of a trial.
>
> **Parameters**
>
> - **trial_id** (*int*) – ID of the trial.
>
> - **param_name** (*str*) – Name of the parameter.
>
> **Returns**
>   Internal representation of the parameter.
>
> **Raises**
>   **KeyError** – If no trial with the matching trial_id exists. If no such parameter exists.
>
> **Return type**
>   float

get_trial_params(*trial_id*)

> Read the parameter dictionary of a trial.
>
> **Parameters**
>   **trial_id** (*int*) – ID of the trial.
>
> **Returns**
>   Dictionary of a parameters. Keys are parameter names and values are internal representations
>   of the parameter values.

> **Raises**
> > **KeyError** – If no trial with the matching `trial_id` exists.
>
> **Return type**
> > *Dict*[str, *Any*]

get_trial_system_attrs(*trial_id*)

> Read the optuna-internal attributes of a trial.
>
> > **Parameters**
> > > **trial_id** (*int*) – ID of the trial.
> >
> > **Returns**
> > > Dictionary with the optuna-internal attributes of the trial.
> >
> > **Raises**
> > > **KeyError** – If no trial with the matching `trial_id` exists.
> >
> > **Return type**
> > > *Dict*[str, *Any*]

get_trial_user_attrs(*trial_id*)

> Read the user-defined attributes of a trial.
>
> > **Parameters**
> > > **trial_id** (*int*) – ID of the trial.
> >
> > **Returns**
> > > Dictionary with the user-defined attributes of the trial.
> >
> > **Raises**
> > > **KeyError** – If no trial with the matching `trial_id` exists.
> >
> > **Return type**
> > > *Dict*[str, *Any*]

record_heartbeat(*trial_id*)

> Record the heartbeat of the trial.
>
> > **Parameters**
> > > **trial_id** (*int*) – ID of the trial.
> >
> > **Return type**
> > > None

remove_session()

> Removes the current session.
>
> A session is stored in SQLAlchemy's ThreadLocalRegistry for each thread. This method closes and removes the session which is associated to the current thread. Particularly, under multi-thread use cases, it is important to call this method *from each thread*. Otherwise, all sessions and their associated DB connections are destructed by a thread that occasionally invoked the garbage collector. By default, it is not allowed to touch a SQLite connection from threads other than the thread that created the connection. Therefore, we need to explicitly close the connection from each thread.
>
> > **Return type**
> > > None

set_study_system_attr(*study_id*, *key*, *value*)

> Register an optuna-internal attribute to a study.
>
> This method overwrites any existing attribute.

> **Parameters**
>
> - **study_id** (*int*) – ID of the study.
>
> - **key** (*str*) – Attribute key.
>
> - **value** (*Mapping[str, JSONSerializable] | Sequence[JSONSerializable] | str | int | float | bool | None*) – Attribute value. It should be JSON serializable.
>
> **Raises**
>   **KeyError** – If no study with the matching study_id exists.
>
> **Return type**
>   None

set_study_user_attr(*study_id*, *key*, *value*)

> Register a user-defined attribute to a study.
>
> This method overwrites any existing attribute.
>
> **Parameters**
>
> - **study_id** (*int*) – ID of the study.
>
> - **key** (*str*) – Attribute key.
>
> - **value** (*Any*) – Attribute value. It should be JSON serializable.
>
> **Raises**
>   **KeyError** – If no study with the matching study_id exists.
>
> **Return type**
>   None

set_trial_intermediate_value(*trial_id*, *step*, *intermediate_value*)

> Report an intermediate value of an objective function.
>
> This method overwrites any existing intermediate value associated with the given step.
>
> **Parameters**
>
> - **trial_id** (*int*) – ID of the trial.
>
> - **step** (*int*) – Step of the trial (e.g., the epoch when training a neural network).
>
> - **intermediate_value** (*float*) – Intermediate value corresponding to the step.
>
> **Raises**
>
> - **KeyError** – If no trial with the matching trial_id exists.
>
> - **RuntimeError** – If the trial is already finished.
>
> **Return type**
>   None

set_trial_param(*trial_id*, *param_name*, *param_value_internal*, *distribution*)

> Set a parameter to a trial.
>
> **Parameters**
>
> - **trial_id** (*int*) – ID of the trial.
>
> - **param_name** (*str*) – Name of the parameter.
>
> - **param_value_internal** (*float*) – Internal representation of the parameter value.

- **distribution** (`BaseDistribution`) – Sampled distribution of the parameter.

Raises

- **KeyError** – If no trial with the matching `trial_id` exists.

- **RuntimeError** – If the trial is already finished.

Return type

None

set_trial_state_values(*trial_id*, *state*, *values=None*)

Update the state and values of a trial.

Set return values of an objective function to values argument. If values argument is not None, this method overwrites any existing trial values.

Parameters

- **trial_id** (`int`) – ID of the trial.

- **state** (`TrialState`) – New state of the trial.

- **values** (`Sequence[float] | None`) – Values of the objective function.

Returns

True if the state is successfully updated. False if the state is kept the same. The latter happens when this method tries to update the state of *RUNNING* trial to *RUNNING*.

Raises

- **KeyError** – If no trial with the matching `trial_id` exists.

- **RuntimeError** – If the trial is already finished.

Return type

bool

set_trial_system_attr(*trial_id*, *key*, *value*)

Set an optuna-internal attribute to a trial.

This method overwrites any existing attribute.

Parameters

- **trial_id** (`int`) – ID of the trial.

- **key** (`str`) – Attribute key.

- **value** (`Mapping[str, JSONSerializable] | Sequence[JSONSerializable] | str | int | float | bool | None`) – Attribute value. It should be JSON serializable.

Raises

- **KeyError** – If no trial with the matching `trial_id` exists.

- **RuntimeError** – If the trial is already finished.

Return type

None

set_trial_user_attr(*trial_id*, *key*, *value*)

Set a user-defined attribute to a trial.

This method overwrites any existing attribute.

>> **Parameters**

>>> - **trial_id** (*int*) – ID of the trial.
>>> - **key** (*str*) – Attribute key.
>>> - **value** (*Any*) – Attribute value. It should be JSON serializable.

>> **Raises**

>>> - **KeyError** – If no trial with the matching `trial_id` exists.
>>> - **RuntimeError** – If the trial is already finished.

>> **Return type**
>>> None

> **upgrade()**

>> Upgrade the storage schema.

>> **Return type**
>>> None

optuna.storages.RetryFailedTrialCallback

class optuna.storages.**RetryFailedTrialCallback**(*max_retry=None*, *inherit_intermediate_values=False*)

> Retry a failed trial up to a maximum number of times.

> When a trial fails, this callback can be used with a class in `optuna.storages` to recreate the trial in `TrialState.WAITING` to queue up the trial to be run again.

> The failed trial can be identified by the `retried_trial_number()` function. Even if repetitive failure occurs (a retried trial fails again), this method returns the number of the original trial. To get a full list including the numbers of the retried trials as well as their original trial, call the `retry_history()` function.

> This callback is helpful in environments where trials may fail due to external conditions, such as being preempted by other processes.

> Usage:

```python
import optuna
from optuna.storages import RetryFailedTrialCallback

storage = optuna.storages.RDBStorage(
    url="sqlite:///:memory:",
    heartbeat_interval=60,
    grace_period=120,
    failed_trial_callback=RetryFailedTrialCallback(max_retry=3),
)

study = optuna.create_study(
    storage=storage,
)
```

> **See also:**

> See *RDBStorage*.

>> **Parameters**

- **max_retry** (*int* | *None*) – The max number of times a trial can be retried. Must be set to None or an integer. If set to the default value of None will retry indefinitely. If set to an integer, will only retry that many times.

- **inherit_intermediate_values** (*bool*) – Option to inherit *trial.intermediate_values* reported by `optuna.trial.Trial.report()` from the failed trial. Default is False.

Note: Added in v2.8.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.8.0.

Methods

retried_trial_number(trial)	Return the number of the original trial being retried.
retry_history(trial)	Return the list of retried trial numbers with respect to the specified trial.

static retried_trial_number(*trial*)

Return the number of the original trial being retried.

Parameters
    trial (FrozenTrial) – The trial object.

Returns
    The number of the first failed trial. If not retry of a previous trial, returns None.

Return type
    int | None

Note: Added in v2.8.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.8.0.

static retry_history(*trial*)

Return the list of retried trial numbers with respect to the specified trial.

Parameters
    trial (FrozenTrial) – The trial object.

Returns
    A list of trial numbers in ascending order of the series of retried trials. The first item of the list indicates the original trial which is identical to the `retried_trial_number()`, and the last item is the one right before the specified trial in the retry series. If the specified trial is not a retry of any trial, returns an empty list.

Return type
    List[int]

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

optuna.storages.fail_stale_trials

optuna.storages.**fail_stale_trials**(*study*)

> Fail stale trials and run their failure callbacks.
>
> The running trials whose heartbeat has not been updated for a long time will be failed, that is, those states will be changed to *FAIL*.
>
> **See also:**
>
> See *RDBStorage*.
>
>> **Parameters**
>>> **study** (*Study*) – Study holding the trials to check.
>>
>> **Return type**
>>> None

> **Note:** Added in v2.9.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.9.0.

optuna.storages.JournalStorage

class optuna.storages.**JournalStorage**(*log_storage*)

> Storage class for Journal storage backend.
>
> Note that library users can instantiate this class, but the attributes provided by this class are not supposed to be directly accessed by them.
>
> Journal storage writes a record of every operation to the database as it is executed and at the same time, keeps a latest snapshot of the database in-memory. If the database crashes for any reason, the storage can re-establish the contents in memory by replaying the operations stored from the beginning.
>
> Journal storage has several benefits over the conventional value logging storages.
>
> 1. The number of IOs can be reduced because of larger granularity of logs.
>
> 2. Journal storage has simpler backend API than value logging storage.
>
> 3. Journal storage keeps a snapshot in-memory so no need to add more cache.

Example

```
import optuna


def objective(trial): ...


storage = optuna.storages.JournalStorage(
    optuna.storages.JournalFileStorage("./journal.log"),
)
```

```
study = optuna.create_study(storage=storage)
study.optimize(objective)
```

In a Windows environment, an error message "A required privilege is not held by the client" may appear. In this case, you can solve the problem with creating storage by specifying *JournalFileOpenLock* as follows.

```
file_path = "./journal.log"
lock_obj = optuna.storages.JournalFileOpenLock(file_path)


storage = optuna.storages.JournalStorage(
    optuna.storages.JournalFileStorage(file_path, lock_obj=lock_obj),
)
```

Note: Added in v3.1.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

Methods

check_trial_is_updatable(trial_id, trial_state)	Check whether a trial state is updatable.
create_new_study(directions[, study_name])	Create a new study from a name.
create_new_trial(study_id[, template_trial])	Create and add a new trial to a study.
delete_study(study_id)	Delete a study.
get_all_studies()	Read a list of `FrozenStudy` objects.
get_all_trials(study_id[, deepcopy, states])	Read all trials in a study.
get_best_trial(study_id)	Return the trial with the best value in a study.
get_n_trials(study_id[, state])	Count the number of trials in a study.
get_study_directions(study_id)	Read whether a study maximizes or minimizes an objective.
get_study_id_from_name(study_name)	Read the ID of a study.
get_study_name_from_id(study_id)	Read the study name of a study.
get_study_system_attrs(study_id)	Read the optuna-internal attributes of a study.
get_study_user_attrs(study_id)	Read the user-defined attributes of a study.
get_trial(trial_id)	Read a trial.
get_trial_id_from_study_id_trial_number(...	Read the trial ID of a trial.
get_trial_number_from_id(trial_id)	Read the trial number of a trial.
get_trial_param(trial_id, param_name)	Read the parameter of a trial.
get_trial_params(trial_id)	Read the parameter dictionary of a trial.
get_trial_system_attrs(trial_id)	Read the optuna-internal attributes of a trial.
get_trial_user_attrs(trial_id)	Read the user-defined attributes of a trial.
remove_session()	Clean up all connections to a database.
`restore_replay_result`(snapshot)	
set_study_system_attr(study_id, key, value)	Register an optuna-internal attribute to a study.
set_study_user_attr(study_id, key, value)	Register a user-defined attribute to a study.
set_trial_intermediate_value(trial_id, step, ...)	Report an intermediate value of an objective function.
set_trial_param(trial_id, param_name, ...)	Set a parameter to a trial.
set_trial_state_values(trial_id, state[, values])	Update the state and values of a trial.
set_trial_system_attr(trial_id, key, value)	Set an optuna-internal attribute to a trial.
set_trial_user_attr(trial_id, key, value)	Set a user-defined attribute to a trial.

> **Parameters**
> **log_storage** (*BaseJournalLogStorage*) –

check_trial_is_updatable(*trial_id*, *trial_state*)

> Check whether a trial state is updatable.
>
> > **Parameters**
> >
> > - **trial_id** (*int*) – ID of the trial. Only used for an error message.
> >
> > - **trial_state** (*TrialState*) – Trial state to check.
> >
> > **Raises**
> > **RuntimeError** – If the trial is already finished.
> >
> > **Return type**
> > None

create_new_study(*directions*, *study_name=None*)

> Create a new study from a name.

If no name is specified, the storage class generates a name. The returned study ID is unique among all current and deleted studies.

> **Parameters**
>
> > - **directions** (*Sequence[*StudyDirection*]*) – A sequence of direction whose element is either *MAXIMIZE* or *MINIMIZE*.
> >
> > - **study_name** (*str | None*) – Name of the new study to create.
>
> **Returns**
> > ID of the created study.
>
> **Raises**
> > *optuna.exceptions.DuplicatedStudyError* – If a study with the same study_name already exists.
>
> **Return type**
> > int

create_new_trial(*study_id*, *template_trial=None*)

> Create and add a new trial to a study.
>
> The returned trial ID is unique among all current and deleted trials.
>
> > **Parameters**
> >
> > > - **study_id** (*int*) – ID of the study.
> > >
> > > - **template_trial** (*FrozenTrial | None*) – Template *FrozenTrial* with default user-attributes, system-attributes, intermediate-values, and a state.
> >
> > **Returns**
> > > ID of the created trial.
> >
> > **Raises**
> > > *KeyError* – If no study with the matching study_id exists.
> >
> > **Return type**
> > > int

delete_study(*study_id*)

> Delete a study.
>
> > **Parameters**
> > > **study_id** (*int*) – ID of the study.
> >
> > **Raises**
> > > *KeyError* – If no study with the matching study_id exists.
> >
> > **Return type**
> > > None

get_all_studies()

> Read a list of FrozenStudy objects.
>
> > **Returns**
> > > A list of FrozenStudy objects, sorted by study_id.
> >
> > **Return type**
> > > *List*[*FrozenStudy*]

get_all_trials(*study_id*, *deepcopy=True*, *states=None*)

> Read all trials in a study.

> > **Parameters**
> >
> > - **study_id** (*int*) – ID of the study.
> >
> > - **deepcopy** (*bool*) – Whether to copy the list of trials before returning. Set to True if you intend to update the list or elements of the list.
> >
> > - **states** (*Container[TrialState] | None*) – Trial states to filter on. If None, include all states.
> >
> > **Returns**
> > List of trials in the study, sorted by `trial_id`.
> >
> > **Raises**
> > **KeyError** – If no study with the matching `study_id` exists.
> >
> > **Return type**
> > *List*[FrozenTrial]

get_best_trial(*study_id*)

> Return the trial with the best value in a study.

> This method is valid only during single-objective optimization.

> > **Parameters**
> > **study_id** (*int*) – ID of the study.
> >
> > **Returns**
> > The trial with the best objective value among all finished trials in the study.
> >
> > **Raises**
> >
> > - **KeyError** – If no study with the matching `study_id` exists.
> >
> > - **RuntimeError** – If the study has more than one direction.
> >
> > - **ValueError** – If no trials have been completed.
> >
> > **Return type**
> > FrozenTrial

get_n_trials(*study_id*, *state=None*)

> Count the number of trials in a study.

> > **Parameters**
> >
> > - **study_id** (*int*) – ID of the study.
> >
> > - **state** (*Tuple[TrialState, ...] | TrialState | None*) – Trial states to filter on. If None, include all states.
> >
> > **Returns**
> > Number of trials in the study.
> >
> > **Raises**
> > **KeyError** – If no study with the matching `study_id` exists.
> >
> > **Return type**
> > int

get_study_directions(*study_id*)

> Read whether a study maximizes or minimizes an objective.
>
> > **Parameters**
> >
> > > **study_id** (*int*) – ID of a study.
> >
> > **Returns**
> >
> > > Optimization directions list of the study.
> >
> > **Raises**
> >
> > > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> >
> > > *List*[StudyDirection]

get_study_id_from_name(*study_name*)

> Read the ID of a study.
>
> > **Parameters**
> >
> > > **study_name** (*str*) – Name of the study.
> >
> > **Returns**
> >
> > > ID of the study.
> >
> > **Raises**
> >
> > > **KeyError** – If no study with the matching study_name exists.
> >
> > **Return type**
> >
> > > int

get_study_name_from_id(*study_id*)

> Read the study name of a study.
>
> > **Parameters**
> >
> > > **study_id** (*int*) – ID of the study.
> >
> > **Returns**
> >
> > > Name of the study.
> >
> > **Raises**
> >
> > > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> >
> > > str

get_study_system_attrs(*study_id*)

> Read the optuna-internal attributes of a study.
>
> > **Parameters**
> >
> > > **study_id** (*int*) – ID of the study.
> >
> > **Returns**
> >
> > > Dictionary with the optuna-internal attributes of the study.
> >
> > **Raises**
> >
> > > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> >
> > > *Dict*[str, *Any*]

get_study_user_attrs(*study_id*)

> Read the user-defined attributes of a study.

> **Parameters**
> > **study_id** (`int`) – ID of the study.
>
> **Returns**
> > Dictionary with the user attributes of the study.
>
> **Raises**
> > [`KeyError`](#) – If no study with the matching `study_id` exists.
>
> **Return type**
> > *Dict*[str, *Any*]

get_trial(*trial_id*)

> Read a trial.
>
> > **Parameters**
> > > **trial_id** (`int`) – ID of the trial.
> >
> > **Returns**
> > > Trial with a matching trial ID.
> >
> > **Raises**
> > > [`KeyError`](#) – If no trial with the matching `trial_id` exists.
> >
> > **Return type**
> > > FrozenTrial

get_trial_id_from_study_id_trial_number(*study_id*, *trial_number*)

> Read the trial ID of a trial.
>
> > **Parameters**
> >
> > - **study_id** (`int`) – ID of the study.
> >
> > - **trial_number** (`int`) – Number of the trial.
> >
> > **Returns**
> > > ID of the trial.
> >
> > **Raises**
> > > [`KeyError`](#) – If no trial with the matching `study_id` and `trial_number` exists.
> >
> > **Return type**
> > > int

get_trial_number_from_id(*trial_id*)

> Read the trial number of a trial.
>
> ---
>
> **Note:** The trial number is only unique within a study, and is sequential.
>
> ---
>
> > **Parameters**
> > > **trial_id** (`int`) – ID of the trial.
> >
> > **Returns**
> > > Number of the trial.
> >
> > **Raises**
> > > [`KeyError`](#) – If no trial with the matching `trial_id` exists.
> >
> > **Return type**
> > > int

get_trial_param(*trial_id*, *param_name*)

    Read the parameter of a trial.

        Parameters

- **trial_id** ([*int*](https://docs.python.org/3/library/functions.html#int)) – ID of the trial.

- **param_name** ([*str*](https://docs.python.org/3/library/stdtypes.html#str)) – Name of the parameter.

        Returns

        Internal representation of the parameter.

        Raises

        [**KeyError**](https://docs.python.org/3/library/exceptions.html#KeyError) – If no trial with the matching `trial_id` exists. If no such parameter exists.

        Return type

        [float](https://docs.python.org/3/library/functions.html#float)

get_trial_params(*trial_id*)

    Read the parameter dictionary of a trial.

        Parameters

        trial_id ([*int*](https://docs.python.org/3/library/functions.html#int)) – ID of the trial.

        Returns

        Dictionary of a parameters. Keys are parameter names and values are internal representations of the parameter values.

        Raises

        [**KeyError**](https://docs.python.org/3/library/exceptions.html#KeyError) – If no trial with the matching `trial_id` exists.

        Return type

        Dict[str, *Any*]

get_trial_system_attrs(*trial_id*)

    Read the optuna-internal attributes of a trial.

        Parameters

        trial_id ([*int*](https://docs.python.org/3/library/functions.html#int)) – ID of the trial.

        Returns

        Dictionary with the optuna-internal attributes of the trial.

        Raises

        [**KeyError**](https://docs.python.org/3/library/exceptions.html#KeyError) – If no trial with the matching `trial_id` exists.

        Return type

        Dict[str, *Any*]

get_trial_user_attrs(*trial_id*)

    Read the user-defined attributes of a trial.

        Parameters

        trial_id ([*int*](https://docs.python.org/3/library/functions.html#int)) – ID of the trial.

        Returns

        Dictionary with the user-defined attributes of the trial.

        Raises

        [**KeyError**](https://docs.python.org/3/library/exceptions.html#KeyError) – If no trial with the matching `trial_id` exists.

        Return type

        Dict[str, *Any*]

remove_session()

> Clean up all connections to a database.
>
> > **Return type**
> > None

set_study_system_attr(*study_id*, *key*, *value*)

> Register an optuna-internal attribute to a study.
>
> This method overwrites any existing attribute.
>
> > **Parameters**
> >
> > - **study_id** (*int*) – ID of the study.
> >
> > - **key** (*str*) – Attribute key.
> >
> > - **value** (*Mapping[str, Mapping[str, JSONSerializable]* | *Sequence[JSONSerializable]* | *str* | *int* | *float* | *bool* | *None]* | *Sequence[Mapping[str, JSONSerializable]* | *Sequence[JSONSerializable]* | *str* | *int* | *float* | *bool* | *None]* | *str* | *int* | *float* | *bool* | *None*) – Attribute value. It should be JSON serializable.
> >
> > **Raises**
> > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> > None

set_study_user_attr(*study_id*, *key*, *value*)

> Register a user-defined attribute to a study.
>
> This method overwrites any existing attribute.
>
> > **Parameters**
> >
> > - **study_id** (*int*) – ID of the study.
> >
> > - **key** (*str*) – Attribute key.
> >
> > - **value** (*Any*) – Attribute value. It should be JSON serializable.
> >
> > **Raises**
> > **KeyError** – If no study with the matching study_id exists.
> >
> > **Return type**
> > None

set_trial_intermediate_value(*trial_id*, *step*, *intermediate_value*)

> Report an intermediate value of an objective function.
>
> This method overwrites any existing intermediate value associated with the given step.
>
> > **Parameters**
> >
> > - **trial_id** (*int*) – ID of the trial.
> >
> > - **step** (*int*) – Step of the trial (e.g., the epoch when training a neural network).
> >
> > - **intermediate_value** (*float*) – Intermediate value corresponding to the step.
> >
> > **Raises**
> >
> > - **KeyError** – If no trial with the matching trial_id exists.
> >
> > - **RuntimeError** – If the trial is already finished.

> **Return type**
>   None

set_trial_param(*trial_id*, *param_name*, *param_value_internal*, *distribution*)

> Set a parameter to a trial.

> **Parameters**
>
> - **trial_id** (*int*) – ID of the trial.
>
> - **param_name** (*str*) – Name of the parameter.
>
> - **param_value_internal** (*float*) – Internal representation of the parameter value.
>
> - **distribution** (*BaseDistribution*) – Sampled distribution of the parameter.
>
> **Raises**
>
> - **KeyError** – If no trial with the matching `trial_id` exists.
>
> - **RuntimeError** – If the trial is already finished.
>
> **Return type**
>   None

set_trial_state_values(*trial_id*, *state*, *values=None*)

> Update the state and values of a trial.

> Set return values of an objective function to values argument. If values argument is not None, this method overwrites any existing trial values.

> **Parameters**
>
> - **trial_id** (*int*) – ID of the trial.
>
> - **state** (*TrialState*) – New state of the trial.
>
> - **values** (*Sequence[float] | None*) – Values of the objective function.
>
> **Returns**
>   True if the state is successfully updated. False if the state is kept the same. The latter happens when this method tries to update the state of *RUNNING* trial to *RUNNING*.
>
> **Raises**
>
> - **KeyError** – If no trial with the matching `trial_id` exists.
>
> - **RuntimeError** – If the trial is already finished.
>
> **Return type**
>   bool

set_trial_system_attr(*trial_id*, *key*, *value*)

> Set an optuna-internal attribute to a trial.

> This method overwrites any existing attribute.

> **Parameters**
>
> - **trial_id** (*int*) – ID of the trial.
>
> - **key** (*str*) – Attribute key.
>
> - **value**   (*Mapping[str, Mapping[str, JSONSerializable] |*
>   *Sequence[JSONSerializable] | str | int | float | bool*
>   *| None] | Sequence[Mapping[str, JSONSerializable] |*

> > > *Sequence[JSONSerializable] | str | int | float | bool | None] | str*
> > > *| int | float | bool | None*) – Attribute value. It should be JSON serializable.

> > **Raises**
> > - **KeyError** – If no trial with the matching `trial_id` exists.
> > - **RuntimeError** – If the trial is already finished.

> > **Return type**
> > None

> **set_trial_user_attr**(*trial_id*, *key*, *value*)
> > Set a user-defined attribute to a trial.

> > This method overwrites any existing attribute.

> > **Parameters**
> > - **trial_id** (*int*) – ID of the trial.
> > - **key** (*str*) – Attribute key.
> > - **value** (*Any*) – Attribute value. It should be JSON serializable.

> > **Raises**
> > - **KeyError** – If no trial with the matching `trial_id` exists.
> > - **RuntimeError** – If the trial is already finished.

> > **Return type**
> > None

optuna.storages.JournalFileStorage

class optuna.storages.**JournalFileStorage**(*file_path*, *lock_obj=None*)
> File storage class for Journal log backend.

> > **Parameters**
> > - **file_path** (*str*) – Path of file to persist the log to.
> > - **lock_obj** (*JournalFileBaseLock | None*) – Lock object for process exclusivity.

Methods

append_logs(logs)	Append logs to the backend.
read_logs(log_number_from)	Read logs with a log number greater than or equal to `log_number_from`.

append_logs(*logs*)
> Append logs to the backend.

> > **Parameters**
> > **logs** (*List[Dict[str, Any]]*) – A list that contains json-serializable logs.

> > **Return type**
> > None

read_logs(*log_number_from*)

> Read logs with a log number greater than or equal to `log_number_from`.
>
> If `log_number_from` is 0, read all the logs.
>
> > **Parameters**
> > > **log_number_from** (`int`) – A non-negative integer value indicating which logs to read.
> >
> > **Returns**
> > > Logs with log number greater than or equal to `log_number_from`.
> >
> > **Return type**
> > > *List*[*Dict*[str, *Any*]]

optuna.storages.JournalFileSymlinkLock

class optuna.storages.**JournalFileSymlinkLock**(*filepath*)

> Lock class for synchronizing processes for NFSv2 or later.
>
> On acquiring the lock, link system call is called to create an exclusive file. The file is deleted when the lock is released. In NFS environments prior to NFSv3, use this instead of *JournalFileOpenLock*
>
> > **Parameters**
> > > **filepath** (`str`) – The path of the file whose race condition must be protected.

Methods

acquire()	Acquire a lock in a blocking way by creating a symbolic link of a file.
release()	Release a lock by removing the symbolic link.

acquire()

> Acquire a lock in a blocking way by creating a symbolic link of a file.
>
> > **Returns**
> > > True if it succeeded in creating a symbolic link of `self._lock_target_file`.
> >
> > **Return type**
> > > bool

release()

> Release a lock by removing the symbolic link.
>
> > **Return type**
> > > None

optuna.storages.JournalFileOpenLock

class optuna.storages.**JournalFileOpenLock**(*filepath*)

>   Lock class for synchronizing processes for NFSv3 or later.
>
>   On acquiring the lock, open system call is called with the O_EXCL option to create an exclusive file. The file is deleted when the lock is released. This class is only supported when using NFSv3 or later on kernel 2.6 or later. In prior NFS environments, use *JournalFileSymlinkLock*.
>
>   > **Parameters**
>   >   **filepath** (*str*) – The path of the file whose race condition must be protected.

Methods

acquire()	Acquire a lock in a blocking way by creating a lock file.
release()	Release a lock by removing the created file.

acquire()

>   Acquire a lock in a blocking way by creating a lock file.
>
>   > **Returns**
>   >   True if it succeeded in creating a self._lock_file.
>   >
>   > **Return type**
>   >   bool

release()

>   Release a lock by removing the created file.
>
>   > **Return type**
>   >   None

optuna.storages.JournalRedisStorage

class optuna.storages.**JournalRedisStorage**(*url*, *use_cluster=False*, *prefix=''*)

>   Redis storage class for Journal log backend.
>
>   > **Parameters**
>   >
>   >   - **url** (*str*) – URL of the redis storage, password and db are optional. (ie: redis://localhost:6379)
>   >
>   >   - **use_cluster** (*bool*) – Flag whether you use the Redis cluster. If this is False, it is assumed that you use the standalone Redis server and ensured that a write operation is atomic. This provides the consistency of the preserved logs. If this is True, it is assumed that you use the Redis cluster and not ensured that a write operation is atomic. This means the preserved logs can be inconsistent due to network errors, and may cause errors.
>   >
>   >   - **prefix** (*str*) – Prefix of the preserved key of logs. This is useful when multiple users work on one Redis server.

Note: Added in v3.1.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

Methods

append_logs(logs)	Append logs to the backend.
load_snapshot()	Load snapshot from the backend.
read_logs(log_number_from)	Read logs with a log number greater than or equal to `log_number_from`.
save_snapshot(snapshot)	Save snapshot to the backend.

append_logs(*logs*)

> Append logs to the backend.
>
> > **Parameters**
> > > **logs** (`List[Dict[str, Any]]`) – A list that contains json-serializable logs.
> >
> > **Return type**
> > > None

load_snapshot()

> Load snapshot from the backend.
>
> > **Returns**
> > > A serialized snapshot (bytes) if found, otherwise None.
> >
> > **Return type**
> > > bytes | None

read_logs(*log_number_from*)

> Read logs with a log number greater than or equal to `log_number_from`.
>
> If `log_number_from` is 0, read all the logs.
>
> > **Parameters**
> > > **log_number_from** (*int*) – A non-negative integer value indicating which logs to read.
> >
> > **Returns**
> > > Logs with log number greater than or equal to `log_number_from`.
> >
> > **Return type**
> > > *List*[*Dict*[str, *Any*]]

save_snapshot(*snapshot*)

> Save snapshot to the backend.
>
> > **Parameters**
> > > **snapshot** (*bytes*) – A serialized snapshot (bytes)
> >
> > **Return type**
> > > None

7.3.13 optuna.study

The *study* module implements the *Study* object and related functions. A public constructor is available for the *Study* class, but direct use of this constructor is not recommended. Instead, library users should create and load a *Study* using *create_study()* and *load_study()* respectively.

optuna.study.Study	A study corresponds to an optimization task, i.e., a set of trials.
optuna.study.create_study	Create a new *Study*.
optuna.study.load_study	Load the existing *Study* that has the specified name.
optuna.study.delete_study	Delete a *Study* object.
optuna.study.copy_study	Copy study from one storage to another.
optuna.study.get_all_study_names	Get all study names stored in a specified storage.
optuna.study.get_all_study_summaries	Get all history of studies stored in a specified storage.
optuna.study.MaxTrialsCallback	Set a maximum number of trials before ending the study.
optuna.study.StudyDirection	Direction of a *Study*.
optuna.study.StudySummary	Basic attributes and aggregated results of a *Study*.

optuna.study.Study

class optuna.study.**Study**(*study_name*, *storage*, *sampler=None*, *pruner=None*)

A study corresponds to an optimization task, i.e., a set of trials.

This object provides interfaces to run a new *Trial*, access trials' history, set/get user-defined attributes of the study itself.

Note that the direct use of this constructor is not recommended. To create and load a study, please refer to the documentation of *create_study()* and *load_study()* respectively.

Methods

add_trial(trial)	Add trial to study.
add_trials(trials)	Add trials to study.
ask([fixed_distributions])	Create a new trial from which hyperparameters can be suggested.
enqueue_trial(params[, user_attrs, ...])	Enqueue a trial with given parameter values.
get_trials([deepcopy, states])	Return all trials in the study.
optimize(func[, n_trials, timeout, n_jobs, ...])	Optimize an objective function.
set_metric_names(metric_names)	Set metric names.
set_system_attr(key, value)	Set a system attribute to the study.
set_user_attr(key, value)	Set a user attribute to the study.
stop()	Exit from the current optimization loop after the running trials finish.
tell(trial[, values, state, skip_if_finished])	Finish a trial created with *ask()*.
trials_dataframe([attrs, multi_index])	Export trials as a pandas DataFrame.

Attributes

best_params	Return parameters of the best trial in the study.
best_trial	Return the best trial in the study.
best_trials	Return trials located at the Pareto front in the study.
best_value	Return the best objective value in the study.
direction	Return the direction of the study.
directions	Return the directions of the study.
metric_names	Return metric names.
system_attrs	Return system attributes.
trials	Return all trials in the study.
user_attrs	Return user attributes.

> **Parameters**
>
> - **study_name** (*str*) –
>
> - **storage** (*str | storages.BaseStorage*) –
>
> - **sampler** (*'samplers.BaseSampler' | None*) –
>
> - **pruner** (*pruners.BasePruner | None*) –

add_trial(*trial*)

> Add trial to study.
>
> The trial is validated before being added.

> **Example**

```python
import optuna
from optuna.distributions import FloatDistribution


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study()
assert len(study.trials) == 0

trial = optuna.trial.create_trial(
    params={"x": 2.0},
    distributions={"x": FloatDistribution(0, 10)},
    value=4.0,
)

study.add_trial(trial)
assert len(study.trials) == 1

study.optimize(objective, n_trials=3)
assert len(study.trials) == 4
```

```python
other_study = optuna.create_study()

for trial in study.trials:
    other_study.add_trial(trial)
assert len(other_study.trials) == len(study.trials)

other_study.optimize(objective, n_trials=2)
assert len(other_study.trials) == len(study.trials) + 2
```

See also:

This method should in general be used to add already evaluated trials (`trial.state.is_finished()` `==` `True`). To queue trials for evaluation, please refer to *`enqueue_trial()`*.

See also:

See *`create_trial()`* for how to create trials.

See also:

Please refer to add_trial_tutorial for the tutorial of specifying hyperparameters with the evaluated value manually.

> **Parameters**
>> **trial** (*FrozenTrial*) – Trial to add.
>
> **Return type**
>> None

add_trials(*trials*)

> Add trials to study.
>
> The trials are validated before being added.

> **Example**

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)
assert len(study.trials) == 3

other_study = optuna.create_study()
other_study.add_trials(study.trials)
assert len(other_study.trials) == len(study.trials)

other_study.optimize(objective, n_trials=2)
assert len(other_study.trials) == len(study.trials) + 2
```

> **See also:**
>
> See *add_trial()* for addition of each trial.
>
> > **Parameters**
> > > **trials** (*Iterable*[*FrozenTrial*]) – Trials to add.
> >
> > **Return type**
> > > None

ask(*fixed_distributions=None*)

> Create a new trial from which hyperparameters can be suggested.
>
> This method is part of an alternative to *optimize()* that allows controlling the lifetime of a trial outside the scope of func. Each call to this method should be followed by a call to *tell()* to finish the created trial.
>
> **See also:**
>
> The ask_and_tell tutorial provides use-cases with examples.
>
> ### Example
>
> Getting the trial object with the *ask()* method.
>
> ```python
> import optuna
>
>
> study = optuna.create_study()
>
> trial = study.ask()
>
> x = trial.suggest_float("x", -1, 1)
>
> study.tell(trial, x**2)
> ```
>
> ### Example
>
> Passing previously defined distributions to the *ask()* method.
>
> ```python
> import optuna
>
>
> study = optuna.create_study()
>
> distributions = {
>     "optimizer": optuna.distributions.CategoricalDistribution(["adam", "sgd"]),
>     "lr": optuna.distributions.FloatDistribution(0.0001, 0.1, log=True),
> }
>
> # You can pass the distributions previously defined.
> trial = study.ask(fixed_distributions=distributions)
>
> # `optimizer` and `lr` are already suggested and accessible with `trial.params`.
> ```

```
assert "optimizer" in trial.params
assert "lr" in trial.params
```

> **Parameters**
>> **fixed_distributions** (`dict[str, BaseDistribution] | None`) – A dictionary containing the parameter names and parameter's distributions. Each parameter in this dictionary is automatically suggested for the returned trial, even when the suggest method is not explicitly invoked by the user. If this argument is set to `None`, no parameter is automatically suggested.
>
> **Returns**
>> A *Trial*.
>
> **Return type**
>> Trial

property best_params: `dict[str, Any]`

> Return parameters of the best trial in the study.

> **Note:** This feature can only be used for single-objective optimization.

> **Returns**
>> A dictionary containing parameters of the best trial.

property best_trial: *FrozenTrial*

> Return the best trial in the study.

> **Note:** This feature can only be used for single-objective optimization. If your study is multi-objective, use *best_trials* instead.

> **Returns**
>> A *FrozenTrial* object of the best trial.
>
> **See also:**
>
> The reuse_best_trial tutorial provides a detailed example of how to use this method.

property best_trials: `list[FrozenTrial]`

> Return trials located at the Pareto front in the study.
>
> A trial is located at the Pareto front if there are no trials that dominate the trial. It's called that a trial `t0` dominates another trial `t1` if `all(v0 <= v1) for v0, v1 in zip(t0.values, t1.values)` and `any(v0 < v1) for v0, v1 in zip(t0.values, t1.values)` are held.
>
> **Returns**
>> A list of *FrozenTrial* objects.

property best_value: `float`

> Return the best objective value in the study.

> **Note:** This feature can only be used for single-objective optimization.

> **Returns**
>> A float representing the best objective value.

property direction: *StudyDirection*

> Return the direction of the study.

> **Note:** This feature can only be used for single-objective optimization. If your study is multi-objective, use *directions* instead.

> **Returns**
>> A *StudyDirection* object.

property directions: list[*StudyDirection*]

> Return the directions of the study.

>> **Returns**
>>> A list of *StudyDirection* objects.

enqueue_trial(*params*, *user_attrs=None*, *skip_if_exists=False*)

> Enqueue a trial with given parameter values.

> You can fix the next sampling parameters which will be evaluated in your objective function.

> **Example**

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study()
study.enqueue_trial({"x": 5})
study.enqueue_trial({"x": 0}, user_attrs={"memo": "optimal"})
study.optimize(objective, n_trials=2)

assert study.trials[0].params == {"x": 5}
assert study.trials[1].params == {"x": 0}
assert study.trials[1].user_attrs == {"memo": "optimal"}
```

> **Parameters**
>> - **params** (*dict[str, Any]*) – Parameter values to pass your objective function.
>> - **user_attrs** (*dict[str, Any] | None*) – A dictionary of user-specific attributes other than params.

- **skip_if_exists** (*bool*) – When `True`, prevents duplicate trials from being enqueued again.

Note: This method might produce duplicated trials if called simultaneously by multiple processes at the same time with same `params` dict.

Return type
    None

See also:

Please refer to enqueue_trial_tutorial for the tutorial of specifying hyperparameters manually.

get_trials(*deepcopy=True*, *states=None*)
    Return all trials in the study.

    The returned trials are ordered by trial number.

    See also:

    See *trials* for related property.

    Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)

trials = study.get_trials()
assert len(trials) == 3
```

Parameters

- **deepcopy** (*bool*) – Flag to control whether to apply `copy.deepcopy()` to the trials. Note that if you set the flag to `False`, you shouldn't mutate any fields of the returned trial. Otherwise the internal state of the study may corrupt and unexpected behavior may happen.

- **states** (*Container[TrialState] | None*) – Trial states to filter on. If `None`, include all states.

Returns
    A list of *FrozenTrial* objects.

Return type
    list[FrozenTrial]

---

property metric_names: list[str] | None

Return metric names.

> **Note:** Use *set_metric_names()* to set the metric names first.

> **Returns**
>
> > A list with names for each dimension of the returned values of the objective function.

optimize(*func, n_trials=None, timeout=None, n_jobs=1, catch=(), callbacks=None, gc_after_trial=False, show_progress_bar=False*)

Optimize an objective function.

Optimization is done by choosing a suitable set of hyperparameter values from a given range. Uses a sampler which implements the task of value suggestion based on a specified distribution. The sampler is specified in *create_study()* and the default choice for the sampler is TPE. See also *TPESampler* for more details on 'TPE'.

Optimization will be stopped when receiving a termination signal such as SIGINT and SIGTERM. Unlike other signals, a trial is automatically and cleanly failed when receiving SIGINT (Ctrl+C). If `n_jobs` is greater than one or if another signal than SIGINT is used, the interrupted trial state won't be properly updated.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)
```

> **Parameters**
>
> - **func** (*Callable[[Trial], float | Sequence[float]]*) – A callable that implements objective function.
>
> - **n_trials** (*int | None*) – The number of trials for each process. `None` represents no limit in terms of the number of trials. The study continues to create trials until the number of trials reaches `n_trials`, `timeout` period elapses, *stop()* is called, or a termination signal such as SIGTERM or Ctrl+C is received.
>
>   **See also:**
>
>   *optuna.study.MaxTrialsCallback* can ensure how many times trials will be performed across all processes.
>
> - **timeout** (*float | None*) – Stop study after the given number of second(s). `None` represents no limit in terms of elapsed time. The study continues to create trials until the number of trials reaches `n_trials`, `timeout` period elapses, *stop()* is called or, a termination signal such as SIGTERM or Ctrl+C is received.

- **n_jobs** (*int*) – The number of parallel jobs. If this argument is set to -1, the number is set to CPU count.

Note: n_jobs allows parallelization using threading and may suffer from Python's GIL. It is recommended to use process-based parallelization if func is CPU bound.

- **catch** (*Iterable[type[Exception]] | type[Exception]*) – A study continues to run even when a trial raises one of the exceptions specified in this argument. Default is an empty tuple, i.e. the study will stop for any exception except for *TrialPruned*.
- **callbacks** (*list[Callable[[Study, FrozenTrial], None]] | None*) – List of callback functions that are invoked at the end of each trial. Each function must accept two parameters with the following types in this order: *Study* and *FrozenTrial*.

    See also:

    See the tutorial of optuna_callback for how to use and implement callback functions.

- **gc_after_trial** (*bool*) – Flag to determine whether to automatically run garbage collection after each trial. Set to True to run the garbage collection, False otherwise. When it runs, it runs a full collection by internally calling gc.collect(). If you see an increase in memory consumption over several trials, try setting this flag to True.

    See also:

    How do I avoid running out of memory (OOM) when optimizing studies?

- **show_progress_bar** (*bool*) – Flag to show progress bars or not. To disable progress bar, set this False. Currently, progress bar is experimental feature and disabled when n_trials is None, timeout is not None, and n_jobs ≠ 1.

Raises
    RuntimeError – If nested invocation of this method occurs.

Return type
    None

set_metric_names(*metric_names*)

Set metric names.

This method names each dimension of the returned values of the objective function. It is particularly useful in multi-objective optimization. The metric names are mainly referenced by the visualization functions.

Example

```python
import optuna
import pandas


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2, x + 1


study = optuna.create_study(directions=["minimize", "minimize"])
study.set_metric_names(["x**2", "x+1"])
```

(continues on next page)

```python
study.optimize(objective, n_trials=3)

df = study.trials_dataframe(multi_index=True)
assert isinstance(df, pandas.DataFrame)
assert list(df.get("values").keys()) == ["x**2", "x+1"]
```

See also:

The names set by this method are used in `trials_dataframe()` and `plot_pareto_front()`.

> **Parameters**
>   **metric_names** (*list[str]*) – A list of metric names for the objective function.
>
> **Return type**
>   None

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

set_system_attr(*key*, *value*)

> Set a system attribute to the study.
>
> Note that Optuna internally uses this method to save system messages. Please use `set_user_attr()` to set users' attributes.
>
> **Parameters**
>
> - **key** (*str*) – A key string of the attribute.
>
> - **value** (*Any*) – A value of the attribute. The value should be JSON serializable.
>
> **Return type**
>   None

Warning: Deprecated in v3.1.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

---

set_user_attr(*key*, *value*)

> Set a user attribute to the study.
>
> **See also:**
>
> See `user_attrs` for related attribute.
>
> **See also:**
>
> See the recipe on attributes.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 1)
    y = trial.suggest_float("y", 0, 1)
    return x**2 + y**2


study = optuna.create_study()

study.set_user_attr("objective function", "quadratic function")
study.set_user_attr("dimensions", 2)
study.set_user_attr("contributors", ["Akiba", "Sano"])

assert study.user_attrs == {
    "objective function": "quadratic function",
    "dimensions": 2,
    "contributors": ["Akiba", "Sano"],
}
```

Parameters

- **key** (*str*) – A key string of the attribute.
- **value** (*Any*) – A value of the attribute. The value should be JSON serializable.

Return type

None

stop()

Exit from the current optimization loop after the running trials finish.

This method lets the running *optimize()* method return immediately after all trials which the *optimize()* method spawned finishes. This method does not affect any behaviors of parallel or successive study processes. This method only works when it is called inside an objective function or callback.

Example

```python
import optuna


def objective(trial):
    if trial.number == 4:
        trial.study.stop()
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=10)
assert len(study.trials) == 5
```

> **Return type**
>> None

property system_attrs: `dict[str, Any]`

> Return system attributes.
>
>> **Returns**
>>> A dictionary containing all system attributes.

> **Warning:** Deprecated in v3.1.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

tell(*trial*, *values=None*, *state=None*, *skip_if_finished=False*)

> Finish a trial created with *ask()*.
>
> **See also:**
>
> The ask_and_tell tutorial provides use-cases with examples.

> ### Example

```python
import optuna
from optuna.trial import TrialState


def f(x):
    return (x - 2) ** 2


def df(x):
    return 2 * x - 4


study = optuna.create_study()

n_trials = 30

for _ in range(n_trials):
    trial = study.ask()

    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)

    # Iterative gradient descent objective function.
    x = 3  # Initial value.
    for step in range(128):
        y = f(x)

        trial.report(y, step=step)

        if trial.should_prune():
            # Finish the trial with the pruned state.
```

<div align="right">(continues on next page)</div>

```
            study.tell(trial, state=TrialState.PRUNED)
            break

        gy = df(x)
        x -= gy * lr
    else:
        # Finish the trial with the final value after all iterations.
        study.tell(trial, y)
```

Parameters

- **trial** (*Trial* | *int*) – A *Trial* object or a trial number.

- **values** (*float* | *Sequence[float]* | *None*) – Optional objective value or a sequence of such values in case the study is used for multi-objective optimization. Argument must be provided if `state` is *COMPLETE* and should be *None* if `state` is *FAIL* or *PRUNED*.

- **state** (*TrialState* | *None*) – State to be reported. Must be *None*, *COMPLETE*, *FAIL* or *PRUNED*. If `state` is *None*, it will be updated to *COMPLETE* or *FAIL* depending on whether validation for `values` reported succeed or not.

- **skip_if_finished** (*bool*) – Flag to control whether exception should be raised when values for already finished trial are told. If *True*, tell is skipped without any error when the trial is already finished.

Returns

A *FrozenTrial* representing the resulting trial. A returned trial is deep copied thus user can modify it as needed.

Return type

FrozenTrial

property trials: list[*FrozenTrial*]

Return all trials in the study.

The returned trials are ordered by trial number.

This is a short form of `self.get_trials(deepcopy=True, states=None)`.

Returns

A list of *FrozenTrial* objects.

See also:

See *get_trials()* for related method.

trials_dataframe(*attrs=('number', 'value', 'datetime_start', 'datetime_complete', 'duration', 'params', 'user_attrs', 'system_attrs', 'state')*, *multi_index=False*)

Export trials as a pandas DataFrame.

The DataFrame provides various features to analyze studies. It is also useful to draw a histogram of objective values and to export trials as a CSV file. If there are no trials, an empty DataFrame is returned.

Example

```python
import optuna
import pandas


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)

# Create a dataframe from the study.
df = study.trials_dataframe()
assert isinstance(df, pandas.DataFrame)
assert df.shape[0] == 3  # n_trials.
```

Parameters

- **attrs** (*tuple[str, ...]*) – Specifies field names of *FrozenTrial* to include them to a DataFrame of trials.

- **multi_index** (*bool*) – Specifies whether the returned DataFrame employs MultiIndex or not. Columns that are hierarchical by nature such as (params, x) will be flattened to params_x when set to False.

Returns

A pandas DataFrame of trials in the *Study*.

Return type

pd.DataFrame

Note: If value is in attrs during multi-objective optimization, it is implicitly replaced with values.

Note: If *set_metric_names()* is called, the value or values is implicitly replaced with the dictionary with the objective name as key and the objective value as value.

---

property user_attrs: dict[str, Any]

Return user attributes.

See also:

See *set_user_attr()* for related method.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 1)
    y = trial.suggest_float("y", 0, 1)
    return x**2 + y**2


study = optuna.create_study()

study.set_user_attr("objective function", "quadratic function")
study.set_user_attr("dimensions", 2)
study.set_user_attr("contributors", ["Akiba", "Sano"])

assert study.user_attrs == {
    "objective function": "quadratic function",
    "dimensions": 2,
    "contributors": ["Akiba", "Sano"],
}
```

Returns

A dictionary containing all user attributes.

optuna.study.create_study

optuna.study.**create_study**(*, *storage=None*, *sampler=None*, *pruner=None*, *study_name=None*, *direction=None*, *load_if_exists=False*, *directions=None*)

Create a new *Study*.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)
```

Parameters

- **storage** (*str | storages.BaseStorage | None*) – Database URL. If this argument is set to None, in-memory storage is used, and the *Study* will not be persistent.

Note:

When a database URL is passed, Optuna internally uses SQLAlchemy to handle the database. Please refer to SQLAlchemy's document for further details. If you want to specify non-default options to SQLAlchemy Engine, you can instantiate *RDBStorage* with your desired options and pass it to the `storage` argument instead of a URL.

- **sampler** (*'samplers.BaseSampler' | None*) – A sampler object that implements background algorithm for value suggestion. If None is specified, *TPESampler* is used during single-objective optimization and *NSGAIISampler* during multi-objective optimization. See also *samplers*.

- **pruner** (*pruners.BasePruner | None*) – A pruner object that decides early stopping of unpromising trials. If None is specified, *MedianPruner* is used as the default. See also *pruners*.

- **study_name** (*str | None*) – Study's name. If this argument is set to None, a unique name is generated automatically.

- **direction** (*str | StudyDirection | None*) – Direction of optimization. Set `minimize` for minimization and `maximize` for maximization. You can also pass the corresponding *StudyDirection* object. `direction` and `directions` must not be specified at the same time.

  Note: If none of *direction* and *directions* are specified, the direction of the study is set to "minimize".

- **load_if_exists** (*bool*) – Flag to control the behavior to handle a conflict of study names. In the case where a study named `study_name` already exists in the `storage`, a *DuplicatedStudyError* is raised if `load_if_exists` is set to False. Otherwise, the creation of the study is skipped, and the existing one is returned.

- **directions** (*Sequence[str | StudyDirection] | None*) – A sequence of directions during multi-objective optimization. `direction` and `directions` must not be specified at the same time.

Returns
  A *Study* object.

Return type
  Study

See also:

`optuna.create_study()` is an alias of `optuna.study.create_study()`.

See also:

The rdb tutorial provides concrete examples to save and resume optimization using RDB.

optuna.study.load_study

optuna.study.**load_study**(*, *study_name*, *storage*, *sampler=None*, *pruner=None*)

Load the existing *Study* that has the specified name.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 10)
    return x**2


study = optuna.create_study(storage="sqlite:///example.db", study_name="my_study")
study.optimize(objective, n_trials=3)

loaded_study = optuna.load_study(study_name="my_study", storage="sqlite:///example.
↪db")
assert len(loaded_study.trials) == len(study.trials)
```

Parameters

- **study_name** (*str | None*) – Study's name. Each study has a unique name as an identifier. If None, checks whether the storage contains a single study, and if so loads that study. study_name is required if there are multiple studies in the storage.

- **storage** (*str | storages.BaseStorage*) – Database URL such as sqlite:/// example.db. Please see also the documentation of *create_study()* for further details.

- **sampler** (*'samplers.BaseSampler' | None*) – A sampler object that implements background algorithm for value suggestion. If None is specified, *TPESampler* is used as the default. See also *samplers*.

- **pruner** (*pruners.BasePruner | None*) – A pruner object that decides early stopping of unpromising trials. If None is specified, *MedianPruner* is used as the default. See also *pruners*.

Return type

Study

See also:

optuna.load_study() is an alias of *optuna.study.load_study()*.

optuna.study.delete_study

optuna.study.**delete_study**(*, *study_name*, *storage*)

>   Delete a *Study* object.

>   **Example**

>   ```python
>   import optuna
>
>
>   def objective(trial):
>       x = trial.suggest_float("x", -10, 10)
>       return (x - 2) ** 2
>
>
>   study = optuna.create_study(study_name="example-study", storage="sqlite:///example.
>   ↪db")
>   study.optimize(objective, n_trials=3)
>
>   optuna.delete_study(study_name="example-study", storage="sqlite:///example.db")
>   ```

>   **Parameters**
>   *   **study_name** (*str*) – Study's name.
>   *   **storage** (*str | BaseStorage*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.

>   **Return type**
>   None

>   **See also:**

>   *optuna.delete_study()* is an alias of *optuna.study.delete_study()*.

optuna.study.copy_study

optuna.study.**copy_study**(*, *from_study_name*, *from_storage*, *to_storage*, *to_study_name=None*)

>   Copy study from one storage to another.

>   The direction(s) of the objective(s) in the study, trials, user attributes and system attributes are copied.

>   **Note:** *copy_study()* copies a study even if the optimization is working on. It means users will get a copied study that contains a trial that is not finished.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(
    study_name="example-study",
    storage="sqlite:///example.db",
)
study.optimize(objective, n_trials=3)

optuna.copy_study(
    from_study_name="example-study",
    from_storage="sqlite:///example.db",
    to_storage="sqlite:///example_copy.db",
)

study = optuna.load_study(
    study_name=None,
    storage="sqlite:///example_copy.db",
)
```

Parameters

- **from_study_name** (*str*) – Name of study.

- **from_storage** (*str | BaseStorage*) – Source database URL such as `sqlite:///` `example.db`. Please see also the documentation of *create_study()* for further details.

- **to_storage** (*str | BaseStorage*) – Destination database URL.

- **to_study_name** (*str | None*) – Name of the created study. If omitted, `from_study_name` is used.

Raises

DuplicatedStudyError – If a study with a conflicting name already exists in the destination storage.

Return type

None

optuna.study.get_all_study_names

optuna.study.**get_all_study_names**(*storage*)

> Get all study names stored in a specified storage.

> **Example**

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(study_name="example-study", storage="sqlite:///example.
→db")
study.optimize(objective, n_trials=3)

study_names = optuna.study.get_all_study_names(storage="sqlite:///example.db")
assert len(study_names) == 1

assert study_names[0] == "example-study"
```

> **Parameters**
>> **storage** (*str | BaseStorage*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.

> **Returns**
>> List of all study names in the storage.

> **Return type**
>> list[str]

> **See also:**

> *optuna.get_all_study_names()* is an alias of *optuna.study.get_all_study_names()*.

optuna.study.get_all_study_summaries

optuna.study.**get_all_study_summaries**(*storage*, *include_best_trial=True*)

> Get all history of studies stored in a specified storage.

Example

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2


study = optuna.create_study(study_name="example-study", storage="sqlite:///example.
↪db")
study.optimize(objective, n_trials=3)

study_summaries = optuna.study.get_all_study_summaries(storage="sqlite:///example.db
↪")
assert len(study_summaries) == 1

study_summary = study_summaries[0]
assert study_summary.study_name == "example-study"
```

   Parameters

   - **storage** (*str* | *BaseStorage*) – Database URL such as `sqlite:///example.db`.
     Please see also the documentation of *create_study()* for further details.

   - **include_best_trial** (*bool*) – Include the best trials if exist. It potentially increases the
     number of queries and may take longer to fetch summaries depending on the storage.

   Returns

   List of study history summarized as *StudySummary* objects.

   Return type

   list[StudySummary]

See also:

optuna.get_all_study_summaries() is an alias of *optuna.study.get_all_study_summaries()*.

optuna.study.MaxTrialsCallback

class optuna.study.**MaxTrialsCallback**(*n_trials*, *states=(TrialState.COMPLETE,)*)

   Set a maximum number of trials before ending the study.

   While the `n_trials` argument of *optuna.study.Study.optimize()* sets the number of trials that will be
   run, you may want to continue running until you have a certain number of successfully completed trials or stop
   the study when you have a certain number of trials that fail. This `MaxTrialsCallback` class allows you to set
   a maximum number of trials for a particular *TrialState* before stopping the study.

Example

```python
import optuna
from optuna.study import MaxTrialsCallback
from optuna.trial import TrialState


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    return x**2


study = optuna.create_study()
study.optimize(
    objective,
    callbacks=[MaxTrialsCallback(10, states=(TrialState.COMPLETE,))],
)
```

> **Parameters**
>
> - **n_trials** (*int*) – The max number of trials. Must be set to an integer.
> - **states** (*Container[TrialState]* | *None*) – Tuple of the *TrialState* to be counted towards the max trials limit. Default value is (`TrialState.COMPLETE,`). If `None`, count all states.

optuna.study.StudyDirection

class optuna.study.**StudyDirection**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

> Direction of a *Study*.
>
> **NOT_SET**
>
> > Direction has not been set.
>
> **MINIMIZE**
>
> > *Study* minimizes the objective function.
>
> **MAXIMIZE**
>
> > *Study* maximizes the objective function.

Methods

as_integer_ratio()	Return integer ratio.
bit_count()	Number of ones in the binary representation of the absolute value of self.
bit_length()	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes([byteorder, signed])	Return the integer represented by the given array of bytes.
to_bytes([length, byteorder, signed])	Return an array of bytes representing an integer.

Attributes

NOT_SET	
MINIMIZE	
MAXIMIZE	
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

as_integer_ratio()

> Return integer ratio.
>
> Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.
>
> ```
> >>> (10).as_integer_ratio()
> (10, 1)
> >>> (-10).as_integer_ratio()
> (-10, 1)
> >>> (0).as_integer_ratio()
> (0, 1)
> ```

bit_count()

> Number of ones in the binary representation of the absolute value of self.
>
> Also known as the population count.
>
> ```
> >>> bin(13)
> '0b1101'
> >>> (13).bit_count()
> 3
> ```

bit_length()

> Number of bits necessary to represent self in binary.
>
> ```
> >>> bin(37)
> '0b100101'
> >>> (37).bit_length()
> 6
> ```

conjugate()

> Returns self, the complex conjugate of any int.

denominator

> the denominator of a rational number in lowest terms

from_bytes(*byteorder='big'*, ***, *signed=False*)

> Return the integer represented by the given array of bytes.
>
> **bytes**
>
> > Holds the array of bytes to convert. The argument must either support the buffer protocol or be an

> iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder
> The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use 'big'.

signed
> Indicates whether two's complement is used to represent the integer.

`imag`

> the imaginary part of a complex number

`numerator`

> the numerator of a rational number in lowest terms

`real`

> the real part of a complex number

`to_bytes`(*length=1*, *byteorder='big'*, ***, *signed=False*)

> Return an array of bytes representing an integer.

length
> Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
> The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use 'big'.

signed
> Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

optuna.study.StudySummary

class `optuna.study.`**StudySummary**(*study_name*, *direction*, *best_trial*, *user_attrs*, *system_attrs*, *n_trials*, *datetime_start*, *study_id*, ***, *directions=None*)

> Basic attributes and aggregated results of a *Study*.
>
> See also *optuna.study.get_all_study_summaries()*.
>
> **Parameters**
>
> - **study_name** (*str*) –
> - **direction** (*StudyDirection | None*) –
> - **best_trial** (*trial.FrozenTrial | None*) –
> - **user_attrs** (*dict[str, Any]*) –
> - **system_attrs** (*dict[str, Any]*) –
> - **n_trials** (*int*) –
> - **datetime_start** (*datetime.datetime | None*) –

- **study_id** (*int*) –

- **directions** (*Sequence[*StudyDirection*] | None*) –

study_name
> Name of the *Study*.

direction
> *StudyDirection* of the *Study*.

> **Note:** This attribute is only available during single-objective optimization.

---

directions
> A sequence of *StudyDirection* objects.

best_trial
> *optuna.trial.FrozenTrial* with best objective value in the *Study*.

user_attrs
> Dictionary that contains the attributes of the *Study* set with *optuna.study.Study.set_user_attr()*.

system_attrs
> Dictionary that contains the attributes of the *Study* internally set by Optuna.

> **Warning:** Deprecated in v3.1.0. `system_attrs` argument will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

---

n_trials
> The number of trials ran in the *Study*.

datetime_start
> Datetime where the *Study* started.

Attributes

direction
directions
system_attrs

7.3.14 optuna.terminator

The `terminator` module implements a mechanism for automatically terminating the optimization process, accompanied by a callback class for the termination and evaluators for the estimated room for improvement in the optimization and statistical error of the objective function. The terminator stops the optimization process when the estimated potential improvement is smaller than the statistical error.

`optuna.terminator.BaseTerminator`	Base class for terminators.
`optuna.terminator.Terminator`	Automatic stopping mechanism for Optuna studies.
`optuna.terminator.BaseImprovementEvaluator`	Base class for improvement evaluators.
`optuna.terminator.RegretBoundEvaluator`	An error evaluator for upper bound on the regret with high-probability confidence.
`optuna.terminator.BestValueStagnationEvaluator`	Evaluates the stagnation period of the best value in an optimization process.
`optuna.terminator.BaseErrorEvaluator`	Base class for error evaluators.
`optuna.terminator.CrossValidationErrorEvaluator`	An error evaluator for objective functions based on cross-validation.
`optuna.terminator.StaticErrorEvaluator`	An error evaluator that always returns a constant value.
`optuna.terminator.TerminatorCallback`	A callback that terminates the optimization using Terminator.
`optuna.terminator.report_cross_validation_scores`	A function to report cross-validation scores of a trial.

optuna.terminator.BaseTerminator

class optuna.terminator.**BaseTerminator**

Base class for terminators.

Methods

should_terminate(study)

optuna.terminator.Terminator

class optuna.terminator.**Terminator**(*improvement_evaluator=None*, *error_evaluator=None*, *min_n_trials=20*)

Automatic stopping mechanism for Optuna studies.

This class implements an automatic stopping mechanism for Optuna studies, aiming to prevent unnecessary computation. The study is terminated when the statistical error, e.g. cross-validation error, exceeds the room left for optimization.

For further information about the algorithm, please refer to the following paper:

- A. Makarova et al. Automatic termination for hyperparameter optimization.

   Parameters

- **improvement_evaluator** (*Optional[BaseImprovementEvaluator]*) – An evaluator object for assessing the room left for optimization. Defaults to a *RegretBoundEvaluator* object.

- **error_evaluator** (*Optional[BaseErrorEvaluator]*) – An evaluator for calculating the statistical error, e.g. cross-validation error. Defaults to a *CrossValidationErrorEvaluator* object.

- **min_n_trials** (*int*) – The minimum number of trials before termination is considered. Defaults to 20.

Raises
    ValueError – If `min_n_trials` is not a positive integer.

Example

```python
import logging
import sys

from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

import optuna
from optuna.terminator import Terminator
from optuna.terminator import report_cross_validation_scores


study = optuna.create_study(direction="maximize")
terminator = Terminator()
min_n_trials = 20

while True:
    trial = study.ask()

    X, y = load_wine(return_X_y=True)

    clf = RandomForestClassifier(
        max_depth=trial.suggest_int("max_depth", 2, 32),
        min_samples_split=trial.suggest_float("min_samples_split", 0, 1),
        criterion=trial.suggest_categorical("criterion", ("gini", "entropy")),
    )

    scores = cross_val_score(clf, X, y, cv=KFold(n_splits=5, shuffle=True))
    report_cross_validation_scores(trial, scores)

    value = scores.mean()
    logging.info(f"Trial #{trial.number} finished with value {value}.")
    study.tell(trial, value)

    if trial.number > min_n_trials and terminator.should_terminate(study):
        logging.info("Terminated by Optuna Terminator!")
        break
```

See also:

Please refer to *TerminatorCallback* for how to use the terminator mechanism with the *optimize()* method.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

Methods

should_terminate(study)	Judge whether the study should be terminated based on the reported values.

should_terminate(*study*)

Judge whether the study should be terminated based on the reported values.

> **Parameters**
>   **study** (Study) –
>
> **Return type**
>   bool

optuna.terminator.BaseImprovementEvaluator

class optuna.terminator.**BaseImprovementEvaluator**(**args*, ***kwargs*)

Base class for improvement evaluators.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

Methods

evaluate(trials, study_direction)

optuna.terminator.RegretBoundEvaluator

class optuna.terminator.**RegretBoundEvaluator**(*top_trials_ratio=0.5*, *min_n_trials=20*, *seed=None*)

An error evaluator for upper bound on the regret with high-probability confidence.

This evaluator evaluates the regret of current best solution, which defined as the difference between the objective value of the best solution and of the global optimum. To be specific, this evaluator calculates the upper bound on the regret based on the fact that empirical estimator of the objective function is bounded by lower and upper confidence bounds with high probability under the Gaussian process model assumption.

> **Parameters**
> - **gp** – A Gaussian process model on which evaluation base. If not specified, the default Gaussian process model is used.

- **top_trials_ratio** (*float*) – A ratio of top trials to be considered when estimating the regret. Default to 0.5.

- **min_n_trials** (*int*) – A minimum number of complete trials to estimate the regret. Default to 20.

- **min_lcb_n_additional_samples** – A minimum number of additional samples to estimate the lower confidence bound. Default to 2000.

- **seed** (*int | None*) –

For further information about this evaluator, please refer to the following paper:

- Automatic Termination for Hyperparameter Optimization

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

Methods

evaluate(trials, study_direction)

optuna.terminator.BestValueStagnationEvaluator

class optuna.terminator.**BestValueStagnationEvaluator**(*max_stagnation_trials=30*)

Evaluates the stagnation period of the best value in an optimization process.

This class is initialized with a maximum stagnation period (*max_stagnation_trials*) and is designed to evaluate the remaining trials before reaching this maximum period of allowed stagnation. If this remaining trials reach zero, the trial terminates. Therefore, the default error evaluator is instantiated by StaticErrorEvaluator(const=0).

> **Parameters**
> **max_stagnation_trials** (*int*) – The maximum number of trials allowed for stagnation.

Note: Added in v3.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.4.0.

Methods

evaluate(trials, study_direction)

optuna.terminator.BaseErrorEvaluator

class optuna.terminator.**BaseErrorEvaluator**

Base class for error evaluators.

Methods

evaluate(trials, study_direction)

optuna.terminator.CrossValidationErrorEvaluator

class optuna.terminator.**CrossValidationErrorEvaluator**(**args*, ***kwargs*)

An error evaluator for objective functions based on cross-validation.

This evaluator evaluates the objective function's statistical error, which comes from the randomness of dataset. This evaluator assumes that the objective function is the average of the cross-validation and uses the scaled variance of the cross-validation scores in the best trial at the moment as the statistical error.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

Methods

evaluate(trials, study_direction)	Evaluate the statistical error of the objective function based on cross-validation.

evaluate(*trials*, *study_direction*)

Evaluate the statistical error of the objective function based on cross-validation.

> **Parameters**
> - **trials** (*list[FrozenTrial]*) – A list of trials to consider. The best trial in `trials` is used to compute the statistical error.
> - **study_direction** (*StudyDirection*) – The direction of the study.
>
> **Returns**
> A float representing the statistical error of the objective function.
>
> **Return type**
> float

optuna.terminator.StaticErrorEvaluator

class optuna.terminator.**StaticErrorEvaluator**(*constant*)

An error evaluator that always returns a constant value.

This evaluator can be used to terminate the optimization when the evaluated improvement potential is below the fixed threshold.

> **Parameters**
> **constant** (*float*) – A user-specified constant value to always return as an error estimate.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

Methods

evaluate(trials, study_direction)

optuna.terminator.TerminatorCallback

class optuna.terminator.**TerminatorCallback**(*terminator=None*)

A callback that terminates the optimization using Terminator.

This class implements a callback which wraps *Terminator* so that it can be used with the *optimize()* method.

> **Parameters**
> **terminator** (*Optional[BaseTerminator]*) – A terminator object which determines whether to terminate the optimization by assessing the room for optimization and statistical error. Defaults to a *Terminator* object with default improvement_evaluator and error_evaluator.

Example

```python
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

import optuna
from optuna.terminator import TerminatorCallback
from optuna.terminator import report_cross_validation_scores


def objective(trial):
    X, y = load_wine(return_X_y=True)

    clf = RandomForestClassifier(
        max_depth=trial.suggest_int("max_depth", 2, 32),
        min_samples_split=trial.suggest_float("min_samples_split", 0, 1),
```

```
        criterion=trial.suggest_categorical("criterion", ("gini", "entropy")),
    )

    scores = cross_val_score(clf, X, y, cv=KFold(n_splits=5, shuffle=True))
    report_cross_validation_scores(trial, scores)
    return scores.mean()


study = optuna.create_study(direction="maximize")
terminator = TerminatorCallback()
study.optimize(objective, n_trials=50, callbacks=[terminator])
```

See also:

Please refer to *Terminator* for the details of the terminator mechanism.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

optuna.terminator.report_cross_validation_scores

optuna.terminator.**report_cross_validation_scores**(*trial*, *scores*)

>   A function to report cross-validation scores of a trial.
>
>   This function should be called within the objective function to report the cross-validation scores. The reported scores are used to evaluate the statistical error for termination judgement.
>
>   **Parameters**
>
>   - **trial** (*Trial*) – A *Trial* object to report the cross-validation scores.
>
>   - **scores** (*list[float]*) – The cross-validation scores of the trial.
>
>   **Return type**
>   None

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

For an example of using this module, please refer to this example.

7.3.15 optuna.trial

The *trial* module contains *Trial* related classes and functions.

A *Trial* instance represents a process of evaluating an objective function. This instance is passed to an objective function and provides interfaces to get parameter suggestion, manage the trial's state, and set/get user-defined attributes of the trial, so that Optuna users can define a custom objective function through the interfaces. Basically, Optuna users only use it in their custom objective functions.

`optuna.trial.Trial`	A trial is a process of evaluating an objective function.
`optuna.trial.FixedTrial`	A trial class which suggests a fixed value for each parameter.
`optuna.trial.FrozenTrial`	Status and results of a `Trial`.
`optuna.trial.TrialState`	State of a `Trial`.
`optuna.trial.create_trial`	Create a new `FrozenTrial`.

optuna.trial.Trial

class optuna.trial.**Trial**(*study*, *trial_id*)

A trial is a process of evaluating an objective function.

This object is passed to an objective function and provides interfaces to get parameter suggestion, manage the trial's state, and set/get user-defined attributes of the trial.

Note that the direct use of this constructor is not recommended. This object is seamlessly instantiated and passed to the objective function behind the `optuna.study.Study.optimize()` method; hence library users do not care about instantiation of this object.

> **Parameters**
>
> - **study** (`optuna.study.Study`) – A *Study* object.
>
> - **trial_id** (`int`) – A trial ID that is automatically generated.

Methods

`report`(value, step)	Report an objective function value for a given step.
`set_system_attr`(key, value)	Set system attributes to the trial.
`set_user_attr`(key, value)	Set user attributes to the trial.
`should_prune`()	Suggest whether the trial should be pruned or not.
`suggest_categorical`()	Suggest a value for the categorical parameter.
`suggest_discrete_uniform`(name, low, high, q)	Suggest a value for the discrete parameter.
`suggest_float`(name, low, high, *[, step, log])	Suggest a value for the floating point parameter.
`suggest_int`(name, low, high, *[, step, log])	Suggest a value for the integer parameter.
`suggest_loguniform`(name, low, high)	Suggest a value for the continuous parameter.
`suggest_uniform`(name, low, high)	Suggest a value for the continuous parameter.

Attributes

`datetime_start`	Return start datetime.
`distributions`	Return distributions of parameters to be optimized.
`number`	Return trial's number which is consecutive and unique in a study.
`params`	Return parameters to be optimized.
`relative_params`	
`system_attrs`	Return system attributes.
`user_attrs`	Return user attributes.

property datetime_start: datetime | None

> Return start datetime.
>
> > **Returns**
> >
> > Datetime where the *Trial* started.

property distributions: Dict[str, BaseDistribution]

> Return distributions of parameters to be optimized.
>
> > **Returns**
> >
> > A dictionary containing all distributions.

property number: int

> Return trial's number which is consecutive and unique in a study.
>
> > **Returns**
> >
> > A trial number.

property params: Dict[str, Any]

> Return parameters to be optimized.
>
> > **Returns**
> >
> > A dictionary containing all parameters.

report(*value*, *step*)

> Report an objective function value for a given step.
>
> The reported values are used by the pruners to determine whether this trial should be pruned.
>
> **See also:**
>
> Please refer to *BasePruner*.
>
> ---
>
> **Note:** The reported value is converted to `float` type by applying `float()` function internally. Thus, it accepts all float-like types (e.g., `numpy.float32`). If the conversion fails, a `TypeError` is raised.
>
> ---
>
> ---
>
> **Note:** If this method is called multiple times at the same `step` in a trial, the reported `value` only the first time is stored and the reported values from the second time are ignored.
>
> ---
>
> ---
>
> **Note:** *report()* does not support multi-objective optimization.
>
> ---
>
> ### Example
>
> Report intermediate scores of SGDClassifier training.
>
> ```python
> import numpy as np
> from sklearn.datasets import load_iris
> from sklearn.linear_model import SGDClassifier
> from sklearn.model_selection import train_test_split
>
> import optuna
> ```
>
> (continues on next page)

```python
X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)


def objective(trial):
    clf = SGDClassifier(random_state=0)
    for step in range(100):
        clf.partial_fit(X_train, y_train, np.unique(y))
        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step=step)
        if trial.should_prune():
            raise optuna.TrialPruned()

    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=3)
```

Parameters

- **value** (*float*) – A value returned from the objective function.
- **step** (*int*) – Step of the trial (e.g., Epoch of neural network training). Note that pruners assume that `step` starts at zero. For example, *MedianPruner* simply checks if `step` is less than n_warmup_steps as the warmup mechanism. `step` must be a positive integer.

Return type
    None

set_system_attr(*key*, *value*)

Set system attributes to the trial.

Note that Optuna internally uses this method to save system messages such as failure reason of trials. Please use *set_user_attr()* to set users' attributes.

Parameters

- **key** (*str*) – A key string of the attribute.
- **value** (*Any*) – A value of the attribute. The value should be JSON serializable.

Return type
    None

> **Warning:** Deprecated in v3.1.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

set_user_attr(*key*, *value*)

Set user attributes to the trial.

The user attributes in the trial can be access via *optuna.trial.Trial.user_attrs()*.

See also:

See the recipe on attributes.

Example

Save fixed hyperparameters of neural network training.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y, random_state=0)


def objective(trial):
    trial.set_user_attr("BATCHSIZE", 128)
    momentum = trial.suggest_float("momentum", 0, 1.0)
    clf = MLPClassifier(
        hidden_layer_sizes=(100, 50),
        batch_size=trial.user_attrs["BATCHSIZE"],
        momentum=momentum,
        solver="sgd",
        random_state=0,
    )
    clf.fit(X_train, y_train)

    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=3)
assert "BATCHSIZE" in study.best_trial.user_attrs.keys()
assert study.best_trial.user_attrs["BATCHSIZE"] == 128
```

Parameters

- **key** (*str*) – A key string of the attribute.

- **value** (*Any*) – A value of the attribute. The value should be JSON serializable.

Return type
None

should_prune()

Suggest whether the trial should be pruned or not.

The suggestion is made by a pruning algorithm associated with the trial and is based on previously reported values. The algorithm can be specified when constructing a *Study*.

Note: If no values have been reported, the algorithm cannot make meaningful suggestions. Similarly, if this method is called multiple times with the exact same set of reported values, the suggestions will be the same.

See also:

Please refer to the example code in `optuna.trial.Trial.report()`.

Note: `should_prune()` does not support multi-objective optimization.

> **Returns**
> A boolean value. If `True`, the trial should be pruned according to the configured pruning algorithm. Otherwise, the trial should continue.
>
> **Return type**
> bool

suggest_categorical(*name: str, choices: Sequence[None]*) → None

suggest_categorical(*name: str, choices: Sequence[bool]*) → bool

suggest_categorical(*name: str, choices: Sequence[int]*) → int

suggest_categorical(*name: str, choices: Sequence[float]*) → float

suggest_categorical(*name: str, choices: Sequence[str]*) → str

suggest_categorical(*name: str, choices: Sequence[None | bool | int | float | str]*) → None | bool | int | float | str

Suggest a value for the categorical parameter.

The value is sampled from `choices`.

Example

Suggest a kernel function of SVC.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)


def objective(trial):
    kernel = trial.suggest_categorical("kernel", ["linear", "poly", "rbf"])
    clf = SVC(kernel=kernel, gamma="scale", random_state=0)
    clf.fit(X_train, y_train)
    return clf.score(X_valid, y_valid)
```

(continues on next page)

```
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=3)
```

> **Parameters**
>
> - **name** – A parameter name.
>
> - **choices** – Parameter value candidates.

> **See also:**
>
> *CategoricalDistribution*.

> **Returns**
>   A suggested value.

> **See also:**
>
> configurations tutorial describes more details and flexible usages.

suggest_discrete_uniform(*name*, *low*, *high*, *q*)

> Suggest a value for the discrete parameter.
>
> The value is sampled from the range $[\text{low}, \text{high}]$, and the step of discretization is q. More specifically, this method returns one of the values in the sequence $\text{low}, \text{low} + q, \text{low} + 2q, \ldots, \text{low} + kq \leq \text{high}$, where k denotes an integer. Note that $high$ may be changed due to round-off errors if q is not an integer. Please check warning messages to find the changed values.

> **Parameters**
>
> - **name** (*str*) – A parameter name.
>
> - **low** (*float*) – Lower endpoint of the range of suggested values. `low` is included in the range.
>
> - **high** (*float*) – Upper endpoint of the range of suggested values. `high` is included in the range.
>
> - **q** (*float*) – A step of discretization.

> **Returns**
>   A suggested float value.

> **Return type**
>   float

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float(…, step=…) instead.

suggest_float(*name*, *low*, *high*, ***, *step=None*, *log=False*)

> Suggest a value for the floating point parameter.

Example

Suggest a momentum, learning rate and scaling factor of learning rate for neural network training.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y, random_state=0)


def objective(trial):
    momentum = trial.suggest_float("momentum", 0.0, 1.0)
    learning_rate_init = trial.suggest_float(
        "learning_rate_init", 1e-5, 1e-3, log=True
    )
    power_t = trial.suggest_float("power_t", 0.2, 0.8, step=0.1)
    clf = MLPClassifier(
        hidden_layer_sizes=(100, 50),
        momentum=momentum,
        learning_rate_init=learning_rate_init,
        solver="sgd",
        random_state=0,
        power_t=power_t,
    )
    clf.fit(X_train, y_train)

    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=3)
```

Parameters

- **name** (*str*) – A parameter name.

- **low** (*float*) – Lower endpoint of the range of suggested values. `low` is included in the range. `low` must be less than or equal to `high`. If `log` is `True`, `low` must be larger than 0.

- **high** (*float*) – Upper endpoint of the range of suggested values. `high` is included in the range. `high` must be greater than or equal to `low`.

- **step** (*float | None*) – A step of discretization.

Note: The `step` and `log` arguments cannot be used at the same time. To set the `step` argument to a float number, set the `log` argument to `False`.

- **log** (*bool*) – A flag to sample the value from the log domain or not. If `log` is true, the value is sampled from the range in the log domain. Otherwise, the value is sampled from the range in the linear domain.

> **Note:** The `step` and `log` arguments cannot be used at the same time. To set the `log` argument to `True`, set the `step` argument to `None`.

Returns
    A suggested float value.

Return type
    float

See also:

configurations tutorial describes more details and flexible usages.

suggest_int(*name*, *low*, *high*, *, *step=1*, *log=False*)
    Suggest a value for the integer parameter.

The value is sampled from the integers in $[\text{low}, \text{high}]$.

Example

Suggest the number of trees in RandomForestClassifier.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)


def objective(trial):
    n_estimators = trial.suggest_int("n_estimators", 50, 400)
    clf = RandomForestClassifier(n_estimators=n_estimators, random_state=0)
    clf.fit(X_train, y_train)
    return clf.score(X_valid, y_valid)


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=3)
```

Parameters

- **name** (*str*) – A parameter name.

- **low** (*int*) – Lower endpoint of the range of suggested values. `low` is included in the range. `low` must be less than or equal to `high`. If `log` is `True`, `low` must be larger than 0.

- **high** (*int*) – Upper endpoint of the range of suggested values. `high` is included in the range. `high` must be greater than or equal to `low`.

- **step** (*int*) – A step of discretization.

Note: Note that high is modified if the range is not divisible by step. Please check the warning messages to find the changed values.

Note: The method returns one of the values in the sequence $\text{low}, \text{low} + \text{step}, \text{low} + 2 * \text{step}, \ldots, \text{low} + k * \text{step} \leq \text{high}$, where k denotes an integer.

Note: The step != 1 and log arguments cannot be used at the same time. To set the step argument $\text{step} \geq 2$, set the log argument to False.

- **log** (*bool*) – A flag to sample the value from the log domain or not.

Note: If log is true, at first, the range of suggested values is divided into grid points of width 1. The range of suggested values is then converted to a log domain, from which a value is sampled. The uniformly sampled value is re-converted to the original domain and rounded to the nearest grid point that we just split, and the suggested value is determined. For example, if *low = 2* and *high = 8*, then the range of suggested values is *[2, 3, 4, 5, 6, 7, 8]* and lower values tend to be more sampled than higher values.

Note: The step != 1 and log arguments cannot be used at the same time. To set the log argument to True, set the step argument to 1.

Return type
    int

See also:

configurations tutorial describes more details and flexible usages.

suggest_loguniform(*name*, *low*, *high*)

    Suggest a value for the continuous parameter.

    The value is sampled from the range $[\text{low}, \text{high})$ in the log domain. When $\text{low} = \text{high}$, the value of low will be returned.

Parameters

- **name** (*str*) – A parameter name.

- **low** (*float*) – Lower endpoint of the range of suggested values. low is included in the range.

- **high** (*float*) – Upper endpoint of the range of suggested values. high is included in the range.

Returns
    A suggested float value.

Return type
    float

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float(…, log=True) instead.

suggest_uniform(*name*, *low*, *high*)

Suggest a value for the continuous parameter.

The value is sampled from the range [low, high) in the linear domain. When low = high, the value of low will be returned.

> **Parameters**
>
> - **name** (*str*) – A parameter name.
>
> - **low** (*float*) – Lower endpoint of the range of suggested values. `low` is included in the range.
>
> - **high** (*float*) – Upper endpoint of the range of suggested values. `high` is included in the range.
>
> **Returns**
>
> A suggested float value.
>
> **Return type**
>
> float

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float instead.

property system_attrs: Dict[str, Any]

Return system attributes.

> **Returns**
>
> A dictionary containing all system attributes.

> **Warning:** Deprecated in v3.1.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

property user_attrs: Dict[str, Any]

Return user attributes.

> **Returns**
>
> A dictionary containing all user attributes.

optuna.trial.FixedTrial

class optuna.trial.**FixedTrial**(*params*, *number=0*)

A trial class which suggests a fixed value for each parameter.

This object has the same methods as *Trial*, and it suggests pre-defined parameter values. The parameter values can be determined at the construction of the *FixedTrial* object. In contrast to *Trial*, *FixedTrial* does not depend on *Study*, and it is useful for deploying optimization results.

Example

Evaluate an objective function with parameter values given by a user.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x**2 + y


assert objective(optuna.trial.FixedTrial({"x": 1, "y": 0})) == 1
```

Note: Please refer to *Trial* for details of methods and properties.

Parameters

- **params** (*Dict[str, Any]*) – A dictionary containing all parameters.
- **number** (*int*) – A trial number. Defaults to **0**.

Methods

report(value, step)
set_system_attr(key, value)
set_user_attr(key, value)
should_prune()
suggest_categorical()
suggest_discrete_uniform(name, low, high, q)
suggest_float(name, low, high, *[, step, log])
suggest_int(name, low, high, *[, step, log])
suggest_loguniform(name, low, high)
suggest_uniform(name, low, high)

Attributes

datetime_start
distributions
number
params
system_attrs
user_attrs

set_system_attr(*key*, *value*)

> **Warning:** Deprecated in v3.1.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

> **Parameters**
> - **key** (*str*) –
> - **value** (*Any*) –

> **Return type**
>> None

suggest_discrete_uniform(*name*, *low*, *high*, *q*)

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float(…, step=…) instead.

> **Parameters**
>> - **name** (*str*) –
>> - **low** (*float*) –
>> - **high** (*float*) –
>> - **q** (*float*) –
>
> **Return type**
>> float

suggest_loguniform(*name*, *low*, *high*)

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float(…, log=True) instead.

> **Parameters**
>> - **name** (*str*) –
>> - **low** (*float*) –
>> - **high** (*float*) –
>
> **Return type**
>> float

suggest_uniform(*name*, *low*, *high*)

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float instead.

> **Parameters**
>> - **name** (*str*) –
>> - **low** (*float*) –

- **high** (*float*) –

    Return type
        float

optuna.trial.FrozenTrial

class optuna.trial.**FrozenTrial**(*number*, *state*, *value*, *datetime_start*, *datetime_complete*, *params*, *distributions*, *user_attrs*, *system_attrs*, *intermediate_values*, *trial_id*, ***, *values=None*)

Status and results of a *Trial*.

An object of this class has the same methods as *Trial*, but is not associated with, nor has any references to a *Study*.

It is therefore not possible to make persistent changes to a storage from this object by itself, for instance by using set_user_attr().

It will suggest the parameter values stored in *params* and will not sample values from any distributions.

It can be passed to objective functions (see *optimize()*) and is useful for deploying optimization results.

Example

Re-evaluate an objective function with parameter values optimized study.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)

assert objective(study.best_trial) == study.best_value
```

Note: Instances are mutable, despite the name. For instance, set_user_attr() will update user attributes of objects in-place.

Example:

Overwritten attributes.

```python
import copy
import datetime

import optuna


def objective(trial):
    x = trial.suggest_float("x", -1, 1)
```

```python
    # this user attribute always differs
    trial.set_user_attr("evaluation time", datetime.datetime.now())

    return x**2


study = optuna.create_study()
study.optimize(objective, n_trials=3)

best_trial = study.best_trial
best_trial_copy = copy.deepcopy(best_trial)

# re-evaluate
objective(best_trial)

# the user attribute is overwritten by re-evaluation
assert best_trial.user_attrs != best_trial_copy.user_attrs
```

Note: Please refer to *Trial* for details of methods and properties.

Parameters

- **number** (*int*) –
- **state** (*TrialState*) –
- **value** (*float | None*) –
- **datetime_start** (*datetime | None*) –
- **datetime_complete** (*datetime | None*) –
- **params** (*Dict[str, Any]*) –
- **distributions** (*Dict[str, BaseDistribution]*) –
- **user_attrs** (*Dict[str, Any]*) –
- **system_attrs** (*Dict[str, Any]*) –
- **intermediate_values** (*Dict[int, float]*) –
- **trial_id** (*int*) –
- **values** (*Sequence[float] | None*) –

number

Unique and consecutive number of *Trial* for each *Study*. Note that this field uses zero-based numbering.

state

TrialState of the *Trial*.

value

Objective value of the *Trial*. `value` and `values` must not be specified at the same time.

values

> Sequence of objective values of the *Trial*. The length is greater than 1 if the problem is multi-objective optimization. `value` and `values` must not be specified at the same time.

datetime_start

> Datetime where the *Trial* started.

datetime_complete

> Datetime where the *Trial* finished.

params

> Dictionary that contains suggested parameters.

distributions

> Dictionary that contains the distributions of *params*.

user_attrs

> Dictionary that contains the attributes of the *Trial* set with *optuna.trial.Trial.set_user_attr()*.

system_attrs

> Dictionary that contains the attributes of the *Trial* set with *optuna.trial.Trial.set_system_attr()*.

intermediate_values

> Intermediate objective values set with *optuna.trial.Trial.report()*.

Methods

report(value, step)	Interface of report function.
set_system_attr(key, value)	
set_user_attr(key, value)	
should_prune()	Suggest whether the trial should be pruned or not.
suggest_categorical()	
suggest_discrete_uniform(name, low, high, q)	
suggest_float(name, low, high, *[, step, log])	
suggest_int(name, low, high, *[, step, log])	
suggest_loguniform(name, low, high)	
suggest_uniform(name, low, high)	

Attributes

datetime_start	
distributions	
duration	Return the elapsed time taken to complete the trial.
last_step	Return the maximum step of *intermediate_values* in the trial.
number	
params	
system_attrs	
user_attrs	
value	
values	

property duration: [timedelta](#) | [None](#)

> Return the elapsed time taken to complete the trial.
>
> > **Returns**
> > The duration.

property last_step: [int](#) | [None](#)

> Return the maximum step of *intermediate_values* in the trial.
>
> > **Returns**
> > The maximum step of intermediates.

report(*value*, *step*)

> Interface of report function.
>
> Since *FrozenTrial* is not pruned, this report function does nothing.
>
> **See also:**
>
> Please refer to *should_prune()*.
>
> > **Parameters**
> > - **value** (*float*) – A value returned from the objective function.
> > - **step** (*int*) – Step of the trial (e.g., Epoch of neural network training). Note that pruners assume that `step` starts at zero. For example, *MedianPruner* simply checks if `step` is less than n_warmup_steps as the warmup mechanism.
> >
> > **Return type**
> > None

set_system_attr(*key*, *value*)

> **Warning:** Deprecated in v3.1.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.1.0.

> **Parameters**
> - **key** (*str*) –
> - **value** (*Any*) –
>
> **Return type**
>   None

should_prune()

>   Suggest whether the trial should be pruned or not.
>
>   The suggestion is always `False` regardless of a pruning algorithm.
>
> ---
>
> **Note:** *FrozenTrial* only samples one combination of parameters.
>
> ---
>
> **Returns**
>   `False`.
>
> **Return type**
>   bool

suggest_discrete_uniform(*name*, *low*, *high*, *q*)

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float(…, step=…) instead.

> **Parameters**
> - **name** (*str*) –
> - **low** (*float*) –
> - **high** (*float*) –
> - **q** (*float*) –
>
> **Return type**
>   float

suggest_loguniform(*name*, *low*, *high*)

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

Use suggest_float(..., log=True) instead.

> **Parameters**
>
> - **name** (*str*) –
>
> - **low** (*float*) –
>
> - **high** (*float*) –
>
> **Return type**
>   float

suggest_uniform(*name*, *low*, *high*)

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v6.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.
>
> Use suggest_float instead.

> **Parameters**
>
> - **name** (*str*) –
>
> - **low** (*float*) –
>
> - **high** (*float*) –
>
> **Return type**
>   float

optuna.trial.TrialState

class optuna.trial.**TrialState**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

> State of a *Trial*.
>
> **RUNNING**
>   The *Trial* is running.
>
> **WAITING**
>   The *Trial* is waiting and unfinished.
>
> **COMPLETE**
>   The *Trial* has been finished without any error.
>
> **PRUNED**
>   The *Trial* has been pruned with *TrialPruned*.
>
> **FAIL**
>   The *Trial* has failed due to an uncaught error.

Methods

is_finished()	Return a bool value to represent whether the trial state is unfinished or not.
as_integer_ratio()	Return integer ratio.
bit_count()	Number of ones in the binary representation of the absolute value of self.
bit_length()	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes([byteorder, signed])	Return the integer represented by the given array of bytes.
to_bytes([length, byteorder, signed])	Return an array of bytes representing an integer.

Attributes

RUNNING	
COMPLETE	
PRUNED	
FAIL	
WAITING	
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

as_integer_ratio()

> Return integer ratio.
>
> Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

> Number of ones in the binary representation of the absolute value of self.
>
> Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes(*byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

> **bytes**
>
> Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.
>
> **byteorder**
>
> The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.
>
> **signed**
>
> Indicates whether two's complement is used to represent the integer.

imag

the imaginary part of a complex number

is_finished()

Return a bool value to represent whether the trial state is unfinished or not.

The unfinished state is either `RUNNING` or `WAITING`.

> **Return type**
>
> [bool](#)

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

> **length**
>
> Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.
>
> **byteorder**
>
> The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte

array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
> Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

optuna.trial.create_trial

optuna.trial.**create_trial**(*, *state=TrialState.COMPLETE*, *value=None*, *values=None*, *params=None*, *distributions=None*, *user_attrs=None*, *system_attrs=None*, *intermediate_values=None*)

> Create a new *FrozenTrial*.

Example

```python
import optuna
from optuna.distributions import CategoricalDistribution
from optuna.distributions import FloatDistribution

trial = optuna.trial.create_trial(
    params={"x": 1.0, "y": 0},
    distributions={
        "x": FloatDistribution(0, 10),
        "y": CategoricalDistribution([-1, 0, 1]),
    },
    value=5.0,
)

assert isinstance(trial, optuna.trial.FrozenTrial)
assert trial.value == 5.0
assert trial.params == {"x": 1.0, "y": 0}
```

See also:

See *add_trial()* for how this function can be used to create a study from existing trials.

Note: Please note that this is a low-level API. In general, trials that are passed to objective functions are created inside *optimize()*.

Note: When `state` is *TrialState.COMPLETE*, the following parameters are required:

- `params`
- `distributions`
- `value` or `values`

> **Parameters**
> - **state** (*TrialState*) – Trial state.

- **value** (*float | None*) – Trial objective value. Must be specified if `state` is
  TrialState.COMPLETE. `value` and `values` must not be specified at the same time.

- **values** (*Sequence[float] | None*) – Sequence of the trial objective values. The length
  is greater than 1 if the problem is multi-objective optimization. Must be specified if `state`
  is *TrialState.COMPLETE*. `value` and `values` must not be specified at the same time.

- **params** (*Dict[str, Any] | None*) – Dictionary with suggested parameters of the trial.

- **distributions** (*Dict[str, BaseDistribution] | None*) – Dictionary with parameter distributions of the trial.

- **user_attrs** (*Dict[str, Any] | None*) – Dictionary with user attributes.

- **system_attrs** (*Dict[str, Any] | None*) – Dictionary with system attributes. Should
  not have to be used for most users.

- **intermediate_values** (*Dict[int, float] | None*) – Dictionary with intermediate
  objective values of the trial.

Returns
  Created trial.

Return type
  FrozenTrial

7.3.16 optuna.visualization

The *visualization* module provides utility functions for plotting the optimization process using plotly and matplotlib. Plotting functions generally take a *Study* object and optional parameters are passed as a list to the `params` argument.

Note: In the *optuna.visualization* module, the following functions use plotly to create figures, but JupyterLab cannot render them by default. Please follow this installation guide to show figures in JupyterLab.

Note: The *plot_param_importances()* requires the Python package of scikit-learn.

---

optuna.visualization.plot_contour	Plot the parameter relationship as contour plot in a study.
optuna.visualization.plot_edf	Plot the objective value EDF (empirical distribution function) of a study.
optuna.visualization. plot_hypervolume_history	Plot hypervolume history of all trials in a study.
optuna.visualization. plot_intermediate_values	Plot intermediate values of all trials in a study.
optuna.visualization. plot_optimization_history	Plot optimization history of all trials in a study.
optuna.visualization. plot_parallel_coordinate	Plot the high-dimensional parameter relationships in a study.
optuna.visualization. plot_param_importances	Plot hyperparameter importances.
optuna.visualization.plot_pareto_front	Plot the Pareto front of a study.
optuna.visualization.plot_rank	Plot parameter relations as scatter plots with colors indicating ranks of target value.
optuna.visualization.plot_slice	Plot the parameter relationship as slice plot in a study.
optuna.visualization. plot_terminator_improvement	Plot the potentials for future objective improvement.
optuna.visualization.plot_timeline	Plot the timeline of a study.
optuna.visualization.is_available	Returns whether visualization with plotly is available or not.

optuna.visualization.plot_contour

optuna.visualization.**plot_contour**(*study*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

Plot the parameter relationship as contour plot in a study.

Note that, if a parameter contains missing values, a trial with missing values is not plotted.
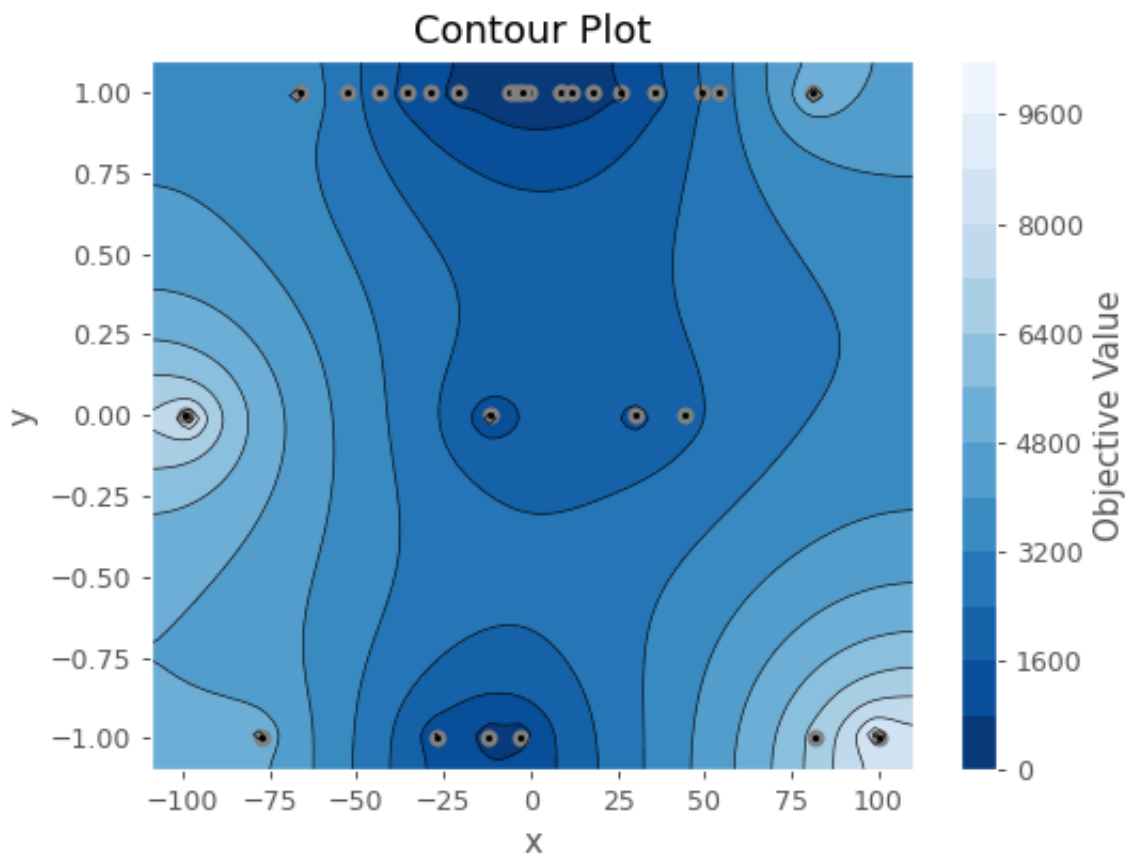
Example

The following code snippet shows how to plot the parameter relationship as contour plot.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=30)

fig = optuna.visualization.plot_contour(study, params=["x", "y"])
fig.show()
```

Parameters

- **study** (Study) – A *Study* object whose trials are plotted for their target values.

- **params** (*list[str]* | *None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float]* | *None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the color bar.

Returns
A plotly.graph_objects.Figure object.

Return type
Figure

Note: The colormap is reversed when the target argument isn't None or direction of *Study* is minimize.

---

optuna.visualization.plot_edf

optuna.visualization.**plot_edf**(*study*, *, *target=None*, *target_name='Objective Value'*)

Plot the objective value EDF (empirical distribution function) of a study.

Note that only the complete trials are considered when plotting the EDF.

Note: EDF is useful to analyze and improve search spaces. For instance, you can see a practical use case of EDF in the paper Designing Network Design Spaces.

Note: The plotted EDF assumes that the value of the objective function is in accordance with the uniform distribution over the objective space.

Example

The following code snippet shows how to plot EDF.

```python
import math

import optuna


def ackley(x, y):
    a = 20 * math.exp(-0.2 * math.sqrt(0.5 * (x ** 2 + y ** 2)))
    b = math.exp(0.5 * (math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y)))
    return -a - b + math.e + 20
```

```python
def objective(trial, low, high):
    x = trial.suggest_float("x", low, high)
    y = trial.suggest_float("y", low, high)
    return ackley(x, y)


sampler = optuna.samplers.RandomSampler(seed=10)

# Widest search space.
study0 = optuna.create_study(study_name="x=[0,5), y=[0,5)", sampler=sampler)
study0.optimize(lambda t: objective(t, 0, 5), n_trials=500)

# Narrower search space.
study1 = optuna.create_study(study_name="x=[0,4), y=[0,4)", sampler=sampler)
study1.optimize(lambda t: objective(t, 0, 4), n_trials=500)

# Narrowest search space but it doesn't include the global optimum point.
study2 = optuna.create_study(study_name="x=[1,3), y=[1,3)", sampler=sampler)
study2.optimize(lambda t: objective(t, 1, 3), n_trials=500)

fig = optuna.visualization.plot_edf([study0, study1, study2])
fig.show()
```
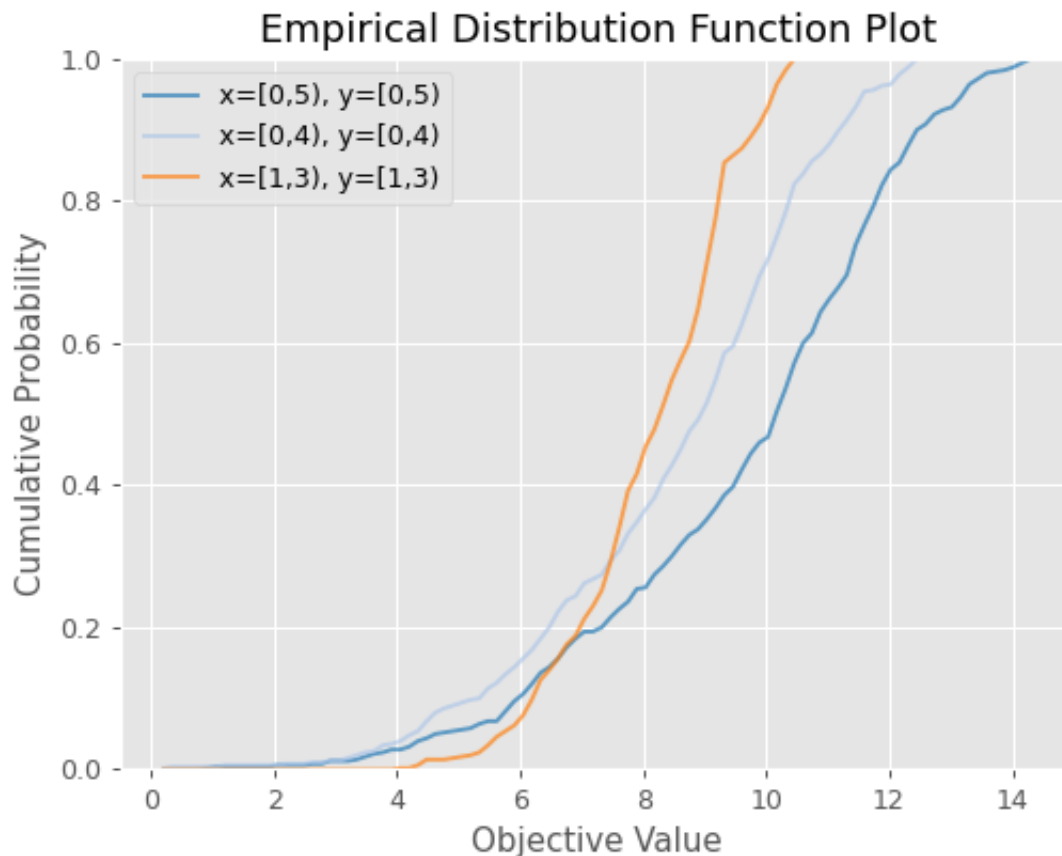
Parameters

- **study** (`Study | Sequence[Study]`) – A target *Study* object. You can pass multiple studies if you want to compare those EDFs.

- **target** (`Callable[[FrozenTrial], float] | None`) – A function to specify the value to display. If it is `None` and `study` is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if `study` is being used for multi-objective optimization.

- **target_name** (`str`) – Target's name to display on the axis label.

Returns

A `plotly.graph_objects.Figure` object.

Return type

Figure

optuna.visualization.plot_hypervolume_history

optuna.visualization.**plot_hypervolume_history**(*study*, *reference_point*)

Plot hypervolume history of all trials in a study.

Example

The following code snippet shows how to plot optimization history.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 5)
    y = trial.suggest_float("y", 0, 3)

    v0 = 4 * x ** 2 + 4 * y ** 2
    v1 = (x - 5) ** 2 + (y - 5) ** 2
    return v0, v1


study = optuna.create_study(directions=["minimize", "minimize"])
study.optimize(objective, n_trials=50)

reference_point=[100., 50.]
fig = optuna.visualization.plot_hypervolume_history(study, reference_point)
fig.show()
```

   Parameters

   - **study** (*Study*) – A *Study* object whose trials are plotted for their hypervolumes. The number of objectives must be 2 or more.

   - **reference_point** (*Sequence[float]*) – A reference point to use for hypervolume computation. The dimension of the reference point must be the same as the number of objectives.

   Returns
       A `plotly.graph_objects.Figure` object.

   Return type
       Figure

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

---

optuna.visualization.plot_intermediate_values

optuna.visualization.**plot_intermediate_values**(*study*)
   Plot intermediate values of all trials in a study.

Example

The following code snippet shows how to plot intermediate values.

```python
import optuna


def f(x):
    return (x - 2) ** 2


def df(x):
    return 2 * x - 4


def objective(trial):
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)

    x = 3
    for step in range(128):
        y = f(x)

        trial.report(y, step=step)
        if trial.should_prune():
            raise optuna.TrialPruned()

        gy = df(x)
        x -= gy * lr

    return y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=16)

fig = optuna.visualization.plot_intermediate_values(study)
fig.show()
```
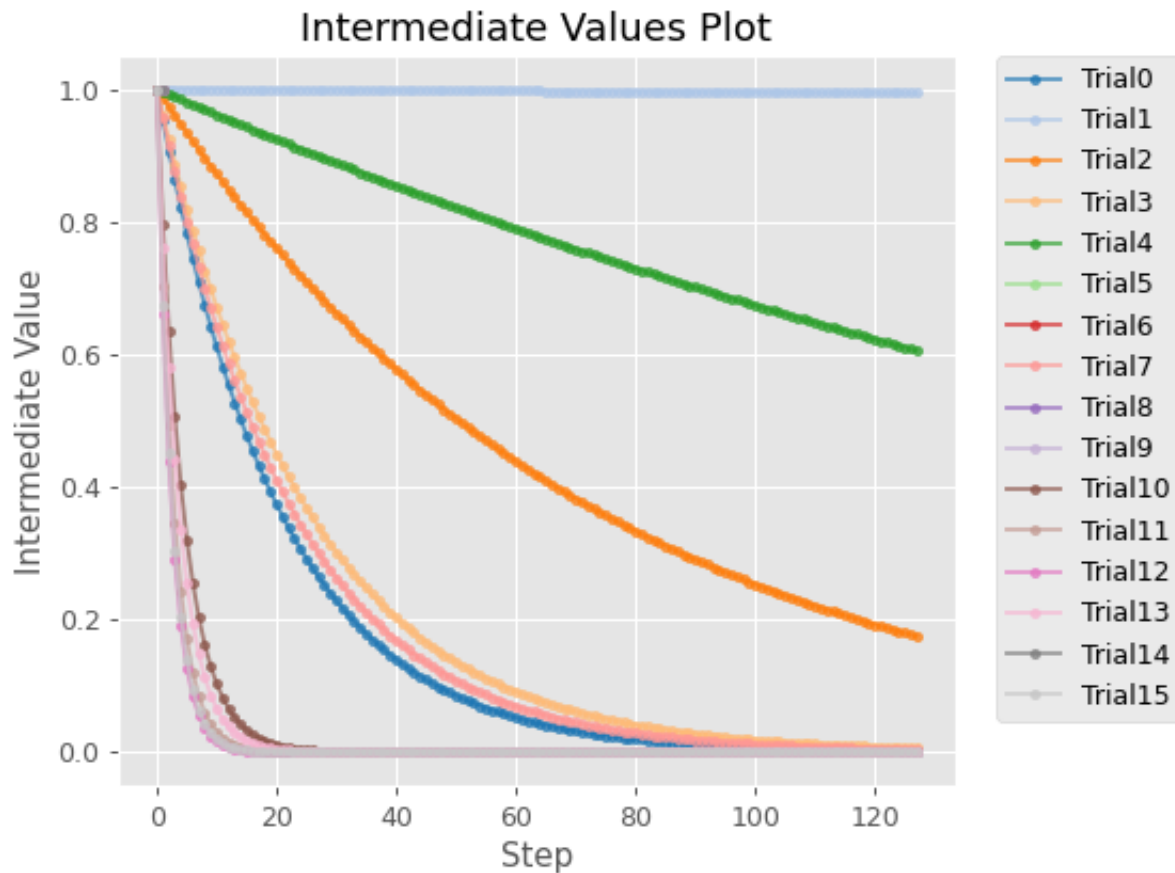
Parameters
  study (Study) – A *Study* object whose trials are plotted for their intermediate values.

Returns
  A `plotly.graph_objects.Figure` object.

Return type
  Figure

optuna.visualization.plot_optimization_history

optuna.visualization.**plot_optimization_history**(*study*, *, *target=None*, *target_name='Objective Value'*, *error_bar=False*)

Plot optimization history of all trials in a study.

Example

The following code snippet shows how to plot optimization history.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)

fig = optuna.visualization.plot_optimization_history(study)
fig.show()
```
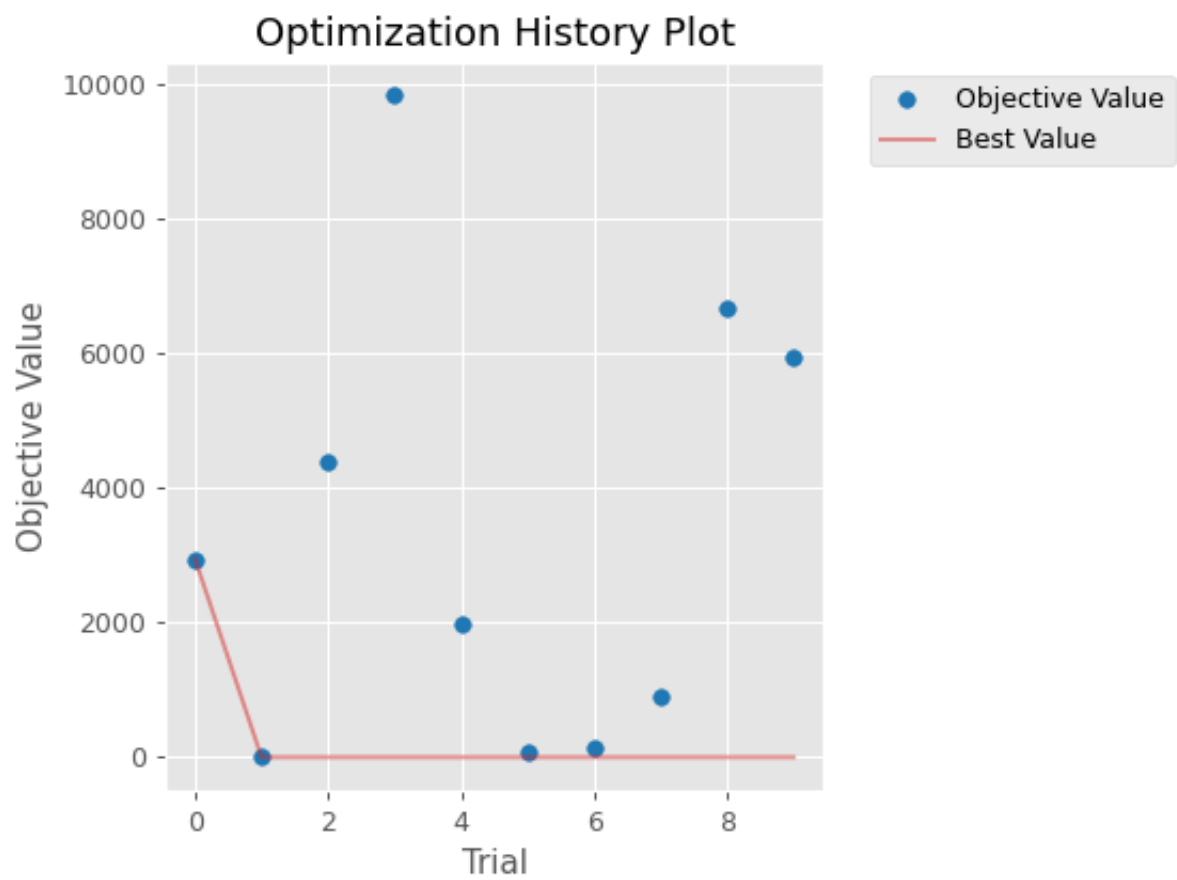
Parameters

- **study** (`Study` | `Sequence[Study]`) – A *Study* object whose trials are plotted for their target values. You can pass multiple studies if you want to compare those optimization histories.

- **target** (`Callable[[FrozenTrial], float]` | *None*) – A function to specify the value to display. If it is `None` and `study` is being used for single-objective optimization, the objective values are plotted.

    Note: Specify this argument if `study` is being used for multi-objective optimization.

- **target_name** (`str`) – Target's name to display on the axis label and the legend.

- **error_bar** (`bool`) – A flag to show the error bar.

Returns
  A `plotly.graph_objects.Figure` object.

Return type
  Figure

optuna.visualization.plot_parallel_coordinate

optuna.visualization.**plot_parallel_coordinate**(*study*, *params=None*, *, *target=None*,
                                                       target_name='Objective Value')

Plot the high-dimensional parameter relationships in a study.

Note that, if a parameter contains missing values, a trial with missing values is not plotted.

Example

The following code snippet shows how to plot the high-dimensional parameter relationships.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)

fig = optuna.visualization.plot_parallel_coordinate(study, params=["x", "y"])
fig.show()
```
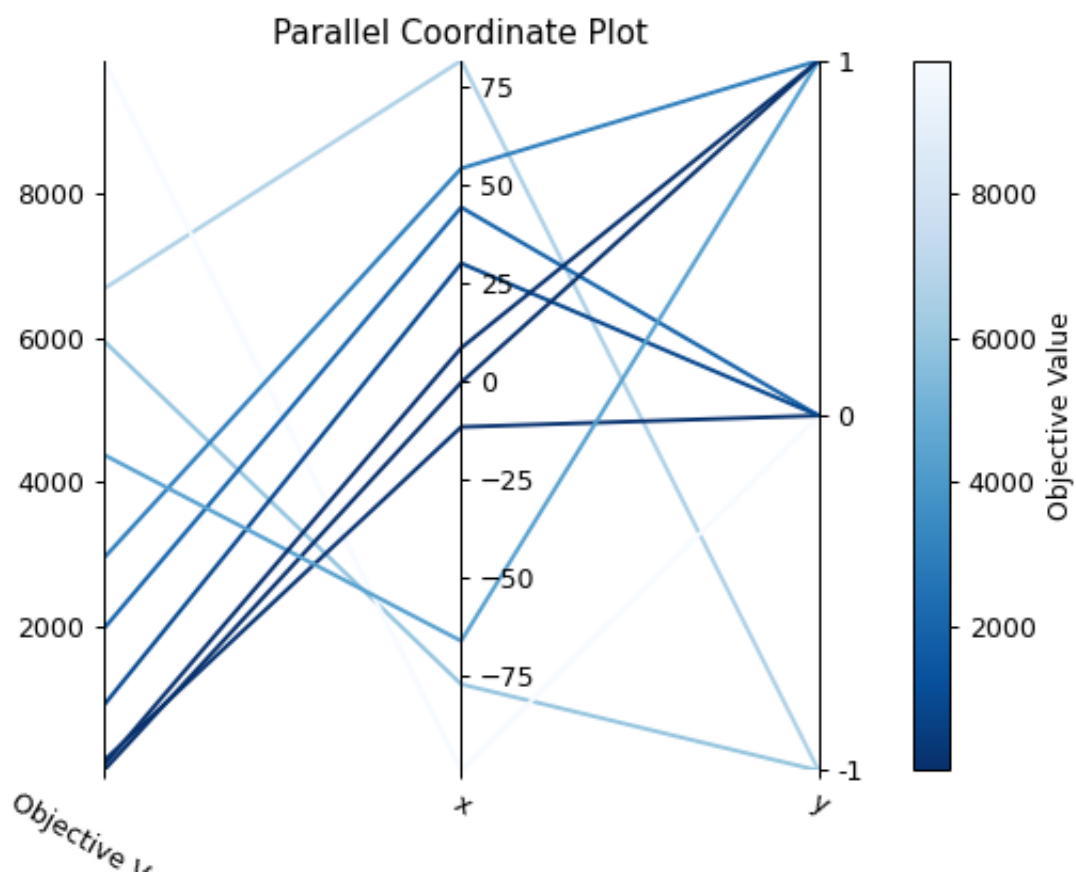
Parameters

- **study** (*Study*) – A *Study* object whose trials are plotted for their target values.

- **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

    Note: Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the axis label and the legend.

Returns
    A plotly.graph_objects.Figure object.

Return type
    Figure

Note: The colormap is reversed when the target argument isn't None or direction of *Study* is minimize.

---

optuna.visualization.plot_param_importances

optuna.visualization.**plot_param_importances**(*study*, *evaluator=None*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

Plot hyperparameter importances.

Example

The following code snippet shows how to plot hyperparameter importances.

```python
import optuna


def objective(trial):
    x = trial.suggest_int("x", 0, 2)
    y = trial.suggest_float("y", -1.0, 1.0)
    z = trial.suggest_float("z", 0.0, 1.5)
    return x ** 2 + y ** 3 - z ** 4


sampler = optuna.samplers.RandomSampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=100)

fig = optuna.visualization.plot_param_importances(study)
fig.show()
```

See also:

This function visualizes the results of *optuna.importance.get_param_importances()*.

> **Parameters**
>
> - **study** (Study) – An optimized study.
>
> - **evaluator** (*BaseImportanceEvaluator | None*) – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *FanovaImportanceEvaluator*.
>
>   ---
>
>   **Note:** *FanovaImportanceEvaluator* takes over 1 minute when given a study that contains 1000+ trials. We published optuna-fast-fanova library, that is a Cython accelerated fANOVA implementation. By using it, you can get hyperparameter importances within a few seconds.
>
>   ---
>
> - **params** (*list[str] | None*) – A list of names of parameters to assess. If None, all parameters that are present in all of the completed trials are assessed.
>
> - **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted. For multi-objective optimization, all objectives will be plotted if target is None.
>
>   ---
>
>   **Note:** This argument can be used to specify which objective to plot if study is being used for multi-objective optimization. For example, to get only the hyperparameter importance of

the first objective, use `target=lambda t:  t.values[0]` for the target parameter.

- **target_name** (*str*) – Target's name to display on the legend.   Names set via
  set_metric_names() will be used if `target` is `None`, overriding this argument.

Returns
> A `plotly.graph_objects.Figure` object.

Return type
> *Figure*

optuna.visualization.plot_pareto_front

optuna.visualization.**plot_pareto_front**(*study*, ***, *target_names=None*, *include_dominated_trials=True*, *axis_order=None*, *constraints_func=None*, *targets=None*)

Plot the Pareto front of a study.

See also:

Please refer to multi_objective for the tutorial of the Pareto front visualization.

Example

The following code snippet shows how to plot the Pareto front of a study.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 5)
    y = trial.suggest_float("y", 0, 3)

    v0 = 4 * x ** 2 + 4 * y ** 2
    v1 = (x - 5) ** 2 + (y - 5) ** 2
    return v0, v1


study = optuna.create_study(directions=["minimize", "minimize"])
study.optimize(objective, n_trials=50)

fig = optuna.visualization.plot_pareto_front(study)
fig.show()
```

Example

The following code snippet shows how to plot a 2-dimensional Pareto front of a 3-dimensional study. This example is scalable, e.g., for plotting a 2- or 3-dimensional Pareto front of a 4-dimensional study and so on.

```python
import optuna

def objective(trial):
    x = trial.suggest_float("x", 0, 5)
```

```python
    y = trial.suggest_float("y", 0, 3)
    v0 = 5 * x ** 2 + 3 * y ** 2
    v1 = (x - 10) ** 2 + (y - 10) ** 2
    v2 = x + y

    return v0, v1, v2


study = optuna.create_study(directions=["minimize", "minimize", "minimize"])

study.optimize(objective, n_trials=100)

fig = optuna.visualization.plot_pareto_front(
    study,
    targets=lambda t: (t.values[0], t.values[1]),
    target_names=["Objective 0", "Objective 1"],
)

fig.show()
```
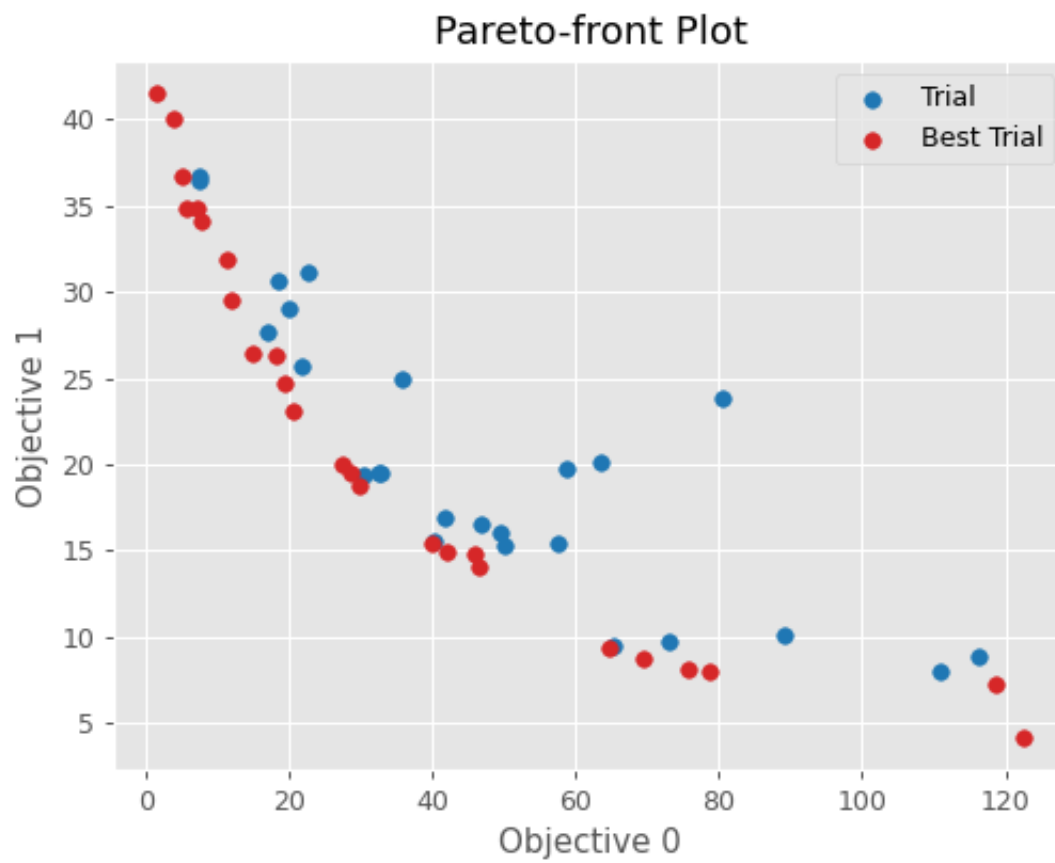
Parameters

- **study** (*Study*) – A *Study* object whose trials are plotted for their objective values. The number of objectives must be either 2 or 3 when `targets` is None.

- **target_names** (*list[str] | None*) – Objective name list used as the axis titles. If None is specified, "Objective {objective_index}" is used instead. If `targets` is specified for a study that does not contain any completed trial, `target_name` must be specified.

- **include_dominated_trials** (*bool*) – A flag to include all dominated trial's objective values.

- **axis_order** (*list[int] | None*) – A list of indices indicating the axis order. If None is specified, default order is used. `axis_order` and `targets` cannot be used at the same time.

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

- **constraints_func** (*Callable[[FrozenTrial], Sequence[float]] | None*) – An optional function that computes the objective constraints. It must take a *FrozenTrial* and return the constraints. The return value must be a sequence of *float* s. A value strictly larger than 0 means that a constraint is violated. A value equal to or smaller than 0 is considered feasible. This specification is the same as in, for example, *NSGAIISampler*.

  If given, trials are classified into three categories: feasible and best, feasible but non-best, and infeasible. Categories are shown in different colors. Here, whether a trial is best (on Pareto front) or not is determined ignoring all infeasible trials.

  > **Note:** Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

- **targets** (*Callable[[FrozenTrial], Sequence[float]] | None*) – A function that

returns targets values to display. The argument to this function is *FrozenTrial*. `axis_order` and `targets` cannot be used at the same time. If `study.n_objectives` is neither 2 nor 3, `targets` must be specified.

Note: Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

> **Returns**
>   A `plotly.graph_objects.Figure` object.
>
> **Return type**
>   *Figure*

optuna.visualization.plot_rank

optuna.visualization.**plot_rank**(*study*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

> Plot parameter relations as scatter plots with colors indicating ranks of target value.
>
> Note that trials missing the specified parameters will not be plotted.
>
> ### Example
>
> The following code snippet shows how to plot the parameter relationship as a rank plot.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])

    c0 = 400 - (x + y)**2
    trial.set_user_attr("constraint", [c0])

    return x ** 2 + y


def constraints(trial):
    return trial.user_attrs["constraint"]


sampler = optuna.samplers.TPESampler(seed=10, constraints_func=constraints)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=30)

fig = optuna.visualization.plot_rank(study, params=["x", "y"])
fig.show()
```

> **Parameters**
>
> - **study** (*Study*) – A *Study* object whose trials are plotted for their target values.
>
> - **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if study is being used for multi-objective optimization.

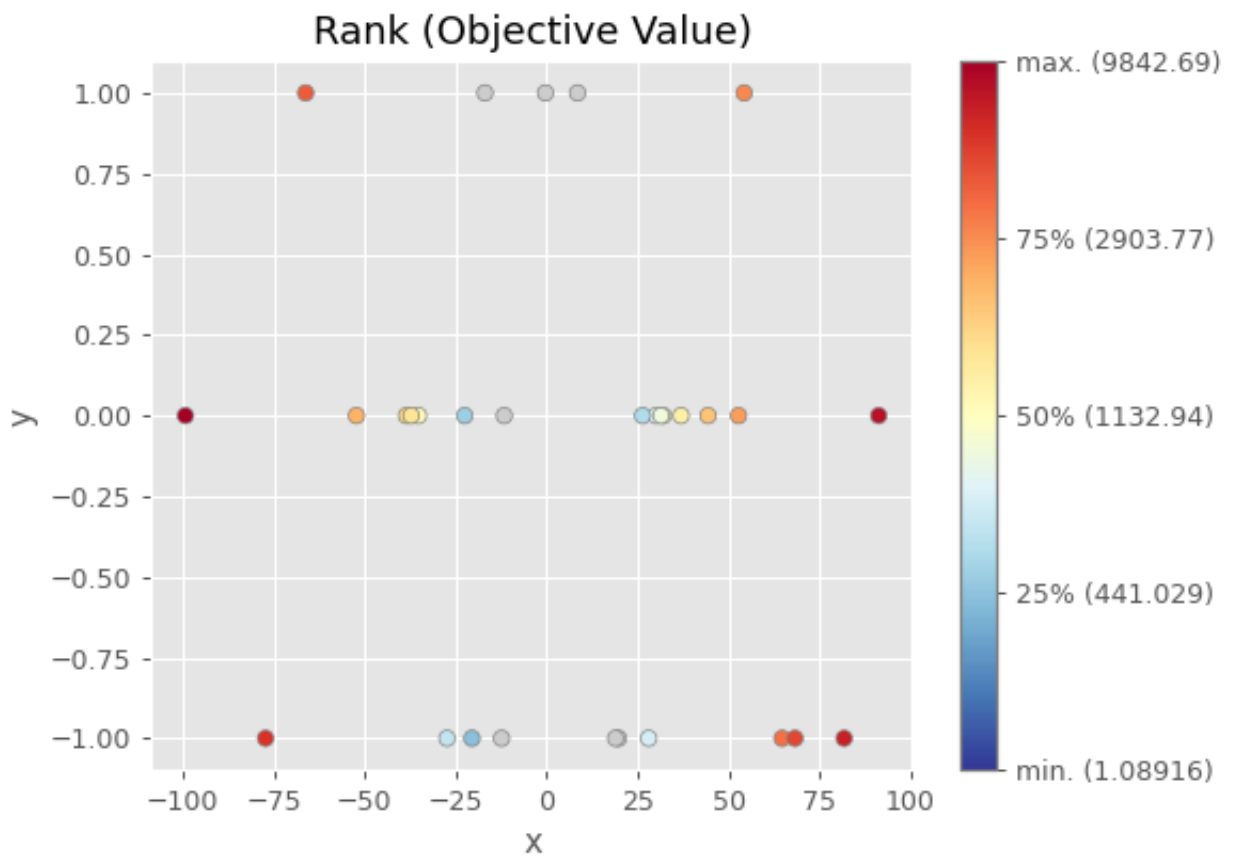- **target_name** (*str*) – Target's name to display on the color bar.

Returns
   A plotly.graph_objects.Figure object.

Return type
   Figure

Note: This function requires plotly >= 5.0.0.

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

optuna.visualization.plot_slice

optuna.visualization.**plot_slice**(*study*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

   Plot the parameter relationship as slice plot in a study.

   Note that, if a parameter contains missing values, a trial with missing values is not plotted.

Example

The following code snippet shows how to plot the parameter relationship as slice plot.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)

fig = optuna.visualization.plot_slice(study, params=["x", "y"])
fig.show()
```

   Parameters

- **study** (*Study*) – A *Study* object whose trials are plotted for their target values.

- **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

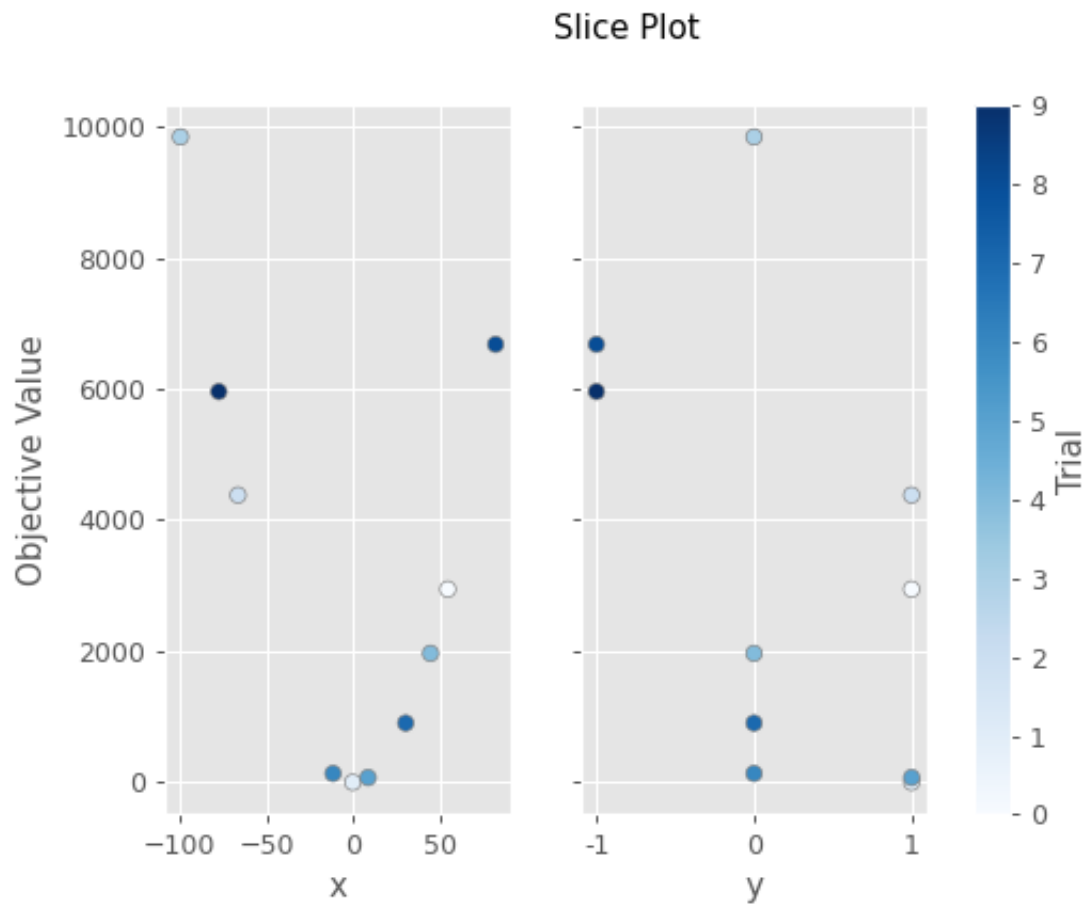> **Note:** Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the axis label.

Returns
> A plotly.graph_objects.Figure object.

Return type
> *Figure*

optuna.visualization.plot_terminator_improvement

optuna.visualization.**plot_terminator_improvement**(*study*, *plot_error=False*, *improvement_evaluator=None*, *error_evaluator=None*, *min_n_trials=20*)

Plot the potentials for future objective improvement.

This function visualizes the objective improvement potentials, evaluated with improvement_evaluator. It helps to determine whether we should continue the optimization or not. You can also plot the error evaluated with error_evaluator if the plot_error argument is set to True. Note that this function may take some time to compute the improvement potentials.

Example

The following code snippet shows how to plot improvement potentials, together with cross-validation errors.

```python
from lightgbm import LGBMClassifier
from sklearn.datasets import load_wine
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
import optuna
from optuna.terminator import report_cross_validation_scores
from optuna.visualization import plot_terminator_improvement


def objective(trial):
    X, y = load_wine(return_X_y=True)
    clf = LGBMClassifier(
        reg_alpha=trial.suggest_float("reg_alpha", 1e-8, 10.0, log=True),
        reg_lambda=trial.suggest_float("reg_lambda", 1e-8, 10.0, log=True),
        num_leaves=trial.suggest_int("num_leaves", 2, 256),
        colsample_bytree=trial.suggest_float("colsample_bytree", 0.4, 1.0),
        subsample=trial.suggest_float("subsample", 0.4, 1.0),
        subsample_freq=trial.suggest_int("subsample_freq", 1, 7),
        min_child_samples=trial.suggest_int("min_child_samples", 5, 100),
    )
    scores = cross_val_score(clf, X, y, cv=KFold(n_splits=5, shuffle=True))
```

```
    report_cross_validation_scores(trial, scores)
    return scores.mean()


study = optuna.create_study()
study.optimize(objective, n_trials=30)

fig = plot_terminator_improvement(study, plot_error=True)
fig.show()
```

>   **Parameters**
>
>   • **study** (Study) – A *Study* object whose trials are plotted for their improvement.
>
>   • **plot_error** (*bool*) – A flag to show the error. If it is set to True, errors evaluated by error_evaluator are also plotted as line graph. Defaults to False.
>
>   • **improvement_evaluator** (BaseImprovementEvaluator | *None*) – An object that evaluates the improvement of the objective function. Defaults to *RegretBoundEvaluator*.
>
>   • **error_evaluator** (BaseErrorEvaluator | *None*) – An object that evaluates the error inherent in the objective function. Defaults to *CrossValidationErrorEvaluator*.
>
>   • **min_n_trials** (*int*) – The minimum number of trials before termination is considered. Terminator improvements for trials below this value are shown in a lighter color. Defaults to 20.
>
>   **Returns**
>   A plotly.graph_objects.Figure object.
>
>   **Return type**
>   *Figure*

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

optuna.visualization.plot_timeline

optuna.visualization.**plot_timeline**(*study*)

>   Plot the timeline of a study.

Example

The following code snippet shows how to plot the timeline of a study. Timeline plot can visualize trials with overlapping execution time (e.g., in distributed environments).

```
import time

import optuna
```

```
def objective(trial):
    x = trial.suggest_float("x", 0, 1)
    time.sleep(x * 0.1)
    if x > 0.8:
        raise ValueError()
    if x > 0.4:
        raise optuna.TrialPruned()
    return x ** 2


study = optuna.create_study(direction="minimize")
study.optimize(
    objective, n_trials=50, n_jobs=2, catch=(ValueError,)
)

fig = optuna.visualization.plot_timeline(study)
fig.show()
```
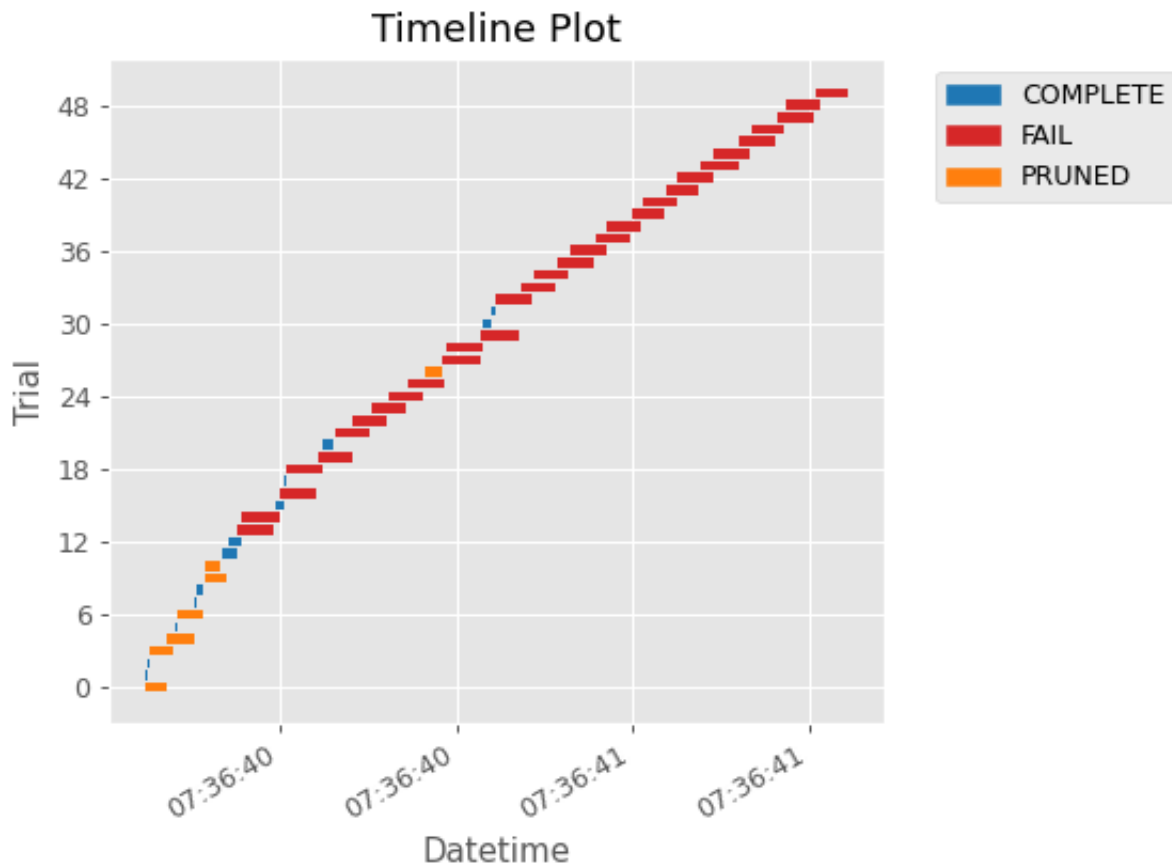
> **Parameters**
> > **study** (Study) – A *Study* object whose trials are plotted with their lifetime.
>
> **Returns**
> > A `plotly.graph_objects.Figure` object.
>
> **Return type**
> > *Figure*

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

optuna.visualization.is_available

optuna.visualization.**is_available**()

> Returns whether visualization with plotly is available or not.

Note: *visualization* module depends on plotly version 4.0.0 or higher. If a supported version of plotly isn't installed in your environment, this function will return `False`. In such case, please execute `$ pip install -U plotly>=4.0.0` to install plotly.

> **Returns**
> > `True` if visualization with plotly is available, `False` otherwise.
>
> **Return type**
> > bool

Note: The following *optuna.visualization.matplotlib* module uses Matplotlib as a backend.

---

optuna.visualization.matplotlib

Note: The following functions use Matplotlib as a backend.

`optuna.visualization.matplotlib.plot_contour`	Plot the parameter relationship as contour plot in a study with Matplotlib.
`optuna.visualization.matplotlib.plot_edf`	Plot the objective value EDF (empirical distribution function) of a study with Matplotlib.
`optuna.visualization.matplotlib.plot_hypervolume_history`	Plot hypervolume history of all trials in a study with Matplotlib.
`optuna.visualization.matplotlib.plot_intermediate_values`	Plot intermediate values of all trials in a study with Matplotlib.
`optuna.visualization.matplotlib.plot_optimization_history`	Plot optimization history of all trials in a study with Matplotlib.
`optuna.visualization.matplotlib.plot_parallel_coordinate`	Plot the high-dimensional parameter relationships in a study with Matplotlib.
`optuna.visualization.matplotlib.plot_param_importances`	Plot hyperparameter importances with Matplotlib.
`optuna.visualization.matplotlib.plot_pareto_front`	Plot the Pareto front of a study.
`optuna.visualization.matplotlib.plot_rank`	Plot parameter relations as scatter plots with colors indicating ranks of target value.
`optuna.visualization.matplotlib.plot_slice`	Plot the parameter relationship as slice plot in a study with Matplotlib.
`optuna.visualization.matplotlib.plot_terminator_improvement`	Plot the potentials for future objective improvement.
`optuna.visualization.matplotlib.plot_timeline`	Plot the timeline of a study.
`optuna.visualization.matplotlib.is_available`	Returns whether visualization with Matplotlib is available or not.

optuna.visualization.matplotlib.plot_contour

optuna.visualization.matplotlib.**plot_contour**(*study*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

Plot the parameter relationship as contour plot in a study with Matplotlib.

Note that, if a parameter contains missing values, a trial with missing values is not plotted.

See also:

Please refer to *optuna.visualization.plot_contour()* for an example.

Warning: Output figures of this Matplotlib-based *plot_contour()* function would be different from those of the Plotly-based *plot_contour()*.

The following code snippet shows how to plot the parameter relationship as contour plot.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=30)

optuna.visualization.matplotlib.plot_contour(study, params=["x", "y"])
```


Parameters

- **study** (*Study*) – A *Study* object whose trials are plotted for their target values.

- **params** (*list[str]* | *None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the color bar.

Returns
> A matplotlib.axes.Axes object.

Return type
> *Axes*

Note: The colormap is reversed when the target argument isn't None or direction of *Study* is minimize.

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_edf

optuna.visualization.matplotlib.**plot_edf**(*study, *, target=None, target_name='Objective Value'*)

> Plot the objective value EDF (empirical distribution function) of a study with Matplotlib.

> Note that only the complete trials are considered when plotting the EDF.

> **See also:**

> Please refer to *optuna.visualization.plot_edf()* for an example, where this function can be replaced with it.

Note: Please refer to matplotlib.pyplot.legend to adjust the style of the generated legend.

Example

The following code snippet shows how to plot EDF.

```python
import math

import optuna


def ackley(x, y):
    a = 20 * math.exp(-0.2 * math.sqrt(0.5 * (x ** 2 + y ** 2)))
    b = math.exp(0.5 * (math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y)))
    return -a - b + math.e + 20
```

(continues on next page)

```python
def objective(trial, low, high):
    x = trial.suggest_float("x", low, high)
    y = trial.suggest_float("y", low, high)
    return ackley(x, y)


sampler = optuna.samplers.RandomSampler(seed=10)

# Widest search space.
study0 = optuna.create_study(study_name="x=[0,5), y=[0,5)", sampler=sampler)
study0.optimize(lambda t: objective(t, 0, 5), n_trials=500)

# Narrower search space.
study1 = optuna.create_study(study_name="x=[0,4), y=[0,4)", sampler=sampler)
study1.optimize(lambda t: objective(t, 0, 4), n_trials=500)

# Narrowest search space but it doesn't include the global optimum point.
study2 = optuna.create_study(study_name="x=[1,3), y=[1,3)", sampler=sampler)
study2.optimize(lambda t: objective(t, 1, 3), n_trials=500)

optuna.visualization.matplotlib.plot_edf([study0, study1, study2])
```

Parameters

- **study** (`Study | Sequence[Study]`) – A target *Study* object. You can pass multiple studies if you want to compare those EDFs.

- **target** (`Callable[[FrozenTrial], float] | None`) – A function to specify the value to display. If it is `None` and `study` is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if `study` is being used for multi-objective optimization.

- **target_name** (`str`) – Target's name to display on the axis label.

Returns

A `matplotlib.axes.Axes` object.

Return type

Axes

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_hypervolume_history

optuna.visualization.matplotlib.**plot_hypervolume_history**(*study*, *reference_point*)

Plot hypervolume history of all trials in a study with Matplotlib.

Example

The following code snippet shows how to plot optimization history.

```python
import optuna
import matplotlib.pyplot as plt


def objective(trial):
    x = trial.suggest_float("x", 0, 5)
    y = trial.suggest_float("y", 0, 3)

    v0 = 4 * x ** 2 + 4 * y ** 2
    v1 = (x - 5) ** 2 + (y - 5) ** 2
    return v0, v1


study = optuna.create_study(directions=["minimize", "minimize"])
study.optimize(objective, n_trials=50)

reference_point=[100, 50]
optuna.visualization.matplotlib.plot_hypervolume_history(study, reference_point)
plt.tight_layout()
```
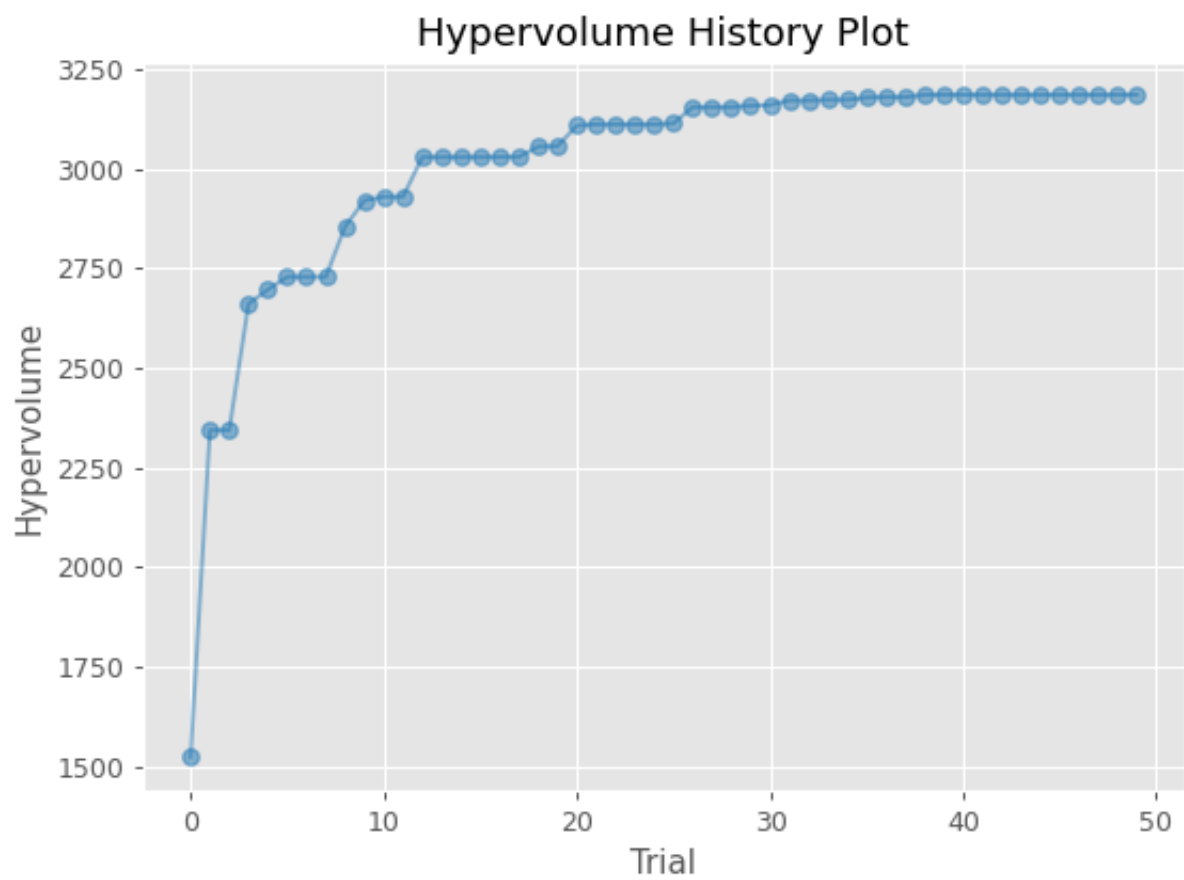
Note: You need to adjust the size of the plot by yourself using `plt.tight_layout()` or `plt.savefig(IMAGE_NAME, bbox_inches='tight')`.

Parameters

- **study** (Study) – A *Study* object whose trials are plotted for their hypervolumes. The number of objectives must be 2 or more.

- **reference_point** (*Sequence[float]*) – A reference point to use for hypervolume computation. The dimension of the reference point must be the same as the number of objectives.

Returns
    A matplotlib.axes.Axes object.

Return type
    Axes

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

---

optuna.visualization.matplotlib.plot_intermediate_values

optuna.visualization.matplotlib.**plot_intermediate_values**(*study*)

Plot intermediate values of all trials in a study with Matplotlib.

Note: Please refer to matplotlib.pyplot.legend to adjust the style of the generated legend.

Example

The following code snippet shows how to plot intermediate values.

```python
import optuna


def f(x):
    return (x - 2) ** 2


def df(x):
    return 2 * x - 4


def objective(trial):
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)

    x = 3
    for step in range(128):
        y = f(x)
```

(continues on next page)

```python
        trial.report(y, step=step)
        if trial.should_prune():
            raise optuna.TrialPruned()

        gy = df(x)
        x -= gy * lr

    return y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=16)

optuna.visualization.matplotlib.plot_intermediate_values(study)
```


See also:

Please refer to *optuna.visualization.plot_intermediate_values()* for an example.

Parameters

study (*Study*) – A *Study* object whose trials are plotted for their intermediate values.

Returns

> A `matplotlib.axes.Axes` object.

Return type

> *Axes*

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_optimization_history

optuna.visualization.matplotlib.**plot_optimization_history**(*study*, ***, *target=None*, *target_name='Objective Value'*, *error_bar=False*)

Plot optimization history of all trials in a study with Matplotlib.

See also:

Please refer to *optuna.visualization.plot_optimization_history()* for an example.

Example

The following code snippet shows how to plot optimization history.

```python
import optuna
import matplotlib.pyplot as plt


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y

sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)

optuna.visualization.matplotlib.plot_optimization_history(study)
plt.tight_layout()
```

Note: You need to adjust the size of the plot by yourself using `plt.tight_layout()` or `plt.savefig(IMAGE_NAME, bbox_inches='tight')`.

Parameters

- **study** (*Study | Sequence[Study]*) – A *Study* object whose trials are plotted for their target values. You can pass multiple studies if you want to compare those optimization histories.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the axis label and the legend.
- **error_bar** (*bool*) – A flag to show the error bar.

Returns
> A matplotlib.axes.Axes object.

Return type
> *Axes*

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_parallel_coordinate

optuna.visualization.matplotlib.**plot_parallel_coordinate**(*study*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

Plot the high-dimensional parameter relationships in a study with Matplotlib.

Note that, if a parameter contains missing values, a trial with missing values is not plotted.

See also:

Please refer to *optuna.visualization.plot_parallel_coordinate()* for an example.

Example

The following code snippet shows how to plot the high-dimensional parameter relationships.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)

optuna.visualization.matplotlib.plot_parallel_coordinate(study, params=["x", "y"])
```

Parameters

- **study** (*Study*) – A *Study* object whose trials are plotted for their target values.

Parallel Coordinate Plot

- **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

Note: Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the axis label and the legend.

Returns

A matplotlib.axes.Axes object.

Return type

Axes

Note: The colormap is reversed when the target argument isn't None or direction of *Study* is minimize.

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_param_importances

optuna.visualization.matplotlib.**plot_param_importances**(*study*, *evaluator=None*, *params=None*, *,
target=None, *target_name='Objective
Value'*)

Plot hyperparameter importances with Matplotlib.

See also:

Please refer to *optuna.visualization.plot_param_importances()* for an example.

Example

The following code snippet shows how to plot hyperparameter importances.

```python
import optuna


def objective(trial):
    x = trial.suggest_int("x", 0, 2)
    y = trial.suggest_float("y", -1.0, 1.0)
    z = trial.suggest_float("z", 0.0, 1.5)
    return x ** 2 + y ** 3 - z ** 4


sampler = optuna.samplers.RandomSampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=100)
```

(continues on next page)

```
optuna.visualization.matplotlib.plot_param_importances(study)
```
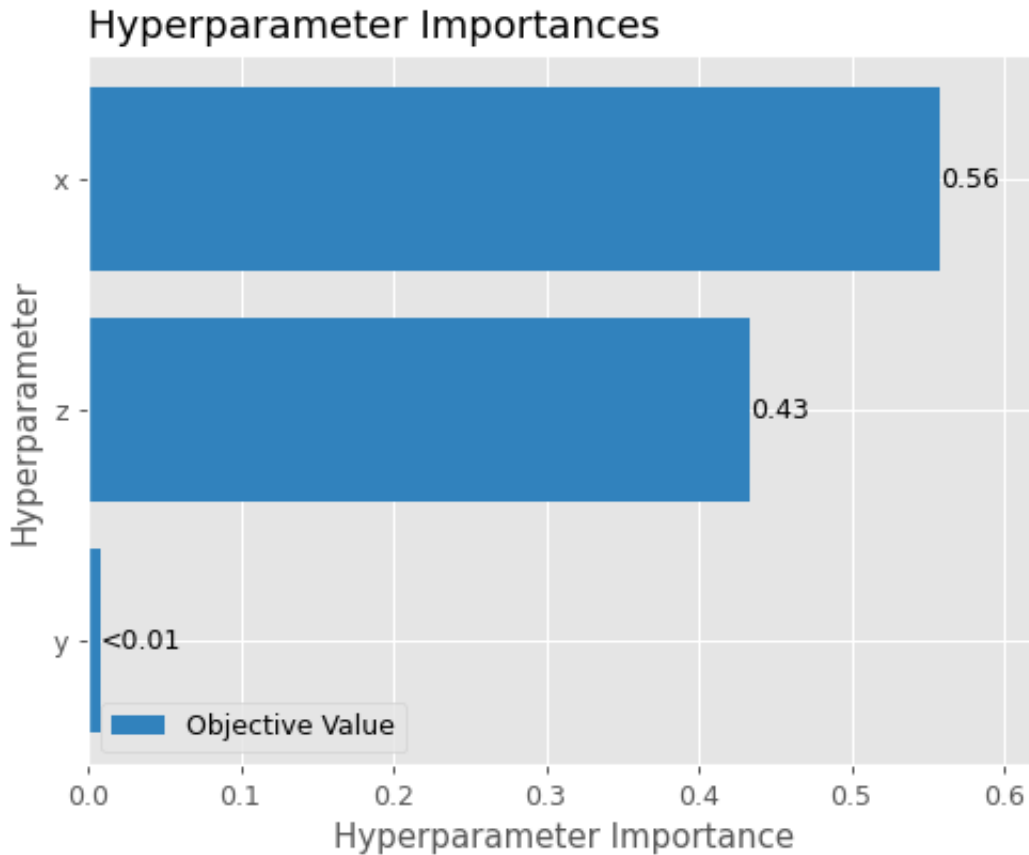

Parameters

- **study** (*Study*) – An optimized study.

- **evaluator** (*BaseImportanceEvaluator | None*) – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *FanovaImportanceEvaluator*.

- **params** (*list[str] | None*) – A list of names of parameters to assess. If *None*, all parameters that are present in all of the completed trials are assessed.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is *None* and `study` is being used for single-objective optimization, the objective values are plotted. For multi-objective optimization, all objectives will be plotted if `target` is *None*.

Note: This argument can be used to specify which objective to plot if `study` is being used for multi-objective optimization. For example, to get only the hyperparameter importance of the first objective, use `target=lambda t:  t.values[0]` for the target parameter.

- **target_name** (*str*) – Target's name to display on the axis label. Names set via *set_metric_names()* will be used if `target` is `None`, overriding this argument.

   Returns
      A `matplotlib.axes.Axes` object.

   Return type
      Axes

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_pareto_front

optuna.visualization.matplotlib.**plot_pareto_front**(*study*, *, *target_names=None*, *include_dominated_trials=True*, *axis_order=None*, *constraints_func=None*, *targets=None*)

Plot the Pareto front of a study.

See also:

Please refer to *optuna.visualization.plot_pareto_front()* for an example.

Example

The following code snippet shows how to plot the Pareto front of a study.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 5)
    y = trial.suggest_float("y", 0, 3)

    v0 = 4 * x ** 2 + 4 * y ** 2
    v1 = (x - 5) ** 2 + (y - 5) ** 2
    return v0, v1


study = optuna.create_study(directions=["minimize", "minimize"])
study.optimize(objective, n_trials=50)

optuna.visualization.matplotlib.plot_pareto_front(study)
```

   Parameters

   - **study** (*Study*) – A *Study* object whose trials are plotted for their objective values. `study.n_objectives` must be either 2 or 3 when `targets` is `None`.

   - **target_names** (*list[str] | None*) – Objective name list used as the axis titles. If `None` is specified, "Objective {objective_index}" is used instead. If `targets` is specified for a study that does not contain any completed trial, `target_name` must be specified.

- **include_dominated_trials** (*bool*) – A flag to include all dominated trial's objective values.

- **axis_order** (*list[int] | None*) – A list of indices indicating the axis order. If *None* is specified, default order is used. axis_order and targets cannot be used at the same time.

> **Warning:** Deprecated in v3.0.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v5.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

- **constraints_func** (*Callable[[FrozenTrial], Sequence[float]] | None*) – An optional function that computes the objective constraints. It must take a *FrozenTrial* and return the constraints. The return value must be a sequence of *float* s. A value strictly larger than 0 means that a constraint is violated. A value equal to or smaller than 0 is considered feasible. This specification is the same as in, for example, *NSGAIISampler*.

  If given, trials are classified into three categories: feasible and best, feasible but non-best, and infeasible. Categories are shown in different colors. Here, whether a trial is best (on Pareto front) or not is determined ignoring all infeasible trials.

- **targets** (*Callable[[FrozenTrial], Sequence[float]] | None*) – A function that returns a tuple of target values to display. The argument to this function is *FrozenTrial*. targets must be *None* or return 2 or 3 values. axis_order and targets cannot be used at the same time. If the number of objectives is neither 2 nor 3, targets must be specified.

> **Note:** Added in v3.0.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.0.0.

> **Returns**
> A matplotlib.axes.Axes object.

> **Return type**
> *Axes*

Note: Added in v2.8.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.8.0.

optuna.visualization.matplotlib.plot_rank

optuna.visualization.matplotlib.**plot_rank**(*study*, *params=None*, *, *target=None*, *target_name='Objective Value'*)

Plot parameter relations as scatter plots with colors indicating ranks of target value.

Note that trials missing the specified parameters will not be plotted.

See also:

Please refer to *optuna.visualization.plot_rank()* for an example.

> **Warning:** Output figures of this Matplotlib-based *plot_rank()* function would be different from those of the Plotly-based *plot_rank()*.

Example

The following code snippet shows how to plot the parameter relationship as a rank plot.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])

    c0 = 400 - (x + y)**2
    trial.set_user_attr("constraint", [c0])

    return x ** 2 + y


def constraints(trial):
    return trial.user_attrs["constraint"]


sampler = optuna.samplers.TPESampler(seed=10, constraints_func=constraints)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=30)

optuna.visualization.matplotlib.plot_rank(study, params=["x", "y"])
```

Parameters

- **study** (Study) – A *Study* object whose trials are plotted for their target values.

- **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.

- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.

  > **Note:** Specify this argument if study is being used for multi-objective optimization.

- **target_name** (*str*) – Target's name to display on the color bar.

Returns
  A matplotlib.axes.Axes object.

Return type
  Axes

> **Note:** Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior

notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

optuna.visualization.matplotlib.plot_slice

optuna.visualization.matplotlib.**plot_slice**(*study*, *params=None*, ***, *target=None*, *target_name='Objective Value'*)

Plot the parameter relationship as slice plot in a study with Matplotlib.

See also:

Please refer to *optuna.visualization.plot_slice()* for an example.

Example

The following code snippet shows how to plot the parameter relationship as slice plot.

```python
import optuna


def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    return x ** 2 + y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=10)

optuna.visualization.matplotlib.plot_slice(study, params=["x", "y"])
```

> **Parameters**
> - **study** (Study) – A *Study* object whose trials are plotted for their target values.
> - **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.
> - **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and study is being used for single-objective optimization, the objective values are plotted.
>
> ---
> **Note:** Specify this argument if study is being used for multi-objective optimization.
>
> ---
>
> - **target_name** (*str*) – Target's name to display on the axis label.
>
> **Returns**
> A matplotlib.axes.Axes object.
>
> **Return type**
> *Axes*

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

---

optuna.visualization.matplotlib.plot_terminator_improvement

optuna.visualization.matplotlib.**plot_terminator_improvement**(*study*, *plot_error=False*, *improvement_evaluator=None*, *error_evaluator=None*, *min_n_trials=20*)

Plot the potentials for future objective improvement.

This function visualizes the objective improvement potentials, evaluated with `improvement_evaluator`. It helps to determine whether we should continue the optimization or not. You can also plot the error evaluated with `error_evaluator` if the `plot_error` argument is set to `True`. Note that this function may take some time to compute the improvement potentials.

See also:

Please refer to *optuna.visualization.plot_terminator_improvement()*.

Example

The following code snippet shows how to plot improvement potentials, together with cross-validation errors.

```python
from lightgbm import LGBMClassifier
from sklearn.datasets import load_wine
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
import optuna
from optuna.terminator import report_cross_validation_scores
from optuna.visualization.matplotlib import plot_terminator_improvement


def objective(trial):
    X, y = load_wine(return_X_y=True)
    clf = LGBMClassifier(
        reg_alpha=trial.suggest_float("reg_alpha", 1e-8, 10.0, log=True),
        reg_lambda=trial.suggest_float("reg_lambda", 1e-8, 10.0, log=True),
        num_leaves=trial.suggest_int("num_leaves", 2, 256),
        colsample_bytree=trial.suggest_float("colsample_bytree", 0.4, 1.0),
        subsample=trial.suggest_float("subsample", 0.4, 1.0),
        subsample_freq=trial.suggest_int("subsample_freq", 1, 7),
        min_child_samples=trial.suggest_int("min_child_samples", 5, 100),
    )
    scores = cross_val_score(clf, X, y, cv=KFold(n_splits=5, shuffle=True))
    report_cross_validation_scores(trial, scores)
    return scores.mean()


study = optuna.create_study()
study.optimize(objective, n_trials=30)

plot_terminator_improvement(study, plot_error=True)
```
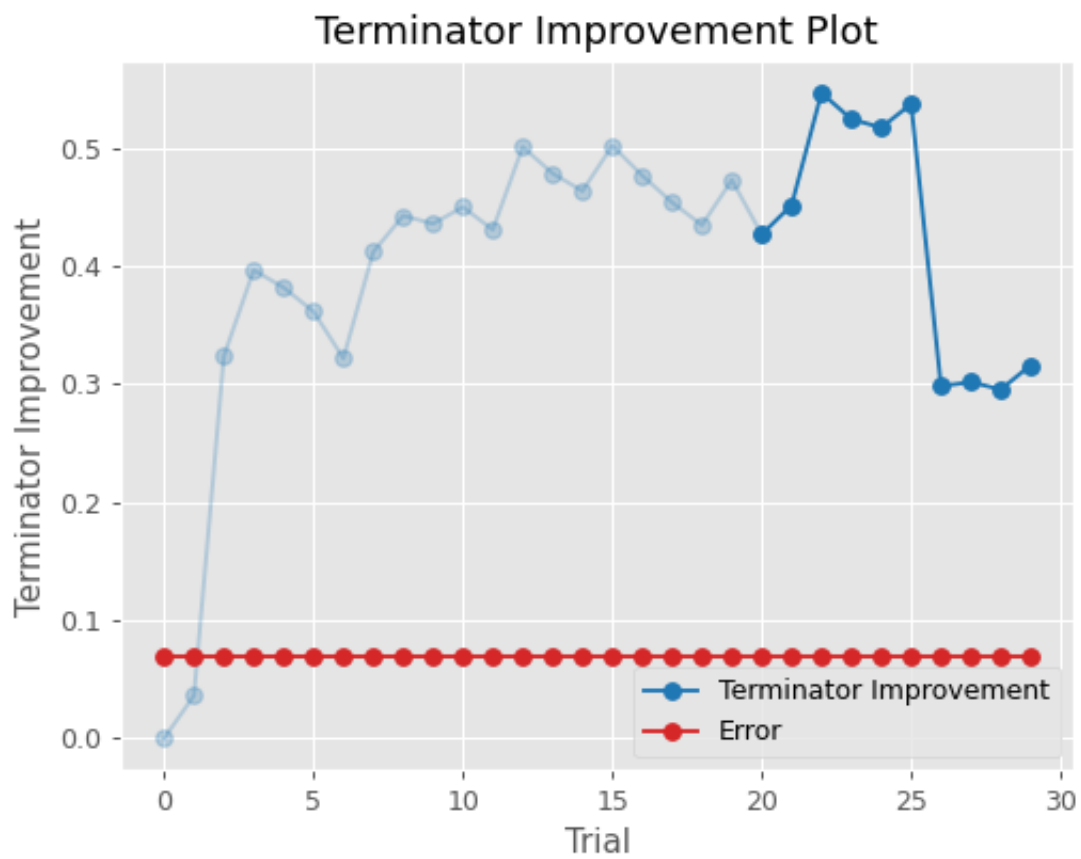
Parameters

- **study** (Study) – A *Study* object whose trials are plotted for their improvement.

- **plot_error** (*bool*) – A flag to show the error. If it is set to True, errors evaluated by error_evaluator are also plotted as line graph. Defaults to False.

- **improvement_evaluator** (BaseImprovementEvaluator | *None*) – An object that evaluates the improvement of the objective function. Default to *RegretBoundEvaluator*.

- **error_evaluator** (BaseErrorEvaluator | *None*) – An object that evaluates the error inherent in the objective function. Default to *CrossValidationErrorEvaluator*.

- **min_n_trials** (*int*) – The minimum number of trials before termination is considered. Terminator improvements for trials below this value are shown in a lighter color. Defaults to 20.

Returns
    A matplotlib.axes.Axes object.

Return type
    Axes

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

optuna.visualization.matplotlib.plot_timeline

optuna.visualization.matplotlib.**plot_timeline**(*study*)

Plot the timeline of a study.

See also:

Please refer to *optuna.visualization.plot_timeline()* for an example.

Example

The following code snippet shows how to plot the timeline of a study.

```python
import time

import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 1)
    time.sleep(x * 0.1)
    if x > 0.8:
        raise ValueError()
    if x > 0.4:
        raise optuna.TrialPruned()
    return x ** 2
```

```python
study = optuna.create_study(direction="minimize")
study.optimize(
    objective, n_trials=50, n_jobs=2, catch=(ValueError,)
)

optuna.visualization.matplotlib.plot_timeline(study)
```


Parameters
> **study** (Study) – A *Study* object whose trials are plotted with their lifetime.

Returns
> A `matplotlib.axes.Axes` object.

Return type
> *Axes*

Note: Added in v3.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.2.0.

---

optuna.visualization.matplotlib.is_available

optuna.visualization.matplotlib.**is_available**()

> Returns whether visualization with Matplotlib is available or not.

Note: `matplotlib` module depends on Matplotlib version 3.0.0 or higher. If a supported version of Matplotlib isn't installed in your environment, this function will return `False`. In such a case, please execute `$ pip install -U matplotlib>=3.0.0` to install Matplotlib.

> **Returns**
> > `True` if visualization with Matplotlib is available, `False` otherwise.
>
> **Return type**
> > bool

Note: Added in v2.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.2.0.

See also:

The visualization tutorial provides use-cases with examples.

7.4 FAQ

- *Can I use Optuna with X? (where X is your favorite ML library)*
- *How to define objective functions that have own arguments?*
- *Can I use Optuna without remote RDB servers?*
- *How can I save and resume studies?*
- *How to suppress log messages of Optuna?*
- *How to save machine learning models trained in objective functions?*
- *How can I obtain reproducible optimization results?*
- *How are exceptions from trials handled?*
- *How are NaNs returned by trials handled?*
- *What happens when I dynamically alter a search space?*
- *How can I use two GPUs for evaluating two trials simultaneously?*
- *How can I test my objective functions?*
- *How do I avoid running out of memory (OOM) when optimizing studies?*
- *How can I output a log only when the best value is updated?*
- *How do I suggest variables which represent the proportion, that is, are in accordance with Dirichlet distribution?*

7.4.1 Can I use Optuna with X? (where X is your favorite ML library)

Optuna is compatible with most ML libraries, and it's easy to use Optuna with those. Please refer to examples.

7.4.2 How to define objective functions that have own arguments?

There are two ways to realize it.

First, callable classes can be used for that purpose as follows:

```python
import optuna


class Objective:
    def __init__(self, min_x, max_x):
        # Hold this implementation specific arguments as the fields of the class.
        self.min_x = min_x
        self.max_x = max_x

    def __call__(self, trial):
        # Calculate an objective value by using the extra arguments.
        x = trial.suggest_float("x", self.min_x, self.max_x)
        return (x - 2) ** 2


# Execute an optimization by using an `Objective` instance.
study = optuna.create_study()
study.optimize(Objective(-100, 100), n_trials=100)
```

Second, you can use `lambda` or `functools.partial` for creating functions (closures) that hold extra arguments. Below is an example that uses `lambda`:

```python
import optuna


# Objective function that takes three arguments.
def objective(trial, min_x, max_x):
    x = trial.suggest_float("x", min_x, max_x)
```

(continues on next page)

```python
    return (x - 2) ** 2


# Extra arguments.
min_x = -100
max_x = 100

# Execute an optimization by using the above objective function wrapped by `lambda`.
study = optuna.create_study()
study.optimize(lambda trial: objective(trial, min_x, max_x), n_trials=100)
```

Please also refer to sklearn_additional_args.py example, which reuses the dataset instead of loading it in each trial execution.

7.4.3 Can I use Optuna without remote RDB servers?

Yes, it's possible.

In the simplest form, Optuna works with in-memory storage:

```python
study = optuna.create_study()
study.optimize(objective)
```

If you want to save and resume studies, it's handy to use SQLite as the local storage:

```python
study = optuna.create_study(study_name="foo_study", storage="sqlite:///example.db")
study.optimize(objective)  # The state of `study` will be persisted to the local SQLite
→file.
```

Please see rdb for more details.

7.4.4 How can I save and resume studies?

There are two ways of persisting studies, which depend if you are using in-memory storage (default) or remote databases (RDB). In-memory studies can be saved and loaded like usual Python objects using `pickle` or `joblib`. For example, using `joblib`:

```python
study = optuna.create_study()
joblib.dump(study, "study.pkl")
```

And to resume the study:

```python
study = joblib.load("study.pkl")
print("Best trial until now:")
print(" Value: ", study.best_trial.value)
print(" Params: ")
for key, value in study.best_trial.params.items():
    print(f"    {key}: {value}")
```

Note that Optuna does not support saving/reloading across different Optuna versions with `pickle`. To save/reload a study across different Optuna versions, please use RDBs and upgrade storage schema if necessary. If you are using RDBs, see rdb for more details.

7.4.5 How to suppress log messages of Optuna?

By default, Optuna shows log messages at the `optuna.logging.INFO` level. You can change logging levels by using `optuna.logging.set_verbosity()`.

For instance, you can stop showing each trial result as follows:

```
optuna.logging.set_verbosity(optuna.logging.WARNING)

study = optuna.create_study()
study.optimize(objective)
# Logs like '[I 2020-07-21 13:41:45,627] Trial 0 finished with value:...' are disabled.
```

Please refer to `optuna.logging` for further details.

7.4.6 How to save machine learning models trained in objective functions?

Optuna saves hyperparameter values with its corresponding objective value to storage, but it discards intermediate objects such as machine learning models and neural network weights. To save models or weights, please use features of the machine learning library you used.

We recommend saving `optuna.trial.Trial.number` with a model in order to identify its corresponding trial. For example, you can save SVM models trained in the objective function as follows:

```python
def objective(trial):
    svc_c = trial.suggest_float("svc_c", 1e-10, 1e10, log=True)
    clf = sklearn.svm.SVC(C=svc_c)
    clf.fit(X_train, y_train)

    # Save a trained model to a file.
    with open("{}.pickle".format(trial.number), "wb") as fout:
        pickle.dump(clf, fout)
    return 1.0 - accuracy_score(y_valid, clf.predict(X_valid))


study = optuna.create_study()
study.optimize(objective, n_trials=100)

# Load the best model.
with open("{}.pickle".format(study.best_trial.number), "rb") as fin:
    best_clf = pickle.load(fin)
print(accuracy_score(y_valid, best_clf.predict(X_valid)))
```

7.4.7 How can I obtain reproducible optimization results?

To make the parameters suggested by Optuna reproducible, you can specify a fixed random seed via `seed` argument of an instance of `samplers` as follows:

```python
sampler = TPESampler(seed=10)  # Make the sampler behave in a deterministic way.
study = optuna.create_study(sampler=sampler)
study.optimize(objective)
```

However, there are two caveats.

First, when optimizing a study in distributed or parallel mode, there is inherent non-determinism. Thus it is very difficult to reproduce the same results in such condition. We recommend executing optimization of a study sequentially if you would like to reproduce the result.

Second, if your objective function behaves in a non-deterministic way (i.e., it does not return the same value even if the same parameters were suggested), you cannot reproduce an optimization. To deal with this problem, please set an option (e.g., random seed) to make the behavior deterministic if your optimization target (e.g., an ML library) provides it.

7.4.8 How are exceptions from trials handled?

Trials that raise exceptions without catching them will be treated as failures, i.e. with the *FAIL* status.

By default, all exceptions except *TrialPruned* raised in objective functions are propagated to the caller of *optimize()*. In other words, studies are aborted when such exceptions are raised. It might be desirable to continue a study with the remaining trials. To do so, you can specify in *optimize()* which exception types to catch using the `catch` argument. Exceptions of these types are caught inside the study and will not propagate further.

You can find the failed trials in log messages.

```
[W 2018-12-07 16:38:36,889] Setting status of trial#0 as TrialState.FAIL because of \
the following error: ValueError('A sample error in objective.')
```

You can also find the failed trials by checking the trial states as follows:

```
study.trials_dataframe()
```

num-ber	state	value	...	param	system_attrs
0	Trial-State.FAIL		...	0	Setting status of trial#0 as TrialState.FAIL because of the following error: ValueError('A test error in objective.')
1	Trial-State.COMPLE	1269	...	1	

See also:

The `catch` argument in *optimize()*.

7.4.9 How are NaNs returned by trials handled?

Trials that return NaN (`float('nan')`) are treated as failures, but they will not abort studies.

Trials which return NaN are shown as follows:

```
[W 2018-12-07 16:41:59,000] Setting status of trial#2 as TrialState.FAIL because the \
objective function returned nan.
```

7.4.10 What happens when I dynamically alter a search space?

Since parameters search spaces are specified in each call to the suggestion API, e.g. *suggest_float()* and *suggest_int()*, it is possible to, in a single study, alter the range by sampling parameters from different search spaces in different trials. The behavior when altered is defined by each sampler individually.

Note: Discussion about the TPE sampler. https://github.com/optuna/optuna/issues/822

7.4.11 How can I use two GPUs for evaluating two trials simultaneously?

If your optimization target supports GPU (CUDA) acceleration and you want to specify which GPU is used in your script, `main.py`, the easiest way is to set `CUDA_VISIBLE_DEVICES` environment variable:

```
# On a terminal.
#
# Specify to use the first GPU, and run an optimization.
$ export CUDA_VISIBLE_DEVICES=0
$ python main.py

# On another terminal.
#
# Specify to use the second GPU, and run another optimization.
$ export CUDA_VISIBLE_DEVICES=1
$ python main.py
```

Please refer to CUDA C Programming Guide for further details.

7.4.12 How can I test my objective functions?

When you test objective functions, you may prefer fixed parameter values to sampled ones. In that case, you can use *FixedTrial*, which suggests fixed parameter values based on a given dictionary of parameters. For instance, you can input arbitrary values of x and y to the objective function $x + y$ as follows:

```python
def objective(trial):
    x = trial.suggest_float("x", -1.0, 1.0)
    y = trial.suggest_int("y", -5, 5)
    return x + y


objective(FixedTrial({"x": 1.0, "y": -1}))   # 0.0
objective(FixedTrial({"x": -1.0, "y": -4}))   # -5.0
```

Using *FixedTrial*, you can write unit tests as follows:

```python
# A test function of pytest
def test_objective():
    assert 1.0 == objective(FixedTrial({"x": 1.0, "y": 0}))
    assert -1.0 == objective(FixedTrial({"x": 0.0, "y": -1}))
    assert 0.0 == objective(FixedTrial({"x": -1.0, "y": 1}))
```

7.4.13 How do I avoid running out of memory (OOM) when optimizing studies?

If the memory footprint increases as you run more trials, try to periodically run the garbage collector. Specify gc_after_trial to `True` when calling *optimize()* or call `gc.collect()` inside a callback.

```python
def objective(trial):
    x = trial.suggest_float("x", -1.0, 1.0)
    y = trial.suggest_int("y", -5, 5)
    return x + y


study = optuna.create_study()
study.optimize(objective, n_trials=10, gc_after_trial=True)

# `gc_after_trial=True` is more or less identical to the following.
study.optimize(objective, n_trials=10, callbacks=[lambda study, trial: gc.collect()])
```

There is a performance trade-off for running the garbage collector, which could be non-negligible depending on how fast your objective function otherwise is. Therefore, gc_after_trial is `False` by default. Note that the above examples are similar to running the garbage collector inside the objective function, except for the fact that `gc.collect()` is called even when errors, including *TrialPruned* are raised.

Note: ChainerMNStudy does currently not provide gc_after_trial nor callbacks for optimize(). When using this class, you will have to call the garbage collector inside the objective function.

7.4.14 How can I output a log only when the best value is updated?

Here's how to replace the logging feature of optuna with your own logging callback function. The implemented callback can be passed to *optimize()*. Here's an example:

```python
import optuna


# Turn off optuna log notes.
optuna.logging.set_verbosity(optuna.logging.WARN)


def objective(trial):
    x = trial.suggest_float("x", 0, 1)
    return x ** 2


def logging_callback(study, frozen_trial):
    previous_best_value = study.user_attrs.get("previous_best_value", None)
    if previous_best_value != study.best_value:
        study.set_user_attr("previous_best_value", study.best_value)
        print(
            "Trial {} finished with best value: {} and parameters: {}. ".format(
                frozen_trial.number,
                frozen_trial.value,
                frozen_trial.params,
```

(continues on next page)

```
        )
    )


study = optuna.create_study()
study.optimize(objective, n_trials=100, callbacks=[logging_callback])
```

Note that this callback may show incorrect values when you try to optimize an objective function with `n_jobs!=1` (or other forms of distributed optimization) due to its reads and writes to storage that are prone to race conditions.

7.4.15 How do I suggest variables which represent the proportion, that is, are in accordance with Dirichlet distribution?

When you want to suggest n variables which represent the proportion, that is, $p[0], p[1], ..., p[n-1]$ which satisfy $0 \le p[k] \le 1$ for any k and $p[0] + p[1] + ... + p[n-1] = 1$, try the below. For example, these variables can be used as weights when interpolating the loss functions. These variables are in accordance with the flat Dirichlet distribution.

```python
import numpy as np
import matplotlib.pyplot as plt
import optuna


def objective(trial):
    n = 5
    x = []
    for i in range(n):
        x.append(- np.log(trial.suggest_float(f"x_{i}", 0, 1)))

    p = []
    for i in range(n):
        p.append(x[i] / sum(x))

    for i in range(n):
        trial.set_user_attr(f"p_{i}", p[i])

    return 0

study = optuna.create_study(sampler=optuna.samplers.RandomSampler())
study.optimize(objective, n_trials=1000)

n = 5
p = []
for i in range(n):
    p.append([trial.user_attrs[f"p_{i}"] for trial in study.trials])
axes = plt.subplots(n, n, figsize=(20, 20))[1]

for i in range(n):
    for j in range(n):
        axes[j][i].scatter(p[i], p[j], marker=".")
        axes[j][i].set_xlim(0, 1)
        axes[j][i].set_ylim(0, 1)
```

```
        axes[j][i].set_xlabel(f"p_{i}")
        axes[j][i].set_ylabel(f"p_{j}")

plt.savefig("sampled_ps.png")
```

This method is justified in the following way: First, if we apply the transformation $x = -\log(u)$ to the variable u sampled from the uniform distribution $Uni(0, 1)$ in the interval $[0, 1]$, the variable x will follow the exponential distribution $Exp(1)$ with scale parameter 1. Furthermore, for n variables $x[0], ..., x[n-1]$ that follow the exponential distribution of scale parameter 1 independently, normalizing them with $p[i] = x[i]/\sum_i x[i]$, the vector p follows the Dirichlet distribution $Dir(\alpha)$ of scale parameter $\alpha = (1, ..., 1)$. You can verify the transformation by calculating the elements of the Jacobian.

7.4.16 How can I optimize a model with some constraints?

When you want to optimize a model with constraints, you can use the following classes: *TPESampler*, *NSGAIISampler* or BoTorchSampler. The following example is a benchmark of Binh and Korn function, a multi-objective optimization, with constraints using *NSGAIISampler*. This one has two constraints $c_0 = (x-5)^2 + y^2 - 25 \leq 0$ and $c_1 = -(x-8)^2 - (y+3)^2 + 7.7 \leq 0$ and finds the optimal solution satisfying these constraints.

```python
import optuna


def objective(trial):
    # Binh and Korn function with constraints.
    x = trial.suggest_float("x", -15, 30)
    y = trial.suggest_float("y", -15, 30)

    # Constraints which are considered feasible if less than or equal to zero.
    # The feasible region is basically the intersection of a circle centered at (x=5,␣
→y=0)
    # and the complement to a circle centered at (x=8, y=-3).
    c0 = (x - 5) ** 2 + y ** 2 - 25
    c1 = -((x - 8) ** 2) - (y + 3) ** 2 + 7.7

    # Store the constraints as user attributes so that they can be restored after␣
→optimization.
    trial.set_user_attr("constraint", (c0, c1))

    v0 = 4 * x ** 2 + 4 * y ** 2
    v1 = (x - 5) ** 2 + (y - 5) ** 2

    return v0, v1


def constraints(trial):
    return trial.user_attrs["constraint"]


sampler = optuna.samplers.NSGAIISampler(constraints_func=constraints)
study = optuna.create_study(
    directions=["minimize", "minimize"],
```

```
    sampler=sampler,
)
study.optimize(objective, n_trials=32, timeout=600)

print("Number of finished trials: ", len(study.trials))

print("Pareto front:")

trials = sorted(study.best_trials, key=lambda t: t.values)

for trial in trials:
    print("  Trial#{}".format(trial.number))
    print(
        "    Values: Values={}, Constraint={}".format(
            trial.values, trial.user_attrs["constraint"][0]
        )
    )
    print("    Params: {}".format(trial.params))
```

If you are interested in an example for BoTorchSampler, please refer to this sample code.

There are two kinds of constrained optimizations, one with soft constraints and the other with hard constraints. Soft constraints do not have to be satisfied, but an objective function is penalized if they are unsatisfied. On the other hand, hard constraints must be satisfied.

Optuna is adopting the soft one and **DOES NOT** support the hard one. In other words, Optuna **DOES NOT** have built-in samplers for the hard constraints.

7.4.17 How can I parallelize optimization?

The variations of parallelization are in the following three cases.

1. Multi-threading parallelization with single node

2. Multi-processing parallelization with single node

3. Multi-processing parallelization with multiple nodes

1. Multi-threading parallelization with a single node

Parallelization can be achieved by setting the argument n_jobs in optuna.study.Study.optimize(). However, the python code will not be faster due to GIL because optuna.study.Study.optimize() with n_jobs!=1 uses multi-threading.

While optimizing, it will be faster in limited situations, such as waiting for other server requests or C/C++ processing with numpy, etc., but it will not be faster in other cases.

For more information about 1., see APIReference.

2. Multi-processing parallelization with single node

This can be achieved by using `JournalFileStorage` or client/server RDBs (such as PostgreSQL and MySQL).

For more information about 2., see TutorialEasyParallelization.

3. Multi-processing parallelization with multiple nodes

This can be achieved by using client/server RDBs (such as PostgreSQL and MySQL). However, if you are in the environment where you can not install a client/server RDB, you can not run multi-processing parallelization with multiple nodes.

For more information about 3., see TutorialEasyParallelization.

7.4.18 How can I solve the error that occurs when performing parallel optimization with SQLite3?

We would never recommend SQLite3 for parallel optimization in the following reasons.

- To concurrently evaluate trials enqueued by `enqueue_trial()`, `RDBStorage` uses *SELECT . . . FOR UPDATE* syntax, which is unsupported in SQLite3.
- As described in the SQLAlchemy's documentation, SQLite3 (and pysqlite driver) does not support a high level of concurrency. You may get a "database is locked" error, which occurs when one thread or process has an exclusive lock on a database connection (in reality a file handle) and another thread times out waiting for the lock to be released. You can increase the default timeout value like *optuna.storages.RDBStorage("sqlite:///example.db", engine_kwargs={"connect_args": {"timeout": 20.0}})* though.
- For distributed optimization via NFS, SQLite3 does not work as described at FAQ section of sqlite.org.

If you want to use a file-based Optuna storage for these scenarios, please consider using `JournalFileStorage` instead.

```
import optuna
from optuna.storages import JournalStorage, JournalFileStorage

storage = JournalStorage(JournalFileStorage("optuna-journal.log"))
study = optuna.create_study(storage=storage)
...
```

See the Medium blog post for details.

7.4.19 Can I monitor trials and make them failed automatically when they are killed unexpectedly?

Note: Heartbeat mechanism is experimental. API would change in the future.

A process running a trial could be killed unexpectedly, typically by a job scheduler in a cluster environment. If trials are killed unexpectedly, they will be left on the storage with their states *RUNNING* until we remove them or update their state manually. For such a case, Optuna supports monitoring trials using heartbeat mechanism. Using heartbeat, if a process running a trial is killed unexpectedly, Optuna will automatically change the state of the trial that was running on that process to *FAIL* from *RUNNING*.

```
import optuna

def objective(trial):
    (Very time-consuming computation)

# Recording heartbeats every 60 seconds.
# Other processes' trials where more than 120 seconds have passed
# since the last heartbeat was recorded will be automatically failed.
storage = optuna.storages.RDBStorage(url="sqlite:///:memory:", heartbeat_interval=60,
→grace_period=120)
study = optuna.create_study(storage=storage)
study.optimize(objective, n_trials=100)
```

Note: The heartbeat is supposed to be used with *optimize()*. If you use *ask()* and *tell()*, please change the state of the killed trials by calling *tell()* explicitly.

You can also execute a callback function to process the failed trial. Optuna provides a callback to retry failed trials as *RetryFailedTrialCallback*. Note that a callback is invoked at a beginning of each trial, which means *RetryFailedTrialCallback* will retry failed trials when a new trial starts to evaluate.

```
import optuna
from optuna.storages import RetryFailedTrialCallback

storage = optuna.storages.RDBStorage(
    url="sqlite:///:memory:",
    heartbeat_interval=60,
    grace_period=120,
    failed_trial_callback=RetryFailedTrialCallback(max_retry=3),
)

study = optuna.create_study(storage=storage)
```

7.4.20 How can I deal with permutation as a parameter?

Although it is not straightforward to deal with combinatorial search spaces like permutations with existing API, there exists a convenient technique for handling them. It involves re-parametrization of permutation search space of n items as an independent n-dimensional integer search space. This technique is based on the concept of Lehmer code.

A Lehmer code of a sequence is the sequence of integers in the same size, whose i-th entry denotes how many inversions the i-th entry of the permutation has after itself. In other words, the i-th entry of the Lehmer code represents the number of entries that are located after and are smaller than the i-th entry of the original sequence. For instance, the Lehmer code of the permutation $(3, 1, 4, 2, 0)$ is $(3, 1, 2, 1, 0)$.

Not only does the Lehmer code provide a unique encoding of permutations into an integer space, but it also has some desirable properties. For example, the sum of Lehmer code entries is equal to the minimum number of adjacent transpositions necessary to transform the corresponding permutation into the identity permutation. Additionally, the lexicographical order of the encodings of two permutations is the same as that of the original sequence. Therefore, Lehmer code preserves "closeness" among permutations in some sense, which is important for the optimization algorithm. An Optuna implementation example to solve Euclid TSP is as follows:

```python
import numpy as np

import optuna


def decode(lehmer_code: list[int]) -> list[int]:
    """Decode Lehmer code to permutation.

    This function decodes Lehmer code represented as a list of integers to a permutation.
    """
    all_indices = list(range(n))
    output = []
    for k in lehmer_code:
        value = all_indices[k]
        output.append(value)
        all_indices.remove(value)
    return output


# Euclidean coordinates of cities for TSP.
city_coordinates = np.array(
    [[0.0, 0.0], [1.0, 0.0], [0.0, 1.0], [1.0, 1.0], [2.0, 2.0], [-1.0, -1.0]]
)
n = len(city_coordinates)


def objective(trial: optuna.Trial) -> float:
    # Suggest a permutation in the Lehmer code representation.
    lehmer_code = [trial.suggest_int(f"x{i}", 0, n - i - 1) for i in range(n)]
    permutation = decode(lehmer_code)

    # Calculate the total distance of the suggested path.
    total_distance = 0.0
    for i in range(n):
        total_distance += np.linalg.norm(
            city_coordinates[permutation[i]] - city_coordinates[np.roll(permutation,
→1)[i]]
        )
    return total_distance


study = optuna.create_study()
study.optimize(objective, n_trials=10)
lehmer_code = study.best_params.values()
print(decode(lehmer_code))
```

7.4.21 How can I ignore duplicated samples?

Optuna may sometimes suggest parameters evaluated in the past and if you would like to avoid this problem, you can try out the following workaround:

```python
import optuna
from optuna.trial import TrialState


def objective(trial):
    # Sample parameters.
    x = trial.suggest_int("x", -5, 5)
    y = trial.suggest_int("y", -5, 5)
    # Fetch all the trials to consider.
    # In this example, we use only completed trials, but users can specify other states
    # such as TrialState.PRUNED and TrialState.FAIL.
    states_to_consider = (TrialState.COMPLETE,)
    trials_to_consider = trial.study.get_trials(deepcopy=False, states=states_to_
→consider)
    # Check whether we already evaluated the sampled `(x, y)`.
    for t in reversed(trials_to_consider):
        if trial.params == t.params:
            # Use the existing value as trial duplicated the parameters.
            return t.value

    # Compute the objective function if the parameters are not duplicated.
    # We use the 2D sphere function in this example.
    return x ** 2 + y ** 2


study = optuna.create_study()
study.optimize(objective, n_trials=100)
```

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

M

N

O

P

U

V

W