Optuna Documentation

Release 3.2.0.dev0

Optuna Contributors.

Apr 12, 2023

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.

CHAPTER ONE

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.

CHAPTER TWO

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*.

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.
```

CHAPTER THREE

COMMUNICATION

- GitHub Discussions for questions.
- GitHub Issues for bug reports and feature requests.

CHAPTER

FOUR

CONTRIBUTION

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

CHAPTER

FIVE

LICENSE

MIT License (see LICENSE).

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

CHAPTER SIX

REFERENCE

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

6.1 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
```

6.2 Tutorial

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

6.2.1 Key Features

Showcases Optuna's Key Features.

6.2.2 Recipes

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

Key Features

Showcases Optuna's Key Features.

1. Lightweight, versatile, and platform agnostic architecture

Optuna is entirely written in Python and has few dependencies. This means that we can quickly move to the real example once you get interested in Optuna.

Quadratic Function Example

Usually, Optuna is used to optimize hyperparameters, but as an example, let's optimize a simple quadratic function: $(x-2)^2$.

First of all, import `optuna`.

```
import optuna
```

In optuna, conventionally functions to be optimized are named *objective*.

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

This function returns the value of $(x-2)^2$. Our goal is to find the value of `x` that minimizes the output of the `objective` function. This is the "optimization." During the optimization, Optuna repeatedly calls and evaluates the objective function with different values of `x`.

A `Trial` object corresponds to a single execution of the objective function and is internally instantiated upon each invocation of the function.

The *suggest* APIs (for example, `suggest_float()`) are called inside the objective function to obtain parameters for a trial. `suggest_float()` selects parameters uniformly within the range provided. In our example, from -10 to 10.

To start the optimization, we create a study object and pass the objective function to method `optimize()` as follows.

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

You can get the best parameter as follows.

```
best_params = study.best_params
found_x = best_params["x"]
print("Found x: {}, (x - 2)^2: {}".format(found_x, (found_x - 2) ** 2))
```

```
Found x: 2.010821853260173, (x - 2)^2: 0.00011711250798472O3
```

We can see that the `x` value found by Optuna is close to the optimal value of `2`.

Note: When used to search for hyperparameters in machine learning, usually the objective function would return the loss or accuracy of the model.

Study Object

Let us clarify the terminology in Optuna as follows:

- **Trial**: A single call of the objective function
- **Study**: An optimization session, which is a set of trials
- **Parameter**: A variable whose value is to be optimized, such as `x` in the above example

In Optuna, we use the study object to manage optimization. Method *create_study()* returns a study object. A study object has useful properties for analyzing the optimization outcome.

To get the dictionary of parameter name and parameter values:

```
study.best_params
```

```
{'x': 2.010821853260173}
```

To get the best observed value of the objective function:

```
study.best_value
```

```
0.0001171125079847203
```

To get the best trial:

```
study.best_trial
```

```
FrozenTrial(number=54, state=TrialState.COMPLETE, values=[0.0001171125079847203],␣
↪datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 14, 840715), datetime_
↪complete=datetime.datetime(2023, 4, 12, 3, 5, 14, 846232), params={'x': 2.
↪010821853260173}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=54, value=None)
```

To get all trials:

```
study.trials
for trial in study.trials[:2]:  # Show first two trials
    print(trial)
```

```
FrozenTrial(number=0, state=TrialState.COMPLETE, values=[0.5875474922163815], datetime_
↪start=datetime.datetime(2023, 4, 12, 3, 5, 14, 584519), datetime_complete=datetime.
↪datetime(2023, 4, 12, 3, 5, 14, 584979), params={'x': 2.766516465717718}, user_attrs={}
↪, system_attrs={}, intermediate_values={}, distributions={'x':␣
↪FloatDistribution(high=10.0, log=False, low=-10.0, step=None)}, trial_id=0, value=None)
```

(continues on next page)

(continued from previous page)

```
FrozenTrial(number=1, state=TrialState.COMPLETE, values=[0.0260027249504532], datetime_
↪start=datetime.datetime(2023, 4, 12, 3, 5, 14, 585302), datetime_complete=datetime.
↪datetime(2023, 4, 12, 3, 5, 14, 585632), params={'x': 1.8387463955427563}, user_attrs=
↪{}, system_attrs={}, intermediate_values={}, distributions={'x':␣
↪FloatDistribution(high=10.0, log=False, low=-10.0, step=None)}, trial_id=1, value=None)
```

To get the number of trials:

```
len(study.trials)
```

```
100
```

By executing *optimize()* again, we can continue the optimization.

```
study.optimize(objective, n_trials=100)
```

To get the updated number of trials:

```
len(study.trials)
```

```
200
```

As the objective function is so easy that the last 100 trials don't improve the result. However, we can check the result again:

```
best_params = study.best_params
found_x = best_params["x"]
print("Found x: {}, (x - 2)^2: {}".format(found_x, (found_x - 2) ** 2))
```

```
Found x: 2.001475064245528, (x - 2)^2: 2.175814528435453e-06
```

Total running time of the script: (0 minutes 1.211 seconds)

2. Pythonic Search Space

For hyperparameter sampling, Optuna provides the following features:

- *optuna.trial.Trial.suggest_categorical()* for categorical parameters
- *optuna.trial.Trial.suggest_int()* for integer parameters
- *optuna.trial.Trial.suggest_float()* for floating point parameters

With optional arguments of `step` and `log`, we can discretize or take the logarithm of integer and floating point parameters.

```
import optuna


def objective(trial):
    # Categorical parameter
    optimizer = trial.suggest_categorical("optimizer", ["MomentumSGD", "Adam"])
```

(continues on next page)

(continued from previous page)

```python
    # Integer parameter
    num_layers = trial.suggest_int("num_layers", 1, 3)

    # Integer parameter (log)
    num_channels = trial.suggest_int("num_channels", 32, 512, log=True)

    # Integer parameter (discretized)
    num_units = trial.suggest_int("num_units", 10, 100, step=5)

    # Floating point parameter
    dropout_rate = trial.suggest_float("dropout_rate", 0.0, 1.0)

    # Floating point parameter (log)
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-2, log=True)

    # Floating point parameter (discretized)
    drop_path_rate = trial.suggest_float("drop_path_rate", 0.0, 1.0, step=0.1)
```

Defining Parameter Spaces

In Optuna, we define search spaces using familiar Python syntax including conditionals and loops.

Also, you can use branches or loops depending on the parameter values.

For more various use, see examples.

- Branches:

```python
import sklearn.ensemble
import sklearn.svm


def objective(trial):
    classifier_name = trial.suggest_categorical("classifier", ["SVC", "RandomForest"])
    if classifier_name == "SVC":
        svc_c = trial.suggest_float("svc_c", 1e-10, 1e10, log=True)
        classifier_obj = sklearn.svm.SVC(C=svc_c)
    else:
        rf_max_depth = trial.suggest_int("rf_max_depth", 2, 32, log=True)
        classifier_obj = sklearn.ensemble.RandomForestClassifier(max_depth=rf_max_depth)
```

- Loops:

```python
import torch
import torch.nn as nn


def create_model(trial, in_size):
    n_layers = trial.suggest_int("n_layers", 1, 3)

    layers = []
```

(continues on next page)

(continued from previous page)

```
    for i in range(n_layers):
        n_units = trial.suggest_int("n_units_l{}".format(i), 4, 128, log=True)
        layers.append(nn.Linear(in_size, n_units))
        layers.append(nn.ReLU())
        in_size = n_units
    layers.append(nn.Linear(in_size, 10))

    return nn.Sequential(*layers)
```

Note on the Number of Parameters

The difficulty of optimization increases roughly exponentially with regard to the number of parameters. That is, the number of necessary trials increases exponentially when you increase the number of parameters, so it is recommended to not add unimportant parameters.

Total running time of the script: (0 minutes 0.001 seconds)

3. Efficient Optimization Algorithms

Optuna enables efficient hyperparameter optimization by adopting state-of-the-art algorithms for sampling hyperparameters and pruning efficiently unpromising trials.

Sampling Algorithms

Samplers basically continually narrow down the search space using the records of suggested parameter values and evaluated objective values, leading to an optimal search space which giving off parameters leading to better objective values. More detailed explanation of how samplers suggest parameters is in *BaseSampler*.

Optuna provides the following sampling algorithms:

- Grid Search implemented in *GridSampler*
- Random Search implemented in *RandomSampler*
- Tree-structured Parzen Estimator algorithm implemented in *TPESampler*
- CMA-ES based algorithm implemented in *CmaEsSampler*
- Algorithm to enable partial fixed parameters implemented in *PartialFixedSampler*
- Nondominated Sorting Genetic Algorithm II implemented in *NSGAIISampler*
- A Quasi Monte Carlo sampling algorithm implemented in *QMCSampler*

The default sampler is *TPESampler*.

Switching Samplers

```
import optuna
```

By default, Optuna uses *TPESampler* as follows.

```
study = optuna.create_study()
print(f"Sampler is {study.sampler.__class__.__name__}")
```

```
Sampler is TPESampler
```

If you want to use different samplers for example *RandomSampler* and *CmaEsSampler*,

```
study = optuna.create_study(sampler=optuna.samplers.RandomSampler())
print(f"Sampler is {study.sampler.__class__.__name__}")

study = optuna.create_study(sampler=optuna.samplers.CmaEsSampler())
print(f"Sampler is {study.sampler.__class__.__name__}")
```

```
Sampler is RandomSampler
Sampler is CmaEsSampler
```

Pruning Algorithms

`Pruners` automatically stop unpromising trials at the early stages of the training (a.k.a., automated early-stopping).

Optuna provides the following pruning algorithms:

- Median pruning algorithm implemented in *MedianPruner*
- Non-pruning algorithm implemented in *NopPruner*
- Algorithm to operate pruner with tolerance implemented in *PatientPruner*
- Algorithm to prune specified percentile of trials implemented in *PercentilePruner*
- Asynchronous Successive Halving algorithm implemented in *SuccessiveHalvingPruner*
- Hyperband algorithm implemented in *HyperbandPruner*
- Threshold pruning algorithm implemented in *ThresholdPruner*

We use *MedianPruner* in most examples, though basically it is outperformed by *SuccessiveHalvingPruner* and *HyperbandPruner* as in this benchmark result.

Activating Pruners

To turn on the pruning feature, you need to call *report()* and *should_prune()* after each step of the iterative training. *report()* periodically monitors the intermediate objective values. *should_prune()* decides termination of the trial that does not meet a predefined condition.

We would recommend using integration modules for major machine learning frameworks. Exclusive list is *integration* and usecases are available in ~optuna/examples.

```
import logging
import sys

import sklearn.datasets
import sklearn.linear_model
import sklearn.model_selection


def objective(trial):
    iris = sklearn.datasets.load_iris()
    classes = list(set(iris.target))
    train_x, valid_x, train_y, valid_y = sklearn.model_selection.train_test_split(
        iris.data, iris.target, test_size=0.25, random_state=0
    )

    alpha = trial.suggest_float("alpha", 1e-5, 1e-1, log=True)
    clf = sklearn.linear_model.SGDClassifier(alpha=alpha)

    for step in range(100):
        clf.partial_fit(train_x, train_y, classes=classes)

        # Report intermediate objective value.
        intermediate_value = 1.0 - clf.score(valid_x, valid_y)
        trial.report(intermediate_value, step)

        # Handle pruning based on the intermediate value.
        if trial.should_prune():
            raise optuna.TrialPruned()

    return 1.0 - clf.score(valid_x, valid_y)
```

Set up the median stopping rule as the pruning condition.

```
# Add stream handler of stdout to show the messages
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))
study = optuna.create_study(pruner=optuna.pruners.MedianPruner())
study.optimize(objective, n_trials=20)
```

```
A new study created in memory with name: no-name-7bd57034-87a8-4a27-a576-d77e5f33c1d3
Trial 0 finished with value: 0.02631578947368418 and parameters: {'alpha': 0.
↪002876233166428457}. Best is trial 0 with value: 0.02631578947368418.
Trial 1 finished with value: 0.0 and parameters: {'alpha': 0.004732646023084551}. Best␣
↪is trial 1 with value: 0.0.
Trial 2 finished with value: 0.13157894736842102 and parameters: {'alpha': 0.
↪023726241970809043}. Best is trial 1 with value: 0.0.
Trial 3 finished with value: 0.10526315789473684 and parameters: {'alpha': 0.
↪00011412028704278689}. Best is trial 1 with value: 0.0.
Trial 4 finished with value: 0.3157894736842105 and parameters: {'alpha': 0.
↪09693327954213213}. Best is trial 1 with value: 0.0.
Trial 5 finished with value: 0.07894736842105265 and parameters: {'alpha': 0.
↪01825200724176781}. Best is trial 1 with value: 0.0.
Trial 6 pruned.
```

(continues on next page)

(continued from previous page)

```
Trial 7 pruned.
Trial 8 pruned.
Trial 9 pruned.
Trial 10 pruned.
Trial 11 pruned.
Trial 12 pruned.
Trial 13 pruned.
Trial 14 pruned.
Trial 15 pruned.
Trial 16 pruned.
Trial 17 finished with value: 0.07894736842105265 and parameters: {'alpha': 0.
↪006205012996796397}. Best is trial 1 with value: 0.0.
Trial 18 finished with value: 0.3157894736842105 and parameters: {'alpha': 0.
↪02421735738244837}. Best is trial 1 with value: 0.0.
Trial 19 pruned.
```

As you can see, several trials were pruned (stopped) before they finished all of the iterations. The format of message is `"Trial <Trial Number> pruned."`.

Which Sampler and Pruner Should be Used?

From the benchmark results which are available at optuna/optuna - wiki "Benchmarks with Kurobako", at least for not deep learning tasks, we would say that

- For *RandomSampler*, *MedianPruner* is the best.
- For *TPESampler*, *HyperbandPruner* is the best.

However, note that the benchmark is not deep learning. For deep learning tasks, consult the below table. This table is from the Ozaki et al., Hyperparameter Optimization Methods: Overview and Characteristics, in IEICE Trans, Vol.J103-D No.9 pp.615-631, 2020 paper, which is written in Japanese.

Parallel Compute Resource	Categorical/Conditional Hyperparameters	Recommended Algorithms
Limited	No	TPE. GP-EI if search space is low-dimensional and continuous.
	Yes	TPE. GP-EI if search space is low-dimensional and continuous
Sufficient	No	CMA-ES, Random Search
	Yes	Random Search or Genetic Algorithm

Integration Modules for Pruning

To implement pruning mechanism in much simpler forms, Optuna provides integration modules for the following libraries.

For the complete list of Optuna's integration modules, see *integration*.

For example, *XGBoostPruningCallback* introduces pruning without directly changing the logic of training iteration. (See also example for the entire script.)

```
pruning_callback = optuna.integration.XGBoostPruningCallback(trial, 'validation-error')
bst = xgb.train(param, dtrain, evals=[(dvalid, 'validation')], callbacks=[pruning_
↪callback])
```

Total running time of the script: (0 minutes 2.078 seconds)

4. Easy Parallelization

It's straightforward to parallelize *optuna.study.Study.optimize()*.

If you want to manually execute Optuna optimization:

1. start an RDB server (this example uses MySQL)
2. create a study with `--storage` argument
3. share the study among multiple nodes and processes

Of course, you can use Kubernetes as in the kubernetes examples.

To just see how parallel optimization works in Optuna, check the below video.

Create a Study

You can create a study using `optuna create-study` command. Alternatively, in Python script you can use *optuna.create_study()*.

```
$ mysql -u root -e "CREATE DATABASE IF NOT EXISTS example"
$ optuna create-study --study-name "distributed-example" --storage "mysql://
↪root@localhost/example"
[I 2020-07-21 13:43:39,642] A new study created with name: distributed-example
```

Then, write an optimization script. Let's assume that `foo.py` contains the following code.

```
import optuna


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


if __name__ == "__main__":
    study = optuna.load_study(
        study_name="distributed-example", storage="mysql://root@localhost/example"
    )
    study.optimize(objective, n_trials=100)
```

Share the Study among Multiple Nodes and Processes

Finally, run the shared study from multiple processes. For example, run `Process 1` in a terminal, and do `Process 2` in another one. They get parameter suggestions based on shared trials' history.

Process 1:

```
$ python foo.py
[I 2020-07-21 13:45:02,973] Trial 0 finished with value: 45.35553104173011 and␣
↪parameters: {'x': 8.73465151598285}. Best is trial 0 with value: 45.35553104173011.
[I 2020-07-21 13:45:04,013] Trial 2 finished with value: 4.6002397305938905 and␣
↪parameters: {'x': 4.144816945707463}. Best is trial 1 with value: 0.028194513284051464.
...
```

Process 2 (the same command as process 1):

```
$ python foo.py
[I 2020-07-21 13:45:03,748] Trial 1 finished with value: 0.028194513284051464 and␣
↪parameters: {'x': 1.8320877810162361}. Best is trial 1 with value: 0.
↪028194513284051464.
[I 2020-07-21 13:45:05,783] Trial 3 finished with value: 24.45966755098074 and␣
↪parameters: {'x': 6.945671597566982}. Best is trial 1 with value: 0.028194513284051464.
...
```

Note: `n_trials` is the number of trials each process will run, not the total number of trials across all processes. For example, the script given above runs 100 trials for each process, 100 trials * 2 processes = 200 trials. *`optuna.study.MaxTrialsCallback`* can ensure how many times trials will be performed across all processes.

Note: We do not recommend SQLite for distributed optimizations at scale because it may cause deadlocks and serious performance issues. Please consider to use another database engine like PostgreSQL or MySQL.

Note: Please avoid putting the SQLite database on NFS when running distributed optimizations. See also: https://www.sqlite.org/faq.html#q5

Total running time of the script: (0 minutes 0.000 seconds)

5. Quick Visualization for Hyperparameter Optimization Analysis

Optuna provides various visualization features in *`optuna.visualization`* to analyze optimization results visually.

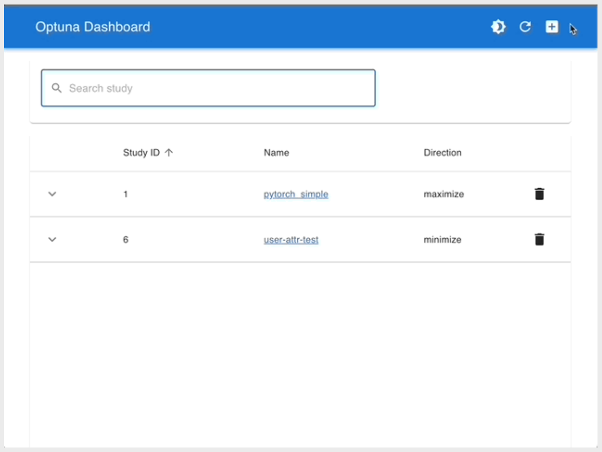
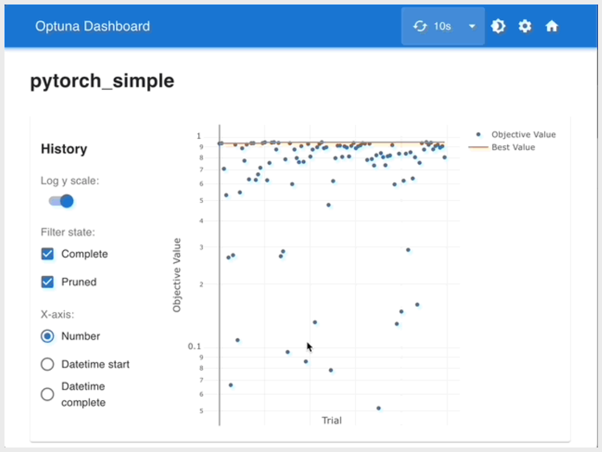
This tutorial walks you through this module by visualizing the history of lightgbm model for breast cancer dataset.

For visualizing multi-objective optimization (i.e., the usage of *`optuna.visualization.plot_pareto_front()`*), please refer to the tutorial of *Multi-objective Optimization with Optuna*.

Note: By using Optuna Dashboard, you can also check the optimization history, hyperparameter importances, hyperparameter relationships, etc. in graphs and tables. Please make your study persistent using *RDB backend* and execute following commands to run Optuna Dashboard.

```
$ pip install optuna-dashboard
$ optuna-dashboard sqlite:///example-study.db
```

Please check out the GitHub repository for more details.

Manage Studies	Visualize with Interactive Graphs
	

	Study ID ↑	Name	Direction
⌄	1	pytorch_simple	maximize
⌄	6	user-attr-test	minimize

```python
import lightgbm as lgb
import numpy as np
import sklearn.datasets
import sklearn.metrics
from sklearn.model_selection import train_test_split

import optuna

# You can use Matplotlib instead of Plotly for visualization by simply replacing `optuna.
↪visualization` with
# `optuna.visualization.matplotlib` in the following examples.
from optuna.visualization import plot_contour
from optuna.visualization import plot_edf
from optuna.visualization import plot_intermediate_values
from optuna.visualization import plot_optimization_history
from optuna.visualization import plot_parallel_coordinate
from optuna.visualization import plot_param_importances
from optuna.visualization import plot_slice

SEED = 42

np.random.seed(SEED)
```

Define the objective function.

```python
def objective(trial):
    data, target = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, valid_x, train_y, valid_y = train_test_split(data, target, test_size=0.25)
    dtrain = lgb.Dataset(train_x, label=train_y)
```

(continues on next page)

(continued from previous page)

```
    dvalid = lgb.Dataset(valid_x, label=valid_y)

    param = {
        "objective": "binary",
        "metric": "auc",
        "verbosity": -1,
        "boosting_type": "gbdt",
        "bagging_fraction": trial.suggest_float("bagging_fraction", 0.4, 1.0),
        "bagging_freq": trial.suggest_int("bagging_freq", 1, 7),
        "min_child_samples": trial.suggest_int("min_child_samples", 5, 100),
    }

    # Add a callback for pruning.
    pruning_callback = optuna.integration.LightGBMPruningCallback(trial, "auc")
    gbm = lgb.train(param, dtrain, valid_sets=[dvalid], callbacks=[pruning_callback])

    preds = gbm.predict(valid_x)
    pred_labels = np.rint(preds)
    accuracy = sklearn.metrics.accuracy_score(valid_y, pred_labels)
    return accuracy
```

```
study = optuna.create_study(
    direction="maximize",
    sampler=optuna.samplers.TPESampler(seed=SEED),
    pruner=optuna.pruners.MedianPruner(n_warmup_steps=10),
)
study.optimize(objective, n_trials=100, timeout=600)
```

Plot functions

Visualize the optimization history. See *plot_optimization_history()* for the details.

```
plot_optimization_history(study)
```

Visualize the learning curves of the trials. See *plot_intermediate_values()* for the details.

```
plot_intermediate_values(study)
```

Visualize high-dimensional parameter relationships. See *plot_parallel_coordinate()* for the details.

```
plot_parallel_coordinate(study)
```

Select parameters to visualize.

```
plot_parallel_coordinate(study, params=["bagging_freq", "bagging_fraction"])
```

Visualize hyperparameter relationships. See *plot_contour()* for the details.

```
plot_contour(study)
```

Select parameters to visualize.

```
plot_contour(study, params=["bagging_freq", "bagging_fraction"])
```

Visualize individual hyperparameters as slice plot. See *plot_slice()* for the details.

```
plot_slice(study)
```

Select parameters to visualize.

```
plot_slice(study, params=["bagging_freq", "bagging_fraction"])
```

Visualize parameter importances. See *plot_param_importances()* for the details.

```
plot_param_importances(study)
```

Learn which hyperparameters are affecting the trial duration with hyperparameter importance.

```
optuna.visualization.plot_param_importances(
    study, target=lambda t: t.duration.total_seconds(), target_name="duration"
)
```

Visualize empirical distribution function. See *plot_edf()* for the details.

```
plot_edf(study)
```

Total running time of the script: (0 minutes 5.676 seconds)

Recipes

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

Saving/Resuming Study with RDB Backend

An RDB backend enables persistent experiments (i.e., to save and resume a study) as well as access to history of studies. In addition, we can run multi-node optimization tasks with this feature, which is described in *4. Easy Parallelization*.

In this section, let's try simple examples running on a local environment with SQLite DB.

Note: You can also utilize other RDB backends, e.g., PostgreSQL or MySQL, by setting the storage argument to the DB's URL. Please refer to SQLAlchemy's document for how to set up the URL.

New Study

We can create a persistent study by calling *create_study()* function as follows. An SQLite file `example.db` is automatically initialized with a new study record.

```
import logging
import sys

import optuna
```

(continues on next page)

(continued from previous page)

```
# Add stream handler of stdout to show the messages
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))
study_name = "example-study"  # Unique identifier of the study.
storage_name = "sqlite:///{}.db".format(study_name)
study = optuna.create_study(study_name=study_name, storage=storage_name)
```

```
A new study created in RDB with name: example-study
```

To run a study, call *optimize()* method passing an objective function.

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


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

```
Trial 0 finished with value: 103.12956901377896 and parameters: {'x': -8.155272965990573}
↪. Best is trial 0 with value: 103.12956901377896.
Trial 1 finished with value: 51.05476378686051 and parameters: {'x': -5.145261631799112}.
↪ Best is trial 1 with value: 51.05476378686051.
Trial 2 finished with value: 42.103769322607924 and parameters: {'x': -4.488741736470017}
↪. Best is trial 2 with value: 42.103769322607924.
```

Resume Study

To resume a study, instantiate a *Study* object passing the study name `example-study` and the DB URL `sqlite:///example-study.db`.

```
study = optuna.create_study(study_name=study_name, storage=storage_name, load_if_
↪exists=True)
study.optimize(objective, n_trials=3)
```

```
Using an existing study with name 'example-study' instead of creating a new one.
Trial 3 finished with value: 58.836769787129654 and parameters: {'x': -5.670513006776643}
↪. Best is trial 2 with value: 42.103769322607924.
Trial 4 finished with value: 120.92088444119118 and parameters: {'x': -8.996403250208278}
↪. Best is trial 2 with value: 42.103769322607924.
Trial 5 finished with value: 69.97616173137462 and parameters: {'x': -6.365175535000722}.
↪ Best is trial 2 with value: 42.103769322607924.
```

Note that the storage doesn't store the state of the instance of *samplers*. When we resume a study with a sampler whose `seed` argument is specified for reproducibility, you need to restore the sampler with using `pickle` as follows:

```
import pickle

# Save the sampler with pickle to be loaded later.
with open("sampler.pkl", "wb") as fout:
    pickle.dump(study.sampler, fout)
```

(continues on next page)

(continued from previous page)

```
restored_sampler = pickle.load(open("sampler.pkl", "rb"))
study = optuna.create_study(
    study_name=study_name, storage=storage_name, load_if_exists=True, sampler=restored_
↪sampler
)
study.optimize(objective, n_trials=3)
```

Experimental History

We can access histories of studies and trials via the *Study* class. For example, we can get all trials of `example-study` as:

```
study = optuna.create_study(study_name=study_name, storage=storage_name, load_if_
↪exists=True)
df = study.trials_dataframe(attrs=("number", "value", "params", "state"))
```

```
Using an existing study with name 'example-study' instead of creating a new one.
```

The method *trials_dataframe()* returns a pandas dataframe like:

```
print(df)
```

```
   number       value  params_x     state
0       0  103.129569 -8.155273  COMPLETE
1       1   51.054764 -5.145262  COMPLETE
2       2   42.103769 -4.488742  COMPLETE
3       3   58.836770 -5.670513  COMPLETE
4       4  120.920884 -8.996403  COMPLETE
5       5   69.976162 -6.365176  COMPLETE
```

A *Study* object also provides properties such as *trials*, *best_value*, *best_params* (see also *1. Lightweight, versatile, and platform agnostic architecture*).

```
print("Best params: ", study.best_params)
print("Best value: ", study.best_value)
print("Best Trial: ", study.best_trial)
print("Trials: ", study.trials)
```

```
Best params:  {'x': -4.488741736470017}
Best value:  42.103769322607924
Best Trial:  FrozenTrial(number=2, state=TrialState.COMPLETE, values=[42.
↪103769322607924], datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 819760),␣
↪datetime_complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 835578), params={'x': -4.
↪488741736470017}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=3, value=None)
Trials:  [FrozenTrial(number=0, state=TrialState.COMPLETE, values=[103.12956901377896],␣
↪datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 737304), datetime_
↪complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 759798), params={'x': -8.
```

(continues on next page)

(continued from previous page)

```
↪155272965990573}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=1, value=None), FrozenTrial(number=1, state=TrialState.COMPLETE, values=[51.
↪05476378686051], datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 787416),␣
↪datetime_complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 803735), params={'x': -5.
↪145261631799112}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=2, value=None), FrozenTrial(number=2, state=TrialState.COMPLETE, values=[42.
↪103769322607924], datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 819760),␣
↪datetime_complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 835578), params={'x': -4.
↪488741736470017}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=3, value=None), FrozenTrial(number=3, state=TrialState.COMPLETE, values=[58.
↪836769787129654], datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 900169),␣
↪datetime_complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 920170), params={'x': -5.
↪670513006776643}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=4, value=None), FrozenTrial(number=4, state=TrialState.COMPLETE, values=[120.
↪92088444119118], datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 941744),␣
↪datetime_complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 958085), params={'x': -8.
↪996403250208278}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=5, value=None), FrozenTrial(number=5, state=TrialState.COMPLETE, values=[69.
↪97616173137462], datetime_start=datetime.datetime(2023, 4, 12, 3, 5, 24, 973661),␣
↪datetime_complete=datetime.datetime(2023, 4, 12, 3, 5, 24, 989114), params={'x': -6.
↪365175535000722}, user_attrs={}, system_attrs={}, intermediate_values={},␣
↪distributions={'x': FloatDistribution(high=10.0, log=False, low=-10.0, step=None)},␣
↪trial_id=6, value=None)]
```

Total running time of the script: (0 minutes 0.663 seconds)

Multi-objective Optimization with Optuna

This tutorial showcases Optuna's multi-objective optimization feature by optimizing the validation accuracy of Fashion MNIST dataset and the FLOPS of the model implemented in PyTorch.

We use fvcore to measure FLOPS.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from fvcore.nn import FlopCountAnalysis

import optuna


DEVICE = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
DIR = ".."
BATCHSIZE = 128
N_TRAIN_EXAMPLES = BATCHSIZE * 30
```

(continues on next page)

(continued from previous page)

```
N_VALID_EXAMPLES = BATCHSIZE * 10


def define_model(trial):
    n_layers = trial.suggest_int("n_layers", 1, 3)
    layers = []

    in_features = 28 * 28
    for i in range(n_layers):
        out_features = trial.suggest_int("n_units_l{}".format(i), 4, 128)
        layers.append(nn.Linear(in_features, out_features))
        layers.append(nn.ReLU())
        p = trial.suggest_float("dropout_{}".format(i), 0.2, 0.5)
        layers.append(nn.Dropout(p))

        in_features = out_features

    layers.append(nn.Linear(in_features, 10))
    layers.append(nn.LogSoftmax(dim=1))

    return nn.Sequential(*layers)


# Defines training and evaluation.
def train_model(model, optimizer, train_loader):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.view(-1, 28 * 28).to(DEVICE), target.to(DEVICE)
        optimizer.zero_grad()
        F.nll_loss(model(data), target).backward()
        optimizer.step()


def eval_model(model, valid_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(valid_loader):
            data, target = data.view(-1, 28 * 28).to(DEVICE), target.to(DEVICE)
            pred = model(data).argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    accuracy = correct / N_VALID_EXAMPLES

    flops = FlopCountAnalysis(model, inputs=(torch.randn(1, 28 * 28).to(DEVICE),)).
↪total()
    return flops, accuracy
```

Define multi-objective objective function. Objectives are FLOPS and accuracy.

```
def objective(trial):
    train_dataset = torchvision.datasets.FashionMNIST(
```

(continues on next page)

(continued from previous page)

```
        DIR, train=True, download=True, transform=torchvision.transforms.ToTensor()
    )
    train_loader = torch.utils.data.DataLoader(
        torch.utils.data.Subset(train_dataset, list(range(N_TRAIN_EXAMPLES))),
        batch_size=BATCHSIZE,
        shuffle=True,
    )

    val_dataset = torchvision.datasets.FashionMNIST(
        DIR, train=False, transform=torchvision.transforms.ToTensor()
    )
    val_loader = torch.utils.data.DataLoader(
        torch.utils.data.Subset(val_dataset, list(range(N_VALID_EXAMPLES))),
        batch_size=BATCHSIZE,
        shuffle=True,
    )
    model = define_model(trial).to(DEVICE)

    optimizer = torch.optim.Adam(
        model.parameters(), trial.suggest_float("lr", 1e-5, 1e-1, log=True)
    )

    for epoch in range(10):
        train_model(model, optimizer, train_loader)
    flops, accuracy = eval_model(model, val_loader)
    return flops, accuracy
```

Run multi-objective optimization

If your optimization problem is multi-objective, Optuna assumes that you will specify the optimization direction for each objective. Specifically, in this example, we want to minimize the FLOPS (we want a faster model) and maximize the accuracy. So we set `directions` to `["minimize", "maximize"]`.

```
study = optuna.create_study(directions=["minimize", "maximize"])
study.optimize(objective, n_trials=30, timeout=300)

print("Number of finished trials: ", len(study.trials))
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-
↪ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-
↪ubyte.gz to ../FashionMNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/26421880 [00:00<?, ?it/s]
  0%|          | 34816/26421880 [00:00<01:15, 348142.07it/s]
  0%|          | 69632/26421880 [00:00<01:16, 346364.75it/s]
  0%|          | 104448/26421880 [00:00<01:16, 346238.89it/s]
  1%|          | 224256/26421880 [00:00<00:38, 678676.47it/s]
  2%|1         | 464896/26421880 [00:00<00:20, 1295608.34it/s]
  4%|3         | 946176/26421880 [00:00<00:10, 2477639.06it/s]
```

(continues on next page)

(continued from previous page)

```
  7%|7          | 1908736/26421880 [00:00<00:05, 4793579.40it/s]
 13%|#3         | 3500032/26421880 [00:00<00:02, 8292551.65it/s]
 19%|#9         | 5090304/26421880 [00:00<00:02, 10627288.97it/s]
 25%|##5        | 6687744/26421880 [00:01<00:01, 12236393.68it/s]
 31%|###1       | 8284160/26421880 [00:01<00:01, 13327356.10it/s]
 37%|###7       | 9892864/26421880 [00:01<00:01, 14127583.66it/s]
 44%|####3      | 11500544/26421880 [00:01<00:01, 14678811.86it/s]
 50%|####9      | 13114368/26421880 [00:01<00:00, 15079765.81it/s]
 56%|#####5     | 14739456/26421880 [00:01<00:00, 15389553.98it/s]
 62%|######1    | 16365568/26421880 [00:01<00:00, 15612489.23it/s]
 68%|######8    | 17996800/26421880 [00:01<00:00, 15782933.90it/s]
 74%|#######4   | 19633152/26421880 [00:01<00:00, 15900492.70it/s]
 81%|########   | 21281792/26421880 [00:01<00:00, 16035440.17it/s]
 87%|########6  | 22929408/26421880 [00:02<00:00, 16119988.19it/s]
 93%|#########3| 24595456/26421880 [00:02<00:00, 16239016.01it/s]
 99%|#########9| 26260480/26421880 [00:02<00:00, 16323864.16it/s]
26422272it [00:02, 11886309.54it/s]
Extracting ../FashionMNIST/raw/train-images-idx3-ubyte.gz to ../FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-
↪ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-
↪ubyte.gz to ../FashionMNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/29515 [00:00<?, ?it/s]
 42%|####1     | 12288/29515 [00:00<00:00, 122636.27it/s]
29696it [00:00, 294651.15it/s]
Extracting ../FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-
↪ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-
↪ubyte.gz to ../FashionMNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/4422102 [00:00<?, ?it/s]
  0%|          | 13312/4422102 [00:00<00:33, 132430.24it/s]
  1%|          | 43008/4422102 [00:00<00:19, 227524.52it/s]
  2%|2         | 103424/4422102 [00:00<00:10, 396762.31it/s]
  5%|5         | 223232/4422102 [00:00<00:05, 707958.48it/s]
 10%|#         | 463872/4422102 [00:00<00:03, 1310369.78it/s]
 21%|##1       | 945152/4422102 [00:00<00:01, 2482009.30it/s]
 43%|####3     | 1907712/4422102 [00:00<00:00, 4775569.12it/s]
 79%|#######8  | 3484672/4422102 [00:00<00:00, 8224334.09it/s]
4422656it [00:00, 5346048.14it/s]
Extracting ../FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-
↪ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-
↪ubyte.gz to ../FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/5148 [00:00<?, ?it/s]
```

(continues on next page)

(continued from previous page)

```
6144it [00:00, 32578765.84it/s]
Extracting ../FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../FashionMNIST/raw

Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Unsupported operator aten::log_softmax encountered 1 time(s)
Number of finished trials:  30
```

Check trials on Pareto front visually.

```
optuna.visualization.plot_pareto_front(study, target_names=["FLOPS", "accuracy"])
```

Fetch the list of trials on the Pareto front with *best_trials*.

For example, the following code shows the number of trials on the Pareto front and picks the trial with the highest accuracy.

```
print(f"Number of trials on the Pareto front: {len(study.best_trials)}")

trial_with_highest_accuracy = max(study.best_trials, key=lambda t: t.values[1])
print(f"Trial with highest accuracy: ")
print(f"\tnumber: {trial_with_highest_accuracy.number}")
print(f"\tparams: {trial_with_highest_accuracy.params}")
print(f"\tvalues: {trial_with_highest_accuracy.values}")
```

```
Number of trials on the Pareto front: 4
Trial with highest accuracy:
        number: 29
        params: {'n_layers': 2, 'n_units_l0': 107, 'dropout_0': 0.24874122641825303, 'n_
↪units_l1': 33, 'dropout_1': 0.4764291462189958, 'lr': 0.0024396077822291915}
        values: [87749.0, 0.82734375]
```

Learn which hyperparameters are affecting the flops most with hyperparameter importance.

```
optuna.visualization.plot_param_importances(
    study, target=lambda t: t.values[0], target_name="flops"
)
```

Total running time of the script: (2 minutes 3.644 seconds)

User Attributes

This feature is to annotate experiments with user-defined attributes.

Adding User Attributes to Studies

A *Study* object provides *set_user_attr()* method to register a pair of key and value as an user-defined attribute. A key is supposed to be a `str`, and a value be any object serializable with `json.dumps`.

```
import sklearn.datasets
import sklearn.model_selection
import sklearn.svm

import optuna


study = optuna.create_study(storage="sqlite:///example.db")
study.set_user_attr("contributors", ["Akiba", "Sano"])
study.set_user_attr("dataset", "MNIST")
```

We can access annotated attributes with *user_attrs* property.

```
study.user_attrs  # {'contributors': ['Akiba', 'Sano'], 'dataset': 'MNIST'}
```

```
{'contributors': ['Akiba', 'Sano'], 'dataset': 'MNIST'}
```

StudySummary object, which can be retrieved by *get_all_study_summaries()*, also contains user-defined attributes.

```
study_summaries = optuna.get_all_study_summaries("sqlite:///example.db")
study_summaries[0].user_attrs  # {"contributors": ["Akiba", "Sano"], "dataset": "MNIST"}
```

```
{'contributors': ['Akiba', 'Sano'], 'dataset': 'MNIST'}
```

See also:

`optuna study set-user-attr` command, which sets an attribute via command line interface.

Adding User Attributes to Trials

As with *Study*, a *Trial* object provides *set_user_attr()* method. Attributes are set inside an objective function.

```
def objective(trial):
    iris = sklearn.datasets.load_iris()
    x, y = iris.data, iris.target

    svc_c = trial.suggest_float("svc_c", 1e-10, 1e10, log=True)
    clf = sklearn.svm.SVC(C=svc_c)
    accuracy = sklearn.model_selection.cross_val_score(clf, x, y).mean()

    trial.set_user_attr("accuracy", accuracy)

    return 1.0 - accuracy  # return error for minimization


study.optimize(objective, n_trials=1)
```

We can access annotated attributes as:

```
study.trials[0].user_attrs
```

```
{'accuracy': 0.9400000000000001}
```

Note that, in this example, the attribute is not annotated to a *Study* but a single *Trial*.

Total running time of the script: (0 minutes 0.298 seconds)

Command-Line Interface

Command	Description
ask	Create a new trial and suggest parameters.
best-trial	Show the best trial.
best-trials	Show a list of trials located at the Pareto front.
create-study	Create a new study.
delete-study	Delete a specified study.
storage upgrade	Upgrade the schema of a storage.
studies	Show a list of studies.
study optimize	Start optimization of a study.
study set-user-attr	Set a user attribute to a study.
tell	Finish a trial, which was created by the ask command.
trials	Show a list of trials.

Optuna provides command-line interface as shown in the above table.

Let us assume you are not in IPython shell and writing Python script files instead. It is totally fine to write scripts like the following:

```
import optuna

```

(continues on next page)

(continued from previous page)

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


if __name__ == "__main__":
    study = optuna.create_study()
    study.optimize(objective, n_trials=100)
    print("Best value: {} (params: {})\n".format(study.best_value, study.best_params))
```

```
Best value: 7.112887315051957e-05 (params: {'x': 1.9915662064792574})
```

However, if we cannot write `objective` explicitly in Python code such as developing a new drug in a lab, an interactive way is suitable. In Optuna CLI, *Ask-and-Tell Interface* style commands provide such an interactive and flexible interface.

Let us assume we minimize the objective value depending on a parameter `x` in $[-10, 10]$ and objective value is calculated via some experiments by hand. Even so, we can invoke the optimization as follows. Don't care about `--storage sqlite:///example.db` for now, which is described in *Saving/Resuming Study with RDB Backend*.

```
$ STUDY_NAME=`optuna create-study --storage sqlite:///example.db`
$ optuna ask --storage sqlite:///example.db --study-name $STUDY_NAME --sampler␣
↪TPESampler \
    --search-space '{"x": {"name": "FloatDistribution", "attributes": {"step": null,
↪"low": -10.0, "high": 10.0, "log": false}}}'


[I 2022-08-20 06:08:53,158] Asked trial 0 with parameters {'x': 2.512238141966016}.
{"number": 0, "params": {"x": 2.512238141966016}}
```

The argument of `--search-space` option can be generated by using *`optuna.distributions.distribution_to_json()`*, for example, `optuna.distributions.distribution_to_json(optuna.distributions.FloatDistribution(-10, 10))`. Please refer to *`optuna.distributions.FloatDistribution`* and *`optuna.distributions.IntDistribution`* for detailed explanations of their arguments.

After conducting an experiment using the suggested parameter in the lab, we store the result to Optuna's study as follows:

```
$ optuna tell --storage sqlite:///example.db --study-name $STUDY_NAME --trial-number 0 --
↪values 0.7 --state complete
[I 2022-08-20 06:22:50,888] Told trial 0 with values [0.7] and state TrialState.COMPLETE.
```

Total running time of the script: (0 minutes 0.544 seconds)

User-Defined Sampler

Thanks to user-defined samplers, you can:

- experiment your own sampling algorithms,
- implement task-specific algorithms to refine the optimization performance, or
- wrap other optimization libraries to integrate them into Optuna pipelines (e.g., *SkoptSampler*).

This section describes the internal behavior of sampler classes and shows an example of implementing a user-defined sampler.

Overview of Sampler

A sampler has the responsibility to determine the parameter values to be evaluated in a trial. When a *suggest* API (e.g., *suggest_float()*) is called inside an objective function, the corresponding distribution object (e.g., *FloatDistribution*) is created internally. A sampler samples a parameter value from the distribution. The sampled value is returned to the caller of the *suggest* API and evaluated in the objective function.

To create a new sampler, you need to define a class that inherits *BaseSampler*. The base class has three abstract methods; *infer_relative_search_space()*, *sample_relative()*, and *sample_independent()*.

As the method names imply, Optuna supports two types of sampling: one is **relative sampling** that can consider the correlation of the parameters in a trial, and the other is **independent sampling** that samples each parameter independently.

At the beginning of a trial, *infer_relative_search_space()* is called to provide the relative search space for the trial. Then, *sample_relative()* is invoked to sample relative parameters from the search space. During the execution of the objective function, *sample_independent()* is used to sample parameters that don't belong to the relative search space.

Note: Please refer to the document of *BaseSampler* for further details.

An Example: Implementing SimulatedAnnealingSampler

For example, the following code defines a sampler based on Simulated Annealing (SA):

```python
import numpy as np
import optuna


class SimulatedAnnealingSampler(optuna.samplers.BaseSampler):
    def __init__(self, temperature=100):
        self._rng = np.random.RandomState()
        self._temperature = temperature  # Current temperature.
        self._current_trial = None  # Current state.

    def sample_relative(self, study, trial, search_space):
        if search_space == {}:
            return {}

        # Simulated Annealing algorithm.
```

(continues on next page)

(continued from previous page)

```
        # 1. Calculate transition probability.
        prev_trial = study.trials[-2]
        if self._current_trial is None or prev_trial.value <= self._current_trial.value:
            probability = 1.0
        else:
            probability = np.exp(
                (self._current_trial.value - prev_trial.value) / self._temperature
            )
        self._temperature *= 0.9  # Decrease temperature.

        # 2. Transit the current state if the previous result is accepted.
        if self._rng.uniform(0, 1) < probability:
            self._current_trial = prev_trial

        # 3. Sample parameters from the neighborhood of the current point.
        # The sampled parameters will be used during the next execution of
        # the objective function passed to the study.
        params = {}
        for param_name, param_distribution in search_space.items():
            if (
                not isinstance(param_distribution, optuna.distributions.
↪FloatDistribution)
                or (param_distribution.step is not None and param_distribution.step != 1)
                or param_distribution.log
            ):
                msg = (
                    "Only suggest_float() with `step` `None` or 1.0 and"
                    " `log` `False` is supported"
                )
                raise NotImplementedError(msg)

            current_value = self._current_trial.params[param_name]
            width = (param_distribution.high - param_distribution.low) * 0.1
            neighbor_low = max(current_value - width, param_distribution.low)
            neighbor_high = min(current_value + width, param_distribution.high)
            params[param_name] = self._rng.uniform(neighbor_low, neighbor_high)

        return params

    # The rest are unrelated to SA algorithm: boilerplate
    def infer_relative_search_space(self, study, trial):
        return optuna.search_space.intersection_search_space(study.get_
↪trials(deepcopy=False))

    def sample_independent(self, study, trial, param_name, param_distribution):
        independent_sampler = optuna.samplers.RandomSampler()
        return independent_sampler.sample_independent(study, trial, param_name, param_
↪distribution)
```

Note: In favor of code simplicity, the above implementation doesn't support some features (e.g., maximization). If you're interested in how to support those features, please see examples/samplers/simulated_annealing.py.

You can use SimulatedAnnealingSampler in the same way as built-in samplers as follows:

```
def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    y = trial.suggest_float("y", -5, 5)
    return x**2 + y


sampler = SimulatedAnnealingSampler()
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=100)

best_trial = study.best_trial
print("Best value: ", best_trial.value)
print("Parameters that achieve the best value: ", best_trial.params)
```

```
Best value:  -4.788503327100703
Parameters that achieve the best value:  {'x': -0.44990458761142715, 'y': -4.
↪990917465054512}
```

In this optimization, the values of x and y parameters are sampled by using SimulatedAnnealingSampler.sample_relative method.

Note: Strictly speaking, in the first trial, SimulatedAnnealingSampler.sample_independent method is used to sample parameter values. Because *intersection_search_space()* used in SimulatedAnnealingSampler.infer_relative_search_space cannot infer the search space if there are no complete trials.

Total running time of the script: (0 minutes 0.367 seconds)

User-Defined Pruner

In *optuna.pruners*, we described how an objective function can optionally include calls to a pruning feature which allows Optuna to terminate an optimization trial when intermediate results do not appear promising. In this document, we describe how to implement your own pruner, i.e., a custom strategy for determining when to stop a trial.

Overview of Pruning Interface

The *create_study()* constructor takes, as an optional argument, a pruner inheriting from *BasePruner*. The pruner should implement the abstract method *prune()*, which takes arguments for the associated *Study* and *Trial* and returns a boolean value: True if the trial should be pruned and False otherwise. Using the Study and Trial objects, you can access all other trials through the *get_trials()* method and, and from a trial, its reported intermediate values through the *intermediate_values()* (a dictionary which maps an integer step to a float value).

You can refer to the source code of the built-in Optuna pruners as templates for building your own. In this document, for illustration, we describe the construction and usage of a simple (but aggressive) pruner which prunes trials that are in last place compared to completed trials at the same step.

Note: Please refer to the documentation of *BasePruner* or, for example, *ThresholdPruner* or *PercentilePruner* for more robust examples of pruner implementation, including error checking and complex pruner-internal logic.

An Example: Implementing LastPlacePruner

We aim to optimize the loss and alpha hyperparameters for a stochastic gradient descent classifier (SGDClassifier) run on the sklearn iris dataset. We implement a pruner which terminates a trial at a certain step if it is in last place compared to completed trials at the same step. We begin considering pruning after a "warmup" of 1 training step and 5 completed trials. For demonstration purposes, we print() a diagnostic message from prune when it is about to return True (indicating pruning).

It may be important to note that the SGDClassifier score, as it is evaluated on a holdout set, decreases with enough training steps due to overfitting. This means that a trial could be pruned even if it had a favorable (high) value on a previous training set. After pruning, Optuna will take the intermediate value last reported as the value of the trial.

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

import optuna
from optuna.pruners import BasePruner
from optuna.trial._state import TrialState


class LastPlacePruner(BasePruner):
    def __init__(self, warmup_steps, warmup_trials):
        self._warmup_steps = warmup_steps
        self._warmup_trials = warmup_trials

    def prune(self, study: "optuna.study.Study", trial: "optuna.trial.FrozenTrial") ->␣
↪bool:
        # Get the latest score reported from this trial
        step = trial.last_step

        if step:  # trial.last_step == None when no scores have been reported yet
            this_score = trial.intermediate_values[step]

            # Get scores from other trials in the study reported at the same step
            completed_trials = study.get_trials(deepcopy=False, states=(TrialState.
↪COMPLETE,))
            other_scores = [
                t.intermediate_values[step]
                for t in completed_trials
                if step in t.intermediate_values
            ]
            other_scores = sorted(other_scores)

            # Prune if this trial at this step has a lower value than all completed␣
↪trials
            # at the same step. Note that steps will begin numbering at 0 in the␣
↪objective
            # function definition below.
            if step >= self._warmup_steps and len(other_scores) > self._warmup_trials:
                if this_score < other_scores[0]:
                    print(f"prune() True: Trial {trial.number}, Step {step}, Score {this_
```

(continues on next page)

(continued from previous page)

```
↪score}")
                    return True

        return False
```

Lastly, let's confirm the implementation is correct with the simple hyperparameter optimization.

```
def objective(trial):
    iris = load_iris()
    classes = np.unique(iris.target)
    X_train, X_valid, y_train, y_valid = train_test_split(
        iris.data, iris.target, train_size=100, test_size=50, random_state=0
    )

    loss = trial.suggest_categorical("loss", ["hinge", "log_loss", "perceptron"])
    alpha = trial.suggest_float("alpha", 0.00001, 0.001, log=True)
    clf = SGDClassifier(loss=loss, alpha=alpha, random_state=0)
    score = 0

    for step in range(0, 5):
        clf.partial_fit(X_train, y_train, classes=classes)
        score = clf.score(X_valid, y_valid)

        trial.report(score, step)

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

    return score


pruner = LastPlacePruner(warmup_steps=1, warmup_trials=5)
study = optuna.create_study(direction="maximize", pruner=pruner)
study.optimize(objective, n_trials=50)
```

```
prune() True: Trial 9, Step 3, Score 0.48
prune() True: Trial 15, Step 3, Score 0.48
prune() True: Trial 17, Step 1, Score 0.62
```

Total running time of the script: (0 minutes 0.989 seconds)

Callback for Study.optimize

This tutorial showcases how to use & implement Optuna `Callback` for *`optimize()`*.

`Callback` is called after every evaluation of `objective`, and it takes *`Study`* and *`FrozenTrial`* as arguments, and does some work.

`MLflowCallback` is a great example.

Stop optimization after some trials are pruned in a row

This example implements a stateful callback which stops the optimization if a certain number of trials are pruned in a row. The number of trials pruned in a row is specified by `threshold`.

```
import optuna


class StopWhenTrialKeepBeingPrunedCallback:
    def __init__(self, threshold: int):
        self.threshold = threshold
        self._consequtive_pruned_count = 0

    def __call__(self, study: optuna.study.Study, trial: optuna.trial.FrozenTrial) ->␣
↪None:
        if trial.state == optuna.trial.TrialState.PRUNED:
            self._consequtive_pruned_count += 1
        else:
            self._consequtive_pruned_count = 0

        if self._consequtive_pruned_count >= self.threshold:
            study.stop()
```

This objective prunes all the trials except for the first 5 trials (`trial.number` starts with 0).

```
def objective(trial):
    if trial.number > 4:
        raise optuna.TrialPruned

    return trial.suggest_float("x", 0, 1)
```

Here, we set the threshold to 2: optimization finishes once two trials are pruned in a row. So, we expect this study to stop after 7 trials.

```
import logging
import sys

# Add stream handler of stdout to show the messages
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))

study_stop_cb = StopWhenTrialKeepBeingPrunedCallback(2)
study = optuna.create_study()
study.optimize(objective, n_trials=10, callbacks=[study_stop_cb])
```

```
A new study created in memory with name: no-name-2d8269da-2aaa-494f-8ed8-2c625f472e32
Trial 0 finished with value: 0.246129169977082 and parameters: {'x': 0.246129169977082}.␣
↪Best is trial 0 with value: 0.246129169977082.
Trial 1 finished with value: 0.7371913594932391 and parameters: {'x': 0.7371913594932391}
↪. Best is trial 0 with value: 0.246129169977082.
Trial 2 finished with value: 0.28093950604935836 and parameters: {'x': 0.
↪28093950604935836}. Best is trial 0 with value: 0.246129169977082.
Trial 3 finished with value: 0.6270005851593332 and parameters: {'x': 0.6270005851593332}
↪. Best is trial 0 with value: 0.246129169977082.
```

(continues on next page)

(continued from previous page)

```
Trial 4 finished with value: 0.7254767175920509 and parameters: {'x': 0.7254767175920509}
↪. Best is trial 0 with value: 0.246129169977082.
Trial 5 pruned.
Trial 6 pruned.
```

As you can see in the log above, the study stopped after 7 trials as expected.

Total running time of the script: (0 minutes 0.007 seconds)

Specify Hyperparameters Manually

It's natural that you have some specific sets of hyperparameters to try first such as initial learning rate values and the number of leaves. Also, it's possible that you've already tried those sets before having Optuna find better sets of hyperparameters.

Optuna provides two APIs to support such cases:

1. Passing those sets of hyperparameters and let Optuna evaluate them - *enqueue_trial()*
2. Adding the results of those sets as completed Trials - *add_trial()*

First Scenario: Have Optuna evaluate your hyperparameters

In this scenario, let's assume you have some out-of-box sets of hyperparameters but have not evaluated them yet and decided to use Optuna to find better sets of hyperparameters.

Optuna has *optuna.study.Study.enqueue_trial()* which lets you pass those sets of hyperparameters to Optuna and Optuna will evaluate them.

This section walks you through how to use this lit API with LightGBM.

```python
import lightgbm as lgb
import numpy as np
import sklearn.datasets
import sklearn.metrics
from sklearn.model_selection import train_test_split

import optuna
```

Define the objective function.

```python
def objective(trial):
    data, target = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, valid_x, train_y, valid_y = train_test_split(data, target, test_size=0.25)
    dtrain = lgb.Dataset(train_x, label=train_y)
    dvalid = lgb.Dataset(valid_x, label=valid_y)

    param = {
        "objective": "binary",
        "metric": "auc",
        "verbosity": -1,
        "boosting_type": "gbdt",
        "bagging_fraction": min(trial.suggest_float("bagging_fraction", 0.4, 1.0 + 1e-
```

(continues on next page)

(continued from previous page)

```
↪12), 1),
        "bagging_freq": trial.suggest_int("bagging_freq", 0, 7),
        "min_child_samples": trial.suggest_int("min_child_samples", 5, 100),
    }

    # Add a callback for pruning.
    pruning_callback = optuna.integration.LightGBMPruningCallback(trial, "auc")
    gbm = lgb.train(param, dtrain, valid_sets=[dvalid], callbacks=[pruning_callback])

    preds = gbm.predict(valid_x)
    pred_labels = np.rint(preds)
    accuracy = sklearn.metrics.accuracy_score(valid_y, pred_labels)
    return accuracy
```

Then, construct Study for hyperparameter optimization.

```
study = optuna.create_study(direction="maximize", pruner=optuna.pruners.MedianPruner())
```

Here, we get Optuna evaluate some sets with larger "bagging_fraq" value and the default values.

```
study.enqueue_trial(
    {
        "bagging_fraction": 1.0,
        "bagging_freq": 0,
        "min_child_samples": 20,
    }
)

study.enqueue_trial(
    {
        "bagging_fraction": 0.75,
        "bagging_freq": 5,
        "min_child_samples": 20,
    }
)

import logging
import sys

# Add stream handler of stdout to show the messages to see Optuna works expectedly.
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))
study.optimize(objective, n_trials=100, timeout=600)
```

```
Trial 0 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 1.0,
↪'bagging_freq': 0, 'min_child_samples': 20}. Best is trial 0 with value: 0.
↪972027972027972.
Trial 1 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.75,
↪ 'bagging_freq': 5, 'min_child_samples': 20}. Best is trial 1 with value: 0.
↪986013986013986.
Trial 2 finished with value: 0.958041958041958 and parameters: {'bagging_fraction': 0.
↪5734118261549108, 'bagging_freq': 5, 'min_child_samples': 36}. Best is trial 1 with␣
↪value: 0.986013986013986.
```

(continues on next page)

(continued from previous page)

```
Trial 3 finished with value: 0.965034965034965 and parameters: {'bagging_fraction': 0.
↪9226284232942232, 'bagging_freq': 2, 'min_child_samples': 72}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 4 finished with value: 0.958041958041958 and parameters: {'bagging_fraction': 0.
↪8027152468885187, 'bagging_freq': 0, 'min_child_samples': 43}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 5 pruned. Trial was pruned at iteration 0.
Trial 6 pruned. Trial was pruned at iteration 0.
Trial 7 pruned. Trial was pruned at iteration 0.
Trial 8 pruned. Trial was pruned at iteration 0.
Trial 9 pruned. Trial was pruned at iteration 0.
Trial 10 pruned. Trial was pruned at iteration 0.
Trial 11 pruned. Trial was pruned at iteration 0.
Trial 12 pruned. Trial was pruned at iteration 0.
Trial 13 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction': 0.
↪9837855979501038, 'bagging_freq': 4, 'min_child_samples': 23}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 14 pruned. Trial was pruned at iteration 0.
Trial 15 pruned. Trial was pruned at iteration 74.
Trial 16 pruned. Trial was pruned at iteration 0.
Trial 17 pruned. Trial was pruned at iteration 0.
Trial 18 pruned. Trial was pruned at iteration 0.
Trial 19 pruned. Trial was pruned at iteration 0.
Trial 20 pruned. Trial was pruned at iteration 0.
Trial 21 pruned. Trial was pruned at iteration 0.
Trial 22 finished with value: 0.965034965034965 and parameters: {'bagging_fraction': 0.
↪9433007257983781, 'bagging_freq': 3, 'min_child_samples': 29}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 23 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction': 0.
↪9221201912791204, 'bagging_freq': 1, 'min_child_samples': 14}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 24 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪9065718548872596, 'bagging_freq': 1, 'min_child_samples': 12}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 25 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 0.
↪8594562177623648, 'bagging_freq': 5, 'min_child_samples': 41}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 26 pruned. Trial was pruned at iteration 2.
Trial 27 pruned. Trial was pruned at iteration 0.
Trial 28 pruned. Trial was pruned at iteration 0.
Trial 29 pruned. Trial was pruned at iteration 0.
Trial 30 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪9961005965428342, 'bagging_freq': 5, 'min_child_samples': 38}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 31 pruned. Trial was pruned at iteration 0.
Trial 32 pruned. Trial was pruned at iteration 0.
Trial 33 pruned. Trial was pruned at iteration 0.
Trial 34 pruned. Trial was pruned at iteration 0.
Trial 35 pruned. Trial was pruned at iteration 0.
Trial 36 pruned. Trial was pruned at iteration 0.
Trial 37 pruned. Trial was pruned at iteration 0.
Trial 38 pruned. Trial was pruned at iteration 0.
```

(continues on next page)

(continued from previous page)

```
Trial 39 pruned. Trial was pruned at iteration 0.
Trial 40 pruned. Trial was pruned at iteration 0.
Trial 41 pruned. Trial was pruned at iteration 0.
Trial 42 pruned. Trial was pruned at iteration 0.
Trial 43 pruned. Trial was pruned at iteration 0.
Trial 44 pruned. Trial was pruned at iteration 0.
Trial 45 pruned. Trial was pruned at iteration 0.
Trial 46 pruned. Trial was pruned at iteration 0.
Trial 47 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪9343606874145038, 'bagging_freq': 4, 'min_child_samples': 16}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 48 pruned. Trial was pruned at iteration 0.
Trial 49 pruned. Trial was pruned at iteration 0.
Trial 50 pruned. Trial was pruned at iteration 0.
Trial 51 pruned. Trial was pruned at iteration 0.
Trial 52 pruned. Trial was pruned at iteration 8.
Trial 53 pruned. Trial was pruned at iteration 0.
Trial 54 pruned. Trial was pruned at iteration 0.
Trial 55 pruned. Trial was pruned at iteration 0.
Trial 56 pruned. Trial was pruned at iteration 0.
Trial 57 pruned. Trial was pruned at iteration 0.
Trial 58 pruned. Trial was pruned at iteration 0.
Trial 59 pruned. Trial was pruned at iteration 0.
Trial 60 pruned. Trial was pruned at iteration 0.
Trial 61 pruned. Trial was pruned at iteration 0.
Trial 62 pruned. Trial was pruned at iteration 20.
Trial 63 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 0.
↪9794856209574876, 'bagging_freq': 0, 'min_child_samples': 15}. Best is trial 1 with␣
↪value: 0.986013986013986.
Trial 64 pruned. Trial was pruned at iteration 0.
Trial 65 pruned. Trial was pruned at iteration 0.
Trial 66 pruned. Trial was pruned at iteration 0.
Trial 67 pruned. Trial was pruned at iteration 0.
Trial 68 pruned. Trial was pruned at iteration 0.
Trial 69 pruned. Trial was pruned at iteration 0.
Trial 70 pruned. Trial was pruned at iteration 0.
Trial 71 pruned. Trial was pruned at iteration 0.
Trial 72 pruned. Trial was pruned at iteration 0.
Trial 73 pruned. Trial was pruned at iteration 0.
Trial 74 pruned. Trial was pruned at iteration 0.
Trial 75 pruned. Trial was pruned at iteration 0.
Trial 76 pruned. Trial was pruned at iteration 0.
Trial 77 pruned. Trial was pruned at iteration 0.
Trial 78 pruned. Trial was pruned at iteration 0.
Trial 79 pruned. Trial was pruned at iteration 0.
Trial 80 pruned. Trial was pruned at iteration 0.
Trial 81 pruned. Trial was pruned at iteration 0.
Trial 82 pruned. Trial was pruned at iteration 0.
Trial 83 pruned. Trial was pruned at iteration 0.
Trial 84 pruned. Trial was pruned at iteration 0.
Trial 85 pruned. Trial was pruned at iteration 0.
Trial 86 pruned. Trial was pruned at iteration 0.
```

(continues on next page)

(continued from previous page)

```
Trial 87 pruned. Trial was pruned at iteration 0.
Trial 88 pruned. Trial was pruned at iteration 0.
Trial 89 pruned. Trial was pruned at iteration 0.
Trial 90 pruned. Trial was pruned at iteration 0.
Trial 91 pruned. Trial was pruned at iteration 0.
Trial 92 pruned. Trial was pruned at iteration 0.
Trial 93 pruned. Trial was pruned at iteration 0.
Trial 94 pruned. Trial was pruned at iteration 0.
Trial 95 pruned. Trial was pruned at iteration 0.
Trial 96 pruned. Trial was pruned at iteration 0.
Trial 97 pruned. Trial was pruned at iteration 0.
Trial 98 pruned. Trial was pruned at iteration 0.
Trial 99 pruned. Trial was pruned at iteration 0.
```

Second scenario: Have Optuna utilize already evaluated hyperparameters

In this scenario, let's assume you have some out-of-box sets of hyperparameters and you have already evaluated them but the results are not desirable so that you are thinking of using Optuna.

Optuna has *optuna.study.Study.add_trial()* which lets you register those results to Optuna and then Optuna will sample hyperparameters taking them into account.

In this section, the `objective` is the same as the first scenario.

```python
study = optuna.create_study(direction="maximize", pruner=optuna.pruners.MedianPruner())
study.add_trial(
    optuna.trial.create_trial(
        params={
            "bagging_fraction": 1.0,
            "bagging_freq": 0,
            "min_child_samples": 20,
        },
        distributions={
            "bagging_fraction": optuna.distributions.FloatDistribution(0.4, 1.0 + 1e-12),
            "bagging_freq": optuna.distributions.IntDistribution(0, 7),
            "min_child_samples": optuna.distributions.IntDistribution(5, 100),
        },
        value=0.94,
    )
)
study.add_trial(
    optuna.trial.create_trial(
        params={
            "bagging_fraction": 0.75,
            "bagging_freq": 5,
            "min_child_samples": 20,
        },
        distributions={
            "bagging_fraction": optuna.distributions.FloatDistribution(0.4, 1.0 + 1e-12),
            "bagging_freq": optuna.distributions.IntDistribution(0, 7),
            "min_child_samples": optuna.distributions.IntDistribution(5, 100),
```

(continues on next page)

(continued from previous page)

```
        },
        value=0.95,
    )
)
study.optimize(objective, n_trials=100, timeout=600)
```

```
A new study created in memory with name: no-name-783f91ee-209f-491b-8f20-ef6e8de3a696
Trial 2 finished with value: 0.9440559440559441 and parameters: {'bagging_fraction': 0.
↪82209813689795, 'bagging_freq': 7, 'min_child_samples': 69}. Best is trial 1 with␣
↪value: 0.95.
Trial 3 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪6623793908402318, 'bagging_freq': 4, 'min_child_samples': 37}. Best is trial 3 with␣
↪value: 0.986013986013986.
Trial 4 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 0.
↪5066180182046016, 'bagging_freq': 2, 'min_child_samples': 12}. Best is trial 3 with␣
↪value: 0.986013986013986.
Trial 5 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction': 0.
↪9980826825105295, 'bagging_freq': 7, 'min_child_samples': 23}. Best is trial 3 with␣
↪value: 0.986013986013986.
Trial 6 pruned. Trial was pruned at iteration 0.
Trial 7 pruned. Trial was pruned at iteration 0.
Trial 8 pruned. Trial was pruned at iteration 0.
Trial 9 pruned. Trial was pruned at iteration 0.
Trial 10 pruned. Trial was pruned at iteration 0.
Trial 11 pruned. Trial was pruned at iteration 0.
Trial 12 pruned. Trial was pruned at iteration 0.
Trial 13 pruned. Trial was pruned at iteration 0.
Trial 14 pruned. Trial was pruned at iteration 0.
Trial 15 pruned. Trial was pruned at iteration 0.
Trial 16 pruned. Trial was pruned at iteration 0.
Trial 17 pruned. Trial was pruned at iteration 0.
Trial 18 pruned. Trial was pruned at iteration 0.
Trial 19 finished with value: 0.951048951048951 and parameters: {'bagging_fraction': 0.
↪9382543108834875, 'bagging_freq': 2, 'min_child_samples': 6}. Best is trial 3 with␣
↪value: 0.986013986013986.
Trial 20 pruned. Trial was pruned at iteration 0.
Trial 21 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪5538859989988378, 'bagging_freq': 2, 'min_child_samples': 5}. Best is trial 3 with␣
↪value: 0.986013986013986.
Trial 22 pruned. Trial was pruned at iteration 0.
Trial 23 pruned. Trial was pruned at iteration 0.
Trial 24 pruned. Trial was pruned at iteration 0.
Trial 25 pruned. Trial was pruned at iteration 0.
Trial 26 pruned. Trial was pruned at iteration 0.
Trial 27 pruned. Trial was pruned at iteration 0.
Trial 28 pruned. Trial was pruned at iteration 0.
Trial 29 pruned. Trial was pruned at iteration 0.
Trial 30 pruned. Trial was pruned at iteration 0.
Trial 31 pruned. Trial was pruned at iteration 0.
Trial 32 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪4935591061172445, 'bagging_freq': 1, 'min_child_samples': 17}. Best is trial 3 with␣
```

(continues on next page)

(continued from previous page)

```
↪value: 0.986013986013986.
Trial 33 pruned. Trial was pruned at iteration 0.
Trial 34 pruned. Trial was pruned at iteration 0.
Trial 35 pruned. Trial was pruned at iteration 0.
Trial 36 pruned. Trial was pruned at iteration 0.
Trial 37 pruned. Trial was pruned at iteration 0.
Trial 38 pruned. Trial was pruned at iteration 0.
Trial 39 pruned. Trial was pruned at iteration 0.
Trial 40 pruned. Trial was pruned at iteration 0.
Trial 41 pruned. Trial was pruned at iteration 0.
Trial 42 pruned. Trial was pruned at iteration 0.
Trial 43 pruned. Trial was pruned at iteration 0.
Trial 44 pruned. Trial was pruned at iteration 0.
Trial 45 pruned. Trial was pruned at iteration 0.
Trial 46 pruned. Trial was pruned at iteration 0.
Trial 47 pruned. Trial was pruned at iteration 0.
Trial 48 pruned. Trial was pruned at iteration 0.
Trial 49 pruned. Trial was pruned at iteration 0.
Trial 50 pruned. Trial was pruned at iteration 0.
Trial 51 pruned. Trial was pruned at iteration 0.
Trial 52 pruned. Trial was pruned at iteration 0.
Trial 53 finished with value: 0.986013986013986 and parameters: {'bagging_fraction': 0.
↪9669473466744595, 'bagging_freq': 1, 'min_child_samples': 14}. Best is trial 3 with␣
↪value: 0.986013986013986.
Trial 54 pruned. Trial was pruned at iteration 0.
Trial 55 pruned. Trial was pruned at iteration 0.
Trial 56 pruned. Trial was pruned at iteration 0.
Trial 57 pruned. Trial was pruned at iteration 0.
Trial 58 pruned. Trial was pruned at iteration 0.
Trial 59 pruned. Trial was pruned at iteration 0.
Trial 60 pruned. Trial was pruned at iteration 0.
Trial 61 pruned. Trial was pruned at iteration 0.
Trial 62 pruned. Trial was pruned at iteration 29.
Trial 63 pruned. Trial was pruned at iteration 0.
Trial 64 pruned. Trial was pruned at iteration 0.
Trial 65 pruned. Trial was pruned at iteration 0.
Trial 66 pruned. Trial was pruned at iteration 0.
Trial 67 pruned. Trial was pruned at iteration 0.
Trial 68 pruned. Trial was pruned at iteration 0.
Trial 69 pruned. Trial was pruned at iteration 0.
Trial 70 pruned. Trial was pruned at iteration 0.
Trial 71 pruned. Trial was pruned at iteration 37.
Trial 72 pruned. Trial was pruned at iteration 0.
Trial 73 pruned. Trial was pruned at iteration 0.
Trial 74 finished with value: 1.0 and parameters: {'bagging_fraction': 0.
↪7767294093026217, 'bagging_freq': 5, 'min_child_samples': 15}. Best is trial 74 with␣
↪value: 1.0.
Trial 75 pruned. Trial was pruned at iteration 0.
Trial 76 pruned. Trial was pruned at iteration 0.
Trial 77 pruned. Trial was pruned at iteration 0.
Trial 78 pruned. Trial was pruned at iteration 0.
Trial 79 pruned. Trial was pruned at iteration 0.
```

(continues on next page)

(continued from previous page)

```
Trial 80 pruned. Trial was pruned at iteration 0.
Trial 81 pruned. Trial was pruned at iteration 0.
Trial 82 pruned. Trial was pruned at iteration 0.
Trial 83 pruned. Trial was pruned at iteration 0.
Trial 84 pruned. Trial was pruned at iteration 0.
Trial 85 pruned. Trial was pruned at iteration 0.
Trial 86 pruned. Trial was pruned at iteration 0.
Trial 87 pruned. Trial was pruned at iteration 0.
Trial 88 pruned. Trial was pruned at iteration 0.
Trial 89 pruned. Trial was pruned at iteration 0.
Trial 90 pruned. Trial was pruned at iteration 0.
Trial 91 pruned. Trial was pruned at iteration 0.
Trial 92 pruned. Trial was pruned at iteration 0.
Trial 93 pruned. Trial was pruned at iteration 0.
Trial 94 pruned. Trial was pruned at iteration 0.
Trial 95 pruned. Trial was pruned at iteration 0.
Trial 96 pruned. Trial was pruned at iteration 25.
Trial 97 pruned. Trial was pruned at iteration 0.
Trial 98 pruned. Trial was pruned at iteration 0.
Trial 99 pruned. Trial was pruned at iteration 0.
Trial 100 pruned. Trial was pruned at iteration 0.
Trial 101 pruned. Trial was pruned at iteration 0.
```

Total running time of the script: (0 minutes 9.604 seconds)

Ask-and-Tell Interface

Optuna has an *Ask-and-Tell* interface, which provides a more flexible interface for hyperparameter optimization. This tutorial explains three use-cases when the ask-and-tell interface is beneficial:

- *Apply Optuna to an existing optimization problem with minimum modifications*
- *Define-and-Run*
- *Batch Optimization*

Apply Optuna to an existing optimization problem with minimum modifications

Let's consider the traditional supervised classification problem; you aim to maximize the validation accuracy. To do so, you train *LogisticRegression* as a simple model.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

import optuna


X, y = make_classification(n_features=10)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

(continues on next page)

(continued from previous page)

```
C = 0.01
clf = LogisticRegression(C=C)
clf.fit(X_train, y_train)
val_accuracy = clf.score(X_test, y_test)  # the objective
```

Then you try to optimize hyperparameters C and solver of the classifier by using optuna. When you introduce optuna naively, you define an objective function such that it takes trial and calls suggest_* methods of trial to sample the hyperparameters:

```
def objective(trial):
    X, y = make_classification(n_features=10)
    X_train, X_test, y_train, y_test = train_test_split(X, y)

    C = trial.suggest_float("C", 1e-7, 10.0, log=True)
    solver = trial.suggest_categorical("solver", ("lbfgs", "saga"))

    clf = LogisticRegression(C=C, solver=solver)
    clf.fit(X_train, y_train)
    val_accuracy = clf.score(X_test, y_test)

    return val_accuracy


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=10)
```

This interface is not flexible enough. For example, if objective requires additional arguments other than trial, you need to define a class as in How to define objective functions that have own arguments?. The ask-and-tell interface provides a more flexible syntax to optimize hyperparameters. The following example is equivalent to the previous code block.

```
study = optuna.create_study(direction="maximize")

n_trials = 10
for _ in range(n_trials):
    trial = study.ask()  # `trial` is a `Trial` and not a `FrozenTrial`.

    C = trial.suggest_float("C", 1e-7, 10.0, log=True)
    solver = trial.suggest_categorical("solver", ("lbfgs", "saga"))

    clf = LogisticRegression(C=C, solver=solver)
    clf.fit(X_train, y_train)
    val_accuracy = clf.score(X_test, y_test)

    study.tell(trial, val_accuracy)  # tell the pair of trial and objective value
```

The main difference is to use two methods: *optuna.study.Study.ask()* and *optuna.study.Study.tell()*. *optuna.study.Study.ask()* creates a trial that can sample hyperparameters, and *optuna.study.Study.tell()* finishes the trial by passing trial and an objective value. You can apply Optuna's hyperparameter optimization to your original code without an objective function.

If you want to make your optimization faster with a pruner, you need to explicitly pass the state of trial to the argument of `optuna.study.Study.tell()` method as follows:

```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

# define study with hyperband pruner.
study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.HyperbandPruner(
        min_resource=1, max_resource=n_train_iter, reduction_factor=3
    ),
)

for _ in range(20):
    trial = study.ask()

    alpha = trial.suggest_float("alpha", 0.0, 1.0)

    clf = SGDClassifier(alpha=alpha)
    pruned_trial = False

    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():
            pruned_trial = True
            break

    if pruned_trial:
        study.tell(trial, state=optuna.trial.TrialState.PRUNED)  # tell the pruned state
    else:
        score = clf.score(X_valid, y_valid)
        study.tell(trial, score)  # tell objective value
```

Note: `optuna.study.Study.tell()` method can take a trial number rather than the trial object. `study.tell(trial.number, y)` is equivalent to `study.tell(trial, y)`.

Define-and-Run

The ask-and-tell interface supports both *define-by-run* and *define-and-run* APIs. This section shows the example of the *define-and-run* API in addition to the define-by-run example above.

Define distributions for the hyperparameters before calling the *optuna.study.Study.ask()* method for define-and-run API. For example,

```
distributions = {
    "C": optuna.distributions.FloatDistribution(1e-7, 10.0, log=True),
    "solver": optuna.distributions.CategoricalDistribution(("lbfgs", "saga")),
}
```

Pass distributions to *optuna.study.Study.ask()* method at each call. The retuned trial contains the suggested hyperparameters.

```
study = optuna.create_study(direction="maximize")
n_trials = 10
for _ in range(n_trials):
    trial = study.ask(distributions)  # pass the pre-defined distributions.

    # two hyperparameters are already sampled from the pre-defined distributions
    C = trial.params["C"]
    solver = trial.params["solver"]

    clf = LogisticRegression(C=C, solver=solver)
    clf.fit(X_train, y_train)
    val_accuracy = clf.score(X_test, y_test)

    study.tell(trial, val_accuracy)
```

Batch Optimization

The ask-and-tell interface enables us to optimize a batched objective for faster optimization. For example, parallelizable evaluation, operation over vectors, etc.

The following objective takes batched hyperparameters xs and ys instead of a single pair of hyperparameters x and y and calculates the objective over the full vectors.

```
def batched_objective(xs: np.ndarray, ys: np.ndarray):
    return xs**2 + ys
```

In the following example, the number of pairs of hyperparameters in a batch is 10, and batched_objective is evaluated three times. Thus, the number of trials is 30. Note that you need to store either trial_numbers or trial to call *optuna.study.Study.tell()* method after the batched evaluations.

```
batch_size = 10
study = optuna.create_study(sampler=optuna.samplers.CmaEsSampler())

for _ in range(3):
    # create batch
    trial_numbers = []
    x_batch = []
```

(continues on next page)

(continued from previous page)

```
    y_batch = []
    for _ in range(batch_size):
        trial = study.ask()
        trial_numbers.append(trial.number)
        x_batch.append(trial.suggest_float("x", -10, 10))
        y_batch.append(trial.suggest_float("y", -10, 10))

    # evaluate batched objective
    x_batch = np.array(x_batch)
    y_batch = np.array(y_batch)
    objectives = batched_objective(x_batch, y_batch)

    # finish all trials in the batch
    for trial_number, objective in zip(trial_numbers, objectives):
        study.tell(trial_number, objective)
```

Total running time of the script: (0 minutes 0.145 seconds)

Re-use the best trial

In some cases, you may want to re-evaluate the objective function with the best hyperparameters again after the hyperparameter optimization.

For example,

- You have found good hyperparameters with Optuna and want to run a similar *objective* function using the best hyperparameters found so far to further analyze the results, or
- You have optimized with Optuna using a partial dataset to reduce training time. After the hyperparameter tuning, you want to train the model using the whole dataset with the best hyperparameter values found.

`best_trial` provides an interface to re-evaluate the objective function with the current best hyperparameter values.

This tutorial shows an example of how to re-run a different *objective* function with the current best values, like the first example above.

Investigating the best model further

Let's consider a classical supervised classification problem with Optuna as follows:

```
from sklearn import metrics
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split


import optuna


def objective(trial):
    X, y = make_classification(n_features=10, random_state=1)
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

(continues on next page)

(continued from previous page)

```
    C = trial.suggest_float("C", 1e-7, 10.0, log=True)

    clf = LogisticRegression(C=C)
    clf.fit(X_train, y_train)

    return clf.score(X_test, y_test)


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

print(study.best_trial.value)  # Show the best value.
```

```
0.92
```

Suppose after the hyperparameter optimization, you want to calculate other evaluation metrics such as recall, precision, and f1-score on the same dataset. You can define another objective function that shares most of the `objective` function to reproduce the model with the best hyperparameters.

```
def detailed_objective(trial):
    # Use same code objective to reproduce the best model
    X, y = make_classification(n_features=10, random_state=1)
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)

    C = trial.suggest_float("C", 1e-7, 10.0, log=True)

    clf = LogisticRegression(C=C)
    clf.fit(X_train, y_train)

    # calculate more evaluation metrics
    pred = clf.predict(X_test)

    acc = metrics.accuracy_score(pred, y_test)
    recall = metrics.recall_score(pred, y_test)
    precision = metrics.precision_score(pred, y_test)
    f1 = metrics.f1_score(pred, y_test)

    return acc, f1, recall, precision
```

Pass `study.best_trial` as the argument of `detailed_objective`.

```
detailed_objective(study.best_trial)  # calculate acc, f1, recall, and precision
```

```
(0.92, 0.9285714285714286, 0.9285714285714286, 0.9285714285714286)
```

The difference between `best_trial` and ordinal trials

This uses *best_trial*, which returns the *best_trial* as a *FrozenTrial*. The *FrozenTrial* is different from an active trial and behaves differently from *Trial* in some situations. For example, pruning does not work because *should_prune* always returns `False`.

Note: For multi-objective optimization as demonstrated by *Multi-objective Optimization with Optuna*, `best_trials` returns a list of *FrozenTrial* on Pareto front. So we can re-use each trial in the list by the similar way above.

Total running time of the script: (0 minutes 0.059 seconds)

6.3 API Reference

6.3.1 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_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 | 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** (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** (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 *Saving/Resuming Study with RDB Backend* 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

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

```
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

```
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_summaries

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

Get all history of studies stored in a specified storage.

Example

```
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)
```

6.3.2 optuna.cli

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

For detail, please see the result of

```
$ optuna --help
```

See also:

The *Command-Line Interface* tutorial provides use-cases with examples.

6.3.3 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 [low, 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 [low, 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 [low, high] is not divisible by q, high will be replaced with the maximum of $kq + \mathrm{low} < \mathrm{high}$, 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 $k \times \text{step} + \text{low} < \text{high}$, 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

6.3.4 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.

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

```
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()
```

(continues on next page)

(continued from previous page)

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


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

optuna.exceptions.CLIUsageError

exception optuna.exceptions.**CLIUsageError**

Exception for CLI.

CLI raises this exception when it receives invalid configuration.

optuna.exceptions.StorageInternalError

exception optuna.exceptions.**StorageInternalError**

Exception for storage operation.

This error is raised when an operation failed in backend DB of storage.

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.

6.3.5 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* or *MeanDecreaseImpurityImportanceEvaluator* as a guide, paying close attention to the format of the return value from the Evaluator's `evaluate` function.

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.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 of type `collections.OrderedDict` and 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*.
- **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

An `collections.OrderedDict` 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

An `collections.OrderedDict` 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

An `collections.OrderedDict` where the keys are parameter names and the values are assessed importances.

Return type

Dict[str, float]

6.3.6 optuna.integration

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

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.

BoTorch

optuna.integration.BoTorchSampler	A sampler that uses BoTorch, a Bayesian optimization library built on top of PyTorch.
optuna.integration.botorch.qei_candidates_func	Quasi MC-based batch Expected Improvement (qEI).
optuna.integration.botorch.qehvi_candidates_func	Quasi MC-based batch Expected Hypervolume Improvement (qEHVI).
optuna.integration.botorch.qnehvi_candidates_func	Quasi MC-based batch Expected Noisy Hypervolume Improvement (qNEHVI).
optuna.integration.botorch.qparego_candidates_func	Quasi MC-based extended ParEGO (qParEGO) for constrained multi-objective optimization.

optuna.integration.BoTorchSampler

class `optuna.integration.`**BoTorchSampler**(**, candidates_func=None, constraints_func=None, n_startup_trials=10, independent_sampler=None, seed=None, device=None*)

A sampler that uses BoTorch, a Bayesian optimization library built on top of PyTorch.

This sampler allows using BoTorch's optimization algorithms from Optuna to suggest parameter configurations. Parameters are transformed to continuous space and passed to BoTorch, and then transformed back to Optuna's representations. Categorical parameters are one-hot encoded.

See also:

See an example how to use the sampler.

See also:

See the BoTorch homepage for details and for how to implement your own `candidates_func`.

Note: An instance of this sampler *should not be used with different studies* when used with constraints. Instead, a new instance should be created for each new study. The reason for this is that the sampler is stateful keeping all the computed constraints.

Parameters

- **candidates_func** (*Callable[[torch.Tensor, torch.Tensor, torch.Tensor | None, torch.Tensor], torch.Tensor] | None*) – An optional function that suggests the next candidates. It must take the training data, the objectives, the constraints, the search space bounds and return the next candidates. The arguments are of type `torch.Tensor`. The return value must be a `torch.Tensor`. However, if `constraints_func` is omitted, constraints will be `None`. For any constraints that failed to compute, the tensor will contain NaN.

  If omitted, it is determined automatically based on the number of objectives. If the number of objectives is one, Quasi MC-based batch Expected Improvement (qEI) is used. If the number of objectives is either two or three, Quasi MC-based batch Expected Hypervolume

Improvement (qEHVI) is used. Otherwise, for larger number of objectives, the faster Quasi MC-based extended ParEGO (qParEGO) is used.

The function should assume *maximization* of the objective.

See also:

See `optuna.integration.botorch.qei_candidates_func()` for an example.

- **`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.

  If omitted, no constraints will be passed to `candidates_func` nor taken into account during suggestion.

- **`n_startup_trials`** (*`int`*) – Number of initial trials, that is the number of trials to resort to independent sampling.
- **`independent_sampler`** (`BaseSampler` *`| None`*) – An independent sampler to use for the initial trials and for parameters that are conditional.
- **`seed`** (*`int | None`*) – Seed for random number generator.
- **`device`** (*`torch.device | None`*) – A `torch.device` to store input and output data of BoTorch. Please set a CUDA device if you fasten sampling.

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.
`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

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.integration.botorch.qei_candidates_func

optuna.integration.botorch.**qei_candidates_func**(*train_x*, *train_obj*, *train_con*, *bounds*)

Quasi MC-based batch Expected Improvement (qEI).

The default value of `candidates_func` in *BoTorchSampler* with single-objective optimization.

Parameters

- **train_x** (*Tensor*) – Previous parameter configurations. A `torch.Tensor` of shape `(n_trials, n_params)`. `n_trials` is the number of already observed trials and `n_params` is the number of parameters. `n_params` may be larger than the actual number of parameters if categorical parameters are included in the search space, since these parameters are one-hot encoded. Values are not normalized.
- **train_obj** (*Tensor*) – Previously observed objectives. A `torch.Tensor` of shape `(n_trials, n_objectives)`. `n_trials` is identical to that of `train_x`. `n_objectives` is the number of objectives. Observations are not normalized.
- **train_con** (*Tensor | None*) – Objective constraints. A `torch.Tensor` of shape `(n_trials, n_constraints)`. `n_trials` is identical to that of `train_x`.

`n_constraints` is the number of constraints. A constraint is violated if strictly larger than 0. If no constraints are involved in the optimization, this argument will be `None`.

- **bounds** (*Tensor*) – Search space bounds. A `torch.Tensor` of shape `(2, n_params)`. `n_params` is identical to that of `train_x`. The first and the second rows correspond to the lower and upper bounds for each parameter respectively.

Returns
Next set of candidates. Usually the return value of BoTorch's `optimize_acqf`.

Return type
Tensor

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.

optuna.integration.botorch.qehvi_candidates_func

`optuna.integration.botorch.`**`qehvi_candidates_func`**`(`*train_x*, *train_obj*, *train_con*, *bounds*`)`

Quasi MC-based batch Expected Hypervolume Improvement (qEHVI).

The default value of `candidates_func` in *BoTorchSampler* with multi-objective optimization when the number of objectives is three or less.

See also:

qei_candidates_func() for argument and return value descriptions.

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

- **train_x** (*Tensor*) –
- **train_obj** (*Tensor*) –
- **train_con** (*Tensor | None*) –
- **bounds** (*Tensor*) –

Return type
Tensor

optuna.integration.botorch.qnehvi_candidates_func

`optuna.integration.botorch.`**`qnehvi_candidates_func`**`(`*train_x*, *train_obj*, *train_con*, *bounds*`)`

Quasi MC-based batch Expected Noisy Hypervolume Improvement (qNEHVI).

According to Botorch/Ax documentation, this function may perform better than qEHVI (*qehvi_candidates_func*). (cf. https://botorch.org/tutorials/constrained_multi_objective_bo)

See also:

qei_candidates_func() for argument and return value descriptions.

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.

Parameters

- **train_x** (*Tensor*) –
- **train_obj** (*Tensor*) –
- **train_con** (*Tensor | None*) –
- **bounds** (*Tensor*) –

Return type

Tensor

optuna.integration.botorch.qparego_candidates_func

`optuna.integration.botorch.`**`qparego_candidates_func`**(*train_x*, *train_obj*, *train_con*, *bounds*)

Quasi MC-based extended ParEGO (qParEGO) for constrained multi-objective optimization.

The default value of `candidates_func` in `BoTorchSampler` with multi-objective optimization when the number of objectives is larger than three.

See also:

`qei_candidates_func()` for argument and return value descriptions.

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

- **train_x** (*Tensor*) –
- **train_obj** (*Tensor*) –
- **train_con** (*Tensor | None*) –
- **bounds** (*Tensor*) –

Return type

Tensor

Catalyst

`optuna.integration.CatalystPruningCallback`	Catalyst callback to prune unpromising trials.

optuna.integration.CatalystPruningCallback

class `optuna.integration.`**`CatalystPruningCallback`**(**args*, ***kwargs*)

Catalyst callback to prune unpromising trials.

This class is an alias to Catalyst's OptunaPruningCallback.

See the Catalyst's documentation for the detailed description.

Warning: Deprecated in v2.7.0. 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/v2.7.0.

CatBoost

`optuna.integration.CatBoostPruningCallback`	Callback for catboost to prune unpromising trials.

optuna.integration.CatBoostPruningCallback

class `optuna.integration.`**`CatBoostPruningCallback`**(*trial*, *metric*, *eval_set_index=None*)

Callback for catboost to prune unpromising trials.

See the example if you want to add a pruning callback which observes validation accuracy of a CatBoost model.

Note: *`optuna.TrialPruned`* cannot be raised in *`after_iteration()`* that is called in CatBoost via `CatBoostPruningCallback`. You must call *`check_pruned()`* after training manually unlike other pruning callbacks to raise *`optuna.TrialPruned`*.

Note: This callback cannot be used with CatBoost on GPUs because CatBoost doesn't support a user-defined callback for GPU. Please refer to CatBoost issue.

Parameters

- **`trial`** (`Trial`) – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`metric`** (*`str`*) – An evaluation metric for pruning, e.g., `Logloss` and `AUC`. Please refer to CatBoost reference for further details.
- **`eval_set_index`** (*`int | None`*) – The index of the target validation dataset. If you set only one `eval_set`, `eval_set_index` is None. If you set multiple datasets as `eval_set`, the index of `eval_set` must be `eval_set_index`, e.g., `0` or `1` when `eval_set` contains two datasets.

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_iteration(info)	Report an evaluation metric value for Optuna pruning after each CatBoost's iteration.
check_pruned()	Raise *optuna.TrialPruned* manually if the CatBoost optimization is pruned.

after_iteration(*info*)

Report an evaluation metric value for Optuna pruning after each CatBoost's iteration.

This method is called by CatBoost.

Parameters

info (*Any*) – A `SimpleNamespace` containing iteraion, `validation_name`, `metric_name` and history of losses. For example `SimpleNamespace(iteration=2, metrics={ 'learn': {'Logloss': [0.6, 0.5]}, 'validation': {'Logloss': [0.7, 0.6], 'AUC': [0.8, 0.9]} })`.

Returns

A boolean value. If `False`, CatBoost internally stops the optimization with Optuna's pruning logic without raising *optuna.TrialPruned*. Otherwise, the optimization continues.

Return type

bool

check_pruned()

Raise *optuna.TrialPruned* manually if the CatBoost optimization is pruned.

Return type

None

Dask

optuna.integration.DaskStorage	Dask-compatible storage class.

optuna.integration.DaskStorage

class optuna.integration.**DaskStorage**(*storage=None*, *name=None*, *client=None*, *register=True*)

Dask-compatible storage class.

This storage class wraps a Optuna storage class (e.g. Optuna's in-memory or sqlite storage) and is used to run optimization trials in parallel on a Dask cluster. The underlying Optuna storage object lives on the cluster's scheduler and any method calls on the *DaskStorage* instance results in the same method being called on the underlying Optuna storage object.

See this example for how to use *DaskStorage* to extend Optuna's in-memory storage class to run across multiple processes.

Parameters

- **storage** (*None | str | BaseStorage*) – Optuna storage url to use for underlying Optuna storage class to wrap (e.g. `None` for in-memory storage, `sqlite:///example.db` for SQLite storage). Defaults to `None`.

- **name** (*str | None*) – Unique identifier for the Dask storage class. Specifying a custom name can sometimes be useful for logging or debugging. If `None` is provided, a random name will be automatically generated.
- **client** (*distributed.Client | None*) – Dask `Client` to connect to. If not provided, will attempt to find an existing `Client`.
- **register** (*bool*) – Whether or not to register this storage instance with the cluster scheduler. Most common usage of this storage class will not need to specify this argument. Defaults to `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.

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_base_storage()	Retrieve underlying Optuna storage instance from the scheduler.
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.
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.

Attributes

```
client
```

`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_base_storage`()

Retrieve underlying Optuna storage instance from the scheduler.

This is a convenience method to extract the Optuna storage instance stored on the Dask scheduler process to the local Python process.

Return type
BaseStorage

`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*) – 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*]

`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** (*Any*) – 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

fast.ai

optuna.integration.FastAIV1PruningCallback	FastAI callback to prune unpromising trials for fastai.
optuna.integration.FastAIV2PruningCallback	FastAI callback to prune unpromising trials for fastai.
optuna.integration.FastAIPruningCallback	alias of *FastAIV2PruningCallback*

optuna.integration.FastAIV1PruningCallback

class optuna.integration.**FastAIV1PruningCallback**(*learn*, *trial*, *monitor*)

FastAI callback to prune unpromising trials for fastai.

Note: This callback is for fastai<2.0.

See the example if you want to add a pruning callback which monitors validation loss of a `Learner`.

Example

Register a pruning callback to `learn.fit` and `learn.fit_one_cycle`.

```
learn.fit(n_epochs, callbacks=[FastAIPruningCallback(learn, trial, "valid_loss")])
learn.fit_one_cycle(
    n_epochs,
    cyc_len,
    max_lr,
    callbacks=[FastAIPruningCallback(learn, trial, "valid_loss")],
)
```

Parameters

- **learn** (*Learner*) – fastai.basic_train.Learner.
- **trial** (`Trial`) – A *Trial* corresponding to the current evaluation of the objective function.
- **monitor** (*str*) – An evaluation metric for pruning, e.g. `valid_loss` and `Accuracy`. Please refer to fastai.callbacks.TrackerCallback reference for further details.

Warning: Deprecated in v2.4.0. 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/v2.4.0.

Methods

on_epoch_end(epoch, **kwargs)

optuna.integration.FastAIV2PruningCallback

class optuna.integration.**FastAIV2PruningCallback**(*trial*, *monitor='valid_loss'*)

FastAI callback to prune unpromising trials for fastai.

Note: This callback is for fastai>=2.0.

See the example if you want to add a pruning callback which monitors validation loss of a `Learner`.

Example

Register a pruning callback to `learn.fit` and `learn.fit_one_cycle`.

```
learn = cnn_learner(dls, resnet18, metrics=[error_rate])
learn.fit(n_epochs, cbs=[FastAIPruningCallback(trial)])  # Monitor "valid_loss"
learn.fit_one_cycle(
    n_epochs,
    lr_max,
    cbs=[FastAIPruningCallback(trial, monitor="error_rate")],  # Monitor "error_rate
↪"
)
```

Parameters

- **trial** (Trial) – A *Trial* corresponding to the current evaluation of the objective function.
- **monitor** (*str*) – An evaluation metric for pruning, e.g. `valid_loss` or `accuracy`. Please refer to fastai.callback.TrackerCallback reference for further details.

Methods

after_epoch()

after_fit()

optuna.integration.FastAIPruningCallback

`optuna.integration.FastAIPruningCallback`

alias of *`FastAIV2PruningCallback`*

LightGBM

`optuna.integration.LightGBMPruningCallback`	Callback for LightGBM to prune unpromising trials.
`optuna.integration.lightgbm.train`	Wrapper of LightGBM Training API to tune hyperparameters.
`optuna.integration.lightgbm.LightGBMTuner`	Hyperparameter tuner for LightGBM.
`optuna.integration.lightgbm.LightGBMTunerCV`	Hyperparameter tuner for LightGBM with cross-validation.

optuna.integration.LightGBMPruningCallback

class `optuna.integration.`**`LightGBMPruningCallback`**(*trial*, *metric*, *valid_name='valid_0'*, *report_interval=1*)

Callback for LightGBM to prune unpromising trials.

See the example if you want to add a pruning callback which observes accuracy of a LightGBM model.

Parameters

- **`trial`** (`Trial`) – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`metric`** (*`str`*) – An evaluation metric for pruning, e.g., `binary_error` and `multi_error`. Please refer to LightGBM reference for further details.
- **`valid_name`** (*`str`*) – The name of the target validation. Validation names are specified by `valid_names` option of train method. If omitted, `valid_0` is used which is the default name of the first validation. Note that this argument will be ignored if you are calling cv method instead of train method.
- **`report_interval`** (*`int`*) – Check if the trial should report intermediate values for pruning every n-th boosting iteration. By default `report_interval=1` and reporting is performed after every iteration. Note that the pruning itself is performed according to the interval definition of the pruner.

optuna.integration.lightgbm.train

`optuna.integration.lightgbm.`**`train`**(**args*, ***kwargs*)

Wrapper of LightGBM Training API to tune hyperparameters.

It tunes important hyperparameters (e.g., `min_child_samples` and `feature_fraction`) in a stepwise manner. It is a drop-in replacement for lightgbm.train(). See a simple example of LightGBM Tuner which optimizes the validation log loss of cancer detection.

`train()` is a wrapper function of *`LightGBMTuner`*. To use feature in Optuna such as suspended/resumed optimization and/or parallelization, refer to *`LightGBMTuner`* instead of this function.

Arguments and keyword arguments for lightgbm.train() can be passed.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

optuna.integration.lightgbm.LightGBMTuner

class optuna.integration.lightgbm.**LightGBMTuner**(*params*, *train_set*, *num_boost_round=1000*, *valid_sets=None*, *valid_names=None*, *fobj=None*, *feval=None*, *feature_name='auto'*, *categorical_feature='auto'*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval='warn'*, *learning_rates=None*, *keep_training_booster=False*, *callbacks=None*, *time_budget=None*, *sample_size=None*, *study=None*, *optuna_callbacks=None*, *model_dir=None*, *verbosity=None*, *show_progress_bar=True*, ***, *optuna_seed=None*)

Hyperparameter tuner for LightGBM.

It optimizes the following hyperparameters in a stepwise manner: `lambda_l1`, `lambda_l2`, `num_leaves`, `feature_fraction`, `bagging_fraction`, `bagging_freq` and `min_child_samples`.

You can find the details of the algorithm and benchmark results in this blog article by Kohei Ozaki, a Kaggle Grandmaster.

Arguments and keyword arguments for lightgbm.train() can be passed. The arguments that only *LightGBMTuner* has are listed below:

Parameters

- **time_budget** (*int | None*) – A time budget for parameter tuning in seconds.
- **study** (Study *| None*) – A *Study* instance to store optimization results. The *Trial* instances in it has the following user attributes: `elapsed_secs` is the elapsed time since the optimization starts. `average_iteration_time` is the average time of iteration to train the booster model in the trial. `lgbm_params` is a JSON-serialized dictionary of LightGBM parameters used in the trial.
- **optuna_callbacks** (*List[Callable[[*Study*,* FrozenTrial*], None]] | None*) – List of Optuna 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*. Please note that this is not a `callbacks` argument of lightgbm.train() .
- **model_dir** (*str | None*) – A directory to save boosters. By default, it is set to `None` and no boosters are saved. Please set shared directory (e.g., directories on NFS) if you want to access *get_best_booster()* in distributed environments. Otherwise, it may raise `ValueError`. If the directory does not exist, it will be created. The filenames of the boosters will be `{model_dir}/{trial_number}.pkl` (e.g., `./boosters/0.pkl`).
- **verbosity** (*int | None*) – A verbosity level to change Optuna's logging level. The level is aligned to LightGBM's verbosity .

Warning: Deprecated in v2.0.0. `verbosity` 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.

Please use *set_verbosity()* instead.

- **show_progress_bar** (*bool*) – Flag to show progress bars or not. To disable progress bar, set this `False`.

Note: Progress bars will be fragmented by logging messages of LightGBM and Optuna. Please suppress such messages to show the progress bars properly.

- **optuna_seed** (*int | None*) – seed of *TPESampler* for random number generator that affects sampling for `num_leaves`, `bagging_fraction`, `bagging_freq`, `lambda_l1`, and `lambda_l2`.

Note: The deterministic parameter of LightGBM makes training reproducible. Please enable it when you use this argument.

- **params** (*Dict[str, Any]*) –
- **train_set** (*lgb.Dataset*) –
- **num_boost_round** (*int*) –
- **valid_sets** (*VALID_SET_TYPE | None*) –
- **valid_names** (*Any | None*) –
- **fobj** (*Callable[[...], Any] | None*) –
- **feval** (*Callable[[...], Any] | None*) –
- **feature_name** (*str*) –
- **categorical_feature** (*str*) –
- **early_stopping_rounds** (*int | None*) –
- **evals_result** (*Dict[Any, Any] | None*) –
- **verbose_eval** (*bool | int | str | None*) –
- **learning_rates** (*List[float] | None*) –
- **keep_training_booster** (*bool*) –
- **callbacks** (*List[Callable[[...], Any]] | None*) –
- **sample_size** (*int | None*) –

Methods

`compare_validation_metrics`(val_score, best_score)	
`get_best_booster`()	Return the best booster.
`higher_is_better`()	
`run`()	Perform the hyperparameter-tuning with given parameters.
`sample_train_set`()	Make subset of *self.train_set* Dataset object.
`tune_bagging`([n_trials])	
`tune_feature_fraction`([n_trials])	
`tune_feature_fraction_stage2`([n_trials])	
`tune_min_data_in_leaf`()	
`tune_num_leaves`([n_trials])	
`tune_regularization_factors`([n_trials])	

Attributes

`best_params`	Return parameters of the best booster.
`best_score`	Return the score of the best booster.

property best_params: Dict[str, Any]

Return parameters of the best booster.

property best_score: float

Return the score of the best booster.

get_best_booster()

Return the best booster.

If the best booster cannot be found, `ValueError` will be raised. To prevent the errors, please save boosters by specifying the `model_dir` argument of `__init__()`, when you resume tuning or you run tuning in parallel.

Return type
Booster

run()

Perform the hyperparameter-tuning with given parameters.

Return type
None

sample_train_set()

Make subset of *self.train_set* Dataset object.

Return type
None

optuna.integration.lightgbm.LightGBMTunerCV

`class optuna.integration.lightgbm.LightGBMTunerCV`(*params*, *train_set*, *num_boost_round=1000*, *folds=None*, *nfold=5*, *stratified=True*, *shuffle=True*, *fobj=None*, *feval=None*, *feature_name='auto'*, *categorical_feature='auto'*, *early_stopping_rounds=None*, *fpreproc=None*, *verbose_eval=None*, *show_stdv=True*, *seed=0*, *callbacks=None*, *time_budget=None*, *sample_size=None*, *study=None*, *optuna_callbacks=None*, *verbosity=None*, *show_progress_bar=True*, *model_dir=None*, *return_cvbooster=False*, ***, *optuna_seed=None*)

Hyperparameter tuner for LightGBM with cross-validation.

It employs the same stepwise approach as `LightGBMTuner`. `LightGBMTunerCV` invokes lightgbm.cv() to train and validate boosters while `LightGBMTuner` invokes lightgbm.train(). See a simple example which optimizes the validation log loss of cancer detection.

Arguments and keyword arguments for lightgbm.cv() can be passed except `metrics`, `init_model` and `eval_train_metric`. The arguments that only `LightGBMTunerCV` has are listed below:

Parameters

- **time_budget** (*int | None*) – A time budget for parameter tuning in seconds.
- **study** (Study | *None*) – A `Study` instance to store optimization results. The `Trial` instances in it has the following user attributes: `elapsed_secs` is the elapsed time since the optimization starts. `average_iteration_time` is the average time of iteration to train the booster model in the trial. `lgbm_params` is a JSON-serialized dictionary of LightGBM parameters used in the trial.
- **optuna_callbacks** (*List[Callable[[Study, FrozenTrial], None]] | None*) – List of Optuna 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`. Please note that this is not a `callbacks` argument of lightgbm.train() .
- **model_dir** (*str | None*) – A directory to save boosters. By default, it is set to `None` and no boosters are saved. Please set shared directory (e.g., directories on NFS) if you want to access `get_best_booster()` in distributed environments. Otherwise, it may raise `ValueError`. If the directory does not exist, it will be created. The filenames of the boosters will be `{model_dir}/{trial_number}.pkl` (e.g., `./boosters/0.pkl`).
- **verbosity** (*int | None*) – A verbosity level to change Optuna's logging level. The level is aligned to LightGBM's verbosity .

> **Warning:** Deprecated in v2.0.0. `verbosity` 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.
>
> Please use `set_verbosity()` instead.

- **show_progress_bar** (*bool*) – Flag to show progress bars or not. To disable progress bar, set this `False`.

  Note: Progress bars will be fragmented by logging messages of LightGBM and Optuna. Please suppress such messages to show the progress bars properly.

- **return_cvbooster** (*bool*) – Flag to enable *get_best_booster()*.
- **optuna_seed** (*int | None*) – seed of *TPESampler* for random number generator that affects sampling for `num_leaves`, `bagging_fraction`, `bagging_freq`, `lambda_l1`, and `lambda_l2`.

  Note: The deterministic parameter of LightGBM makes training reproducible. Please enable it when you use this argument.

- **params** (*Dict[str, Any]*) –
- **train_set** (*lgb.Dataset*) –
- **num_boost_round** (*int*) –
- **folds** (*Generator[Tuple[int, int], None, None] | Iterator[Tuple[int, int]] | BaseCrossValidator | None*) –
- **nfold** (*int*) –
- **stratified** (*bool*) –
- **shuffle** (*bool*) –
- **fobj** (*Callable[[...], Any] | None*) –
- **feval** (*Callable[[...], Any] | None*) –
- **feature_name** (*str*) –
- **categorical_feature** (*str*) –
- **early_stopping_rounds** (*int | None*) –
- **fpreproc** (*Callable[[...], Any] | None*) –
- **verbose_eval** (*bool | int | None*) –
- **show_stdv** (*bool*) –
- **seed** (*int*) –
- **callbacks** (*List[Callable[[...], Any]] | None*) –
- **sample_size** (*int | None*) –

Methods

`compare_validation_metrics`(val_score, best_score)	
`get_best_booster`()	Return the best cvbooster.
`higher_is_better`()	
`run`()	Perform the hyperparameter-tuning with given parameters.
`sample_train_set`()	Make subset of *self.train_set* Dataset object.
`tune_bagging`([n_trials])	
`tune_feature_fraction`([n_trials])	
`tune_feature_fraction_stage2`([n_trials])	
`tune_min_data_in_leaf`()	
`tune_num_leaves`([n_trials])	
`tune_regularization_factors`([n_trials])	

Attributes

`best_params`	Return parameters of the best booster.
`best_score`	Return the score of the best booster.

property best_params: `Dict[str, Any]`

Return parameters of the best booster.

property best_score: `float`

Return the score of the best booster.

`get_best_booster`()

Return the best cvbooster.

If the best booster cannot be found, `ValueError` will be raised. To prevent the errors, please save boosters by specifying both of the `model_dir` and the `return_cvbooster` arguments of `__init__()`, when you resume tuning or you run tuning in parallel.

Return type

CVBooster

`run`()

Perform the hyperparameter-tuning with given parameters.

Return type

None

`sample_train_set`()

Make subset of *self.train_set* Dataset object.

Return type
None

MLflow

optuna.integration.MLflowCallback	Callback to track Optuna trials with MLflow.

optuna.integration.MLflowCallback

class optuna.integration.**MLflowCallback**(*tracking_uri=None*, *metric_name='value'*, *create_experiment=True*, *mlflow_kwargs=None*, *tag_study_user_attrs=False*, *tag_trial_user_attrs=True*)

Callback to track Optuna trials with MLflow.

This callback adds relevant information that is tracked by Optuna to MLflow.

Example

Add MLflow callback to Optuna optimization.

```
import optuna
from optuna.integration.mlflow import MLflowCallback


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


mlflc = MLflowCallback(
    tracking_uri=YOUR_TRACKING_URI,
    metric_name="my metric score",
)

study = optuna.create_study(study_name="my_study")
study.optimize(objective, n_trials=10, callbacks=[mlflc])
```

Parameters

- **tracking_uri** (*str | None*) – The URI of the MLflow tracking server.

  Please refer to mlflow.set_tracking_uri for more details.

- **metric_name** (*str | Sequence[str]*) – Name assigned to optimized metric. In case of multi-objective optimization, list of names can be passed. Those names will be assigned to metrics in the order returned by objective function. If single name is provided, or this argument is left to default value, it will be broadcasted to each objective with a number suffix in order returned by objective function e.g. two objectives and default metric name will be logged as `value_0` and `value_1`. The number of metrics must be the same as the number of values an objective function returns.

- **create_experiment** (*bool*) – When `True`, new MLflow experiment will be created for each optimization run, named after the Optuna study. Setting this argument to `False` lets user run optimization under existing experiment, set via mlflow.set_experiment, by passing `experiment_id` as one of `mlflow_kwargs` or under default MLflow experiment, when no additional arguments are passed. Note that this argument must be set to `False` when using Optuna with this callback within Databricks Notebook.
- **mlflow_kwargs** (*Dict[str, Any] | None*) – Set of arguments passed when initializing MLflow run. Please refer to MLflow API documentation for more details.

  Note: `nest_trials` argument added in v2.3.0 is a part of `mlflow_kwargs` since v3.0.0. Anyone using `nest_trials=True` should migrate to `mlflow_kwargs={"nested": True}` to avoid raising `TypeError`.

- **tag_study_user_attrs** (*bool*) – Flag indicating whether or not to add the study's user attrs to the mlflow trial as tags. Please note that when this flag is set, key value pairs in *user_attrs* will supersede existing tags.
- **tag_trial_user_attrs** (*bool*) – Flag indicating whether or not to add the trial's user attrs to the mlflow trial as tags. Please note that when both trial and study user attributes are logged, the latter will supersede the former in case of a collision.

Note: Added in v1.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v1.4.0.

Methods

track_in_mlflow()	Decorator for using MLflow logging in the objective function.

track_in_mlflow()

Decorator for using MLflow logging in the objective function.

This decorator enables the extension of MLflow logging provided by the callback.

All information logged in the decorated objective function will be added to the MLflow run for the trial created by the callback.

Example

Add additional logging to MLflow.

```
import optuna
import mlflow
from optuna.integration.mlflow import MLflowCallback

mlflc = MLflowCallback(
    tracking_uri=YOUR_TRACKING_URI,
    metric_name="my metric score",
```

(continues on next page)

(continued from previous page)

```
)


@mlflc.track_in_mlflow()
def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    mlflow.log_param("power", 2)
    mlflow.log_metric("base of metric", x - 2)

    return (x - 2) ** 2


study = optuna.create_study(study_name="my_other_study")
study.optimize(objective, n_trials=10, callbacks=[mlflc])
```

Returns
Objective function with tracking to MLflow enabled.

Return type
Callable

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.

Weights & Biases

optuna.integration.WeightsAndBiasesCallback	Callback to track Optuna trials with Weights & Biases.

optuna.integration.WeightsAndBiasesCallback

class optuna.integration.**WeightsAndBiasesCallback**(*metric_name='value'*, *wandb_kwargs=None*, *as_multirun=False*)

Callback to track Optuna trials with Weights & Biases.

This callback enables tracking of Optuna study in Weights & Biases. The study is tracked as a single experiment run, where all suggested hyperparameters and optimized metrics are logged and plotted as a function of optimizer steps.

Note: User needs to be logged in to Weights & Biases before using this callback in online mode. For more information, please refer to wandb setup.

Note: Users who want to run multiple Optuna studies within the same process should call `wandb.finish()` between subsequent calls to `study.optimize()`. Calling `wandb.finish()` is not necessary if you are running one Optuna study per process.

Note: To ensure correct trial order in Weights & Biases, this callback should only be used with `study.optimize(n_jobs=1)`.

Example

Add Weights & Biases callback to Optuna optimization.

```
import optuna
from optuna.integration.wandb import WeightsAndBiasesCallback


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


study = optuna.create_study()

wandb_kwargs = {"project": "my-project"}
wandbc = WeightsAndBiasesCallback(wandb_kwargs=wandb_kwargs)

study.optimize(objective, n_trials=10, callbacks=[wandbc])
```

Weights & Biases logging in multirun mode.

```
import optuna
from optuna.integration.wandb import WeightsAndBiasesCallback

wandb_kwargs = {"project": "my-project"}
wandbc = WeightsAndBiasesCallback(wandb_kwargs=wandb_kwargs, as_multirun=True)


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


study = optuna.create_study()
study.optimize(objective, n_trials=10, callbacks=[wandbc])
```

Parameters

- **metric_name** (*str | Sequence[str]*) – Name assigned to optimized metric. In case of multi-objective optimization, list of names can be passed. Those names will be assigned to metrics in the order returned by objective function. If single name is provided, or this argument is left to default value, it will be broadcasted to each objective with a number suffix in order returned by objective function e.g. two objectives and default metric name will be logged as `value_0` and `value_1`. The number of metrics must be the same as the number of values objective function returns.

- **wandb_kwargs** (*Dict[str, Any] | None*) – Set of arguments passed when initializing Weights & Biases run. Please refer to Weights & Biases API documentation for more details.
- **as_multirun** (*bool*) – Creates new runs for each trial. Useful for generating W&B Sweeps like panels (for ex., parameter importance, parallel coordinates, etc).

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.

Methods

track_in_wandb()	Decorator for using W&B for logging inside the objective function.

track_in_wandb()

Decorator for using W&B for logging inside the objective function.

The run is initialized with the same `wandb_kwargs` that are passed to the callback. All the metrics from inside the objective function will be logged into the same run which stores the parameters for a given trial.

Example

Add additional logging to Weights & Biases.

```
import optuna
from optuna.integration.wandb import WeightsAndBiasesCallback
import wandb

wandb_kwargs = {"project": "my-project"}
wandbc = WeightsAndBiasesCallback(wandb_kwargs=wandb_kwargs, as_multirun=True)


@wandbc.track_in_wandb()
def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    wandb.log({"power": 2, "base of metric": x - 2})

    return (x - 2) ** 2


study = optuna.create_study()
study.optimize(objective, n_trials=10, callbacks=[wandbc])
```

Returns
Objective function with W&B tracking enabled.

Return type
Callable

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.

MXNet

optuna.integration.MXNetPruningCallback	MXNet callback to prune unpromising trials.

optuna.integration.MXNetPruningCallback

class optuna.integration.**MXNetPruningCallback**(*trial*, *eval_metric*)

MXNet callback to prune unpromising trials.

See the example if you want to add a pruning callback which observes accuracy.

Parameters

- **trial** (Trial) – A *Trial* corresponding to the current evaluation of the objective function.
- **eval_metric** (*str*) – An evaluation metric name for pruning, e.g., `cross-entropy` and `accuracy`. If using default metrics like mxnet.metrics.Accuracy, use it's default metric name. For custom metrics, use the metric_name provided to constructor. Please refer to mxnet.metrics reference for further details.

pycma

optuna.integration.PyCmaSampler	A Sampler using cma library as the backend.
optuna.integration.CmaEsSampler	Wrapper class of PyCmaSampler for backward compatibility.

optuna.integration.PyCmaSampler

class optuna.integration.**PyCmaSampler**(*x0=None*, *sigma0=None*, *cma_stds=None*, *seed=None*, *cma_opts=None*, *n_startup_trials=1*, *independent_sampler=None*, *warn_independent_sampling=True*)

A Sampler using cma library as the backend.

Example

Optimize a simple quadratic function by using *PyCmaSampler*.

```
import optuna


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

(continues on next page)

(continued from previous page)

```
    return x**2 + y


sampler = optuna.integration.PyCmaSampler()
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=20)
```

Note that parallel execution of trials may affect the optimization performance of CMA-ES, especially if the number of trials running in parallel exceeds the population size.

Note: *CmaEsSampler* is deprecated and renamed to *PyCmaSampler* in v2.0.0. Please use *PyCmaSampler* instead of *CmaEsSampler*.

Parameters

- **x0** (*Dict[str, Any] | None*) – A dictionary of an initial parameter values for CMA-ES. By default, the mean of `low` and `high` for each distribution is used. Please refer to cma.CMAEvolutionStrategy for further details of `x0`.
- **sigma0** (*float | None*) – 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. If distribution is categorical, `min_range` is `len(choices) - 1`. Please refer to cma.CMAEvolutionStrategy for further details of `sigma0`.
- **cma_stds** (*Dict[str, float] | None*) – A dictionary of multipliers of sigma0 for each parameters. The default value is 1.0. Please refer to cma.CMAEvolutionStrategy for further details of `cma_stds`.
- **seed** (*int | None*) – A random seed for CMA-ES.
- **cma_opts** (*Dict[str, Any] | None*) – Options passed to the constructor of cma.CMAEvolutionStrategy class.

  Note that default option is cma_default_options, but `BoundaryHandler`, `bounds`, `CMA_stds` and `seed` arguments in `cma_opts` will be ignored because it is added by *PyCmaSampler* automatically.
- **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** (BaseSampler *| None*) – 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 *PyCmaSampler* 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.

Methods

after_trial(study, trial, state, values)	Trial post-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

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
float

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, float]

optuna.integration.CmaEsSampler

`class optuna.integration.CmaEsSampler`(*x0=None*, *sigma0=None*, *cma_stds=None*, *seed=None*, *cma_opts=None*, *n_startup_trials=1*, *independent_sampler=None*, *warn_independent_sampling=True*)

Wrapper class of PyCmaSampler for backward compatibility.

Warning: Deprecated in v2.0.0. 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/v2.0.0.

This class is renamed to `PyCmaSampler`.

Methods

`after_trial`(study, trial, state, values)	Trial post-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.

Parameters

- **x0** (*Dict[str, Any] | None*) –
- **sigma0** (*float | None*) –
- **cma_stds** (*Dict[str, float] | None*) –
- **seed** (*int | None*) –
- **cma_opts** (*Dict[str, Any] | None*) –
- **n_startup_trials** (*int*) –
- **independent_sampler** (BaseSampler *| None*) –
- **warn_independent_sampling** (*bool*) –

`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

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
float

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, float]

PyTorch

optuna.integration.PyTorchIgnitePruningHandler	PyTorch Ignite handler to prune unpromising trials.
optuna.integration.PyTorchLightningPruningCallback	PyTorch Lightning callback to prune unpromising trials.
optuna.integration.TorchDistributedTrial	A wrapper of *Trial* to incorporate Optuna with PyTorch distributed.

optuna.integration.PyTorchIgnitePruningHandler

`class optuna.integration.`**`PyTorchIgnitePruningHandler`**`(`*trial*, *metric*, *trainer*`)`

PyTorch Ignite handler to prune unpromising trials.

See the example if you want to add a pruning handler which observes validation accuracy.

Parameters

- **`trial`** (`Trial`) – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`metric`** (*`str`*) – A name of metric for pruning, e.g., `accuracy` and `loss`.
- **`trainer`** (*`Engine`*) – A trainer engine of PyTorch Ignite. Please refer to ignite.engine.Engine reference for further details.

optuna.integration.PyTorchLightningPruningCallback

`class optuna.integration.`**`PyTorchLightningPruningCallback`**`(`*trial*, *monitor*`)`

PyTorch Lightning callback to prune unpromising trials.

See the example if you want to add a pruning callback which observes accuracy.

Parameters

- **`trial`** (`Trial`) – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** (*`str`*) – An evaluation metric for pruning, e.g., `val_loss` or `val_acc`. The metrics are obtained from the returned dictionaries from e.g. `pytorch_lightning.LightningModule.training_step` or `pytorch_lightning.LightningModule.validation_epoch_end` and the names thus depend on how this dictionary is formatted.

Note: For the distributed data parallel training, the version of PyTorchLightning needs to be higher than or equal to v1.6.0. In addition, *`Study`* should be instantiated with RDB storage.

Note: If you would like to use PyTorchLightningPruningCallback in a distributed training environment, you need to evoke *PyTorchLightningPruningCallback.check_pruned()* manually so that *`TrialPruned`* is properly handled.

Methods

`check_pruned`()	Raise *`optuna.TrialPruned`* manually if pruned.
`on_fit_start`(trainer, pl_module)	
`on_validation_end`(trainer, pl_module)	

`check_pruned()`

Raise *`optuna.TrialPruned`* manually if pruned.

Currently, `intermediate_values` are not properly propagated between processes due to storage cache. Therefore, necessary information is kept in trial_system_attrs when the trial runs in a distributed situation.

Please call this method right after calling `pytorch_lightning.Trainer.fit()`. If a callback doesn't have any backend storage for DDP, this method does nothing.

Return type
None

optuna.integration.TorchDistributedTrial

class `optuna.integration.`**`TorchDistributedTrial`**(*trial*, *group=None*)

A wrapper of *`Trial`* to incorporate Optuna with PyTorch distributed.

See also:

`TorchDistributedTrial` provides the same interface as *`Trial`*. Please refer to *`optuna.trial.Trial`* for further details.

See the example if you want to optimize an objective function that trains neural network written with PyTorch distributed data parallel.

Parameters

- **`trial`** (`Trial | None`) – A *`Trial`* object or `None`. Please set trial object in rank-0 node and set `None` in the other rank node.
- **`group`** (`ProcessGroup | None`) – A *torch.distributed.ProcessGroup* to communicate with the other nodes. TorchDistributedTrial use CPU tensors to communicate, make sure the group supports CPU tensors communications.

  Use *gloo* backend when group is None. Create a global *gloo* backend when group is None and WORLD is nccl.

Note: The methods of *`TorchDistributedTrial`* are expected to be called by all workers at once. They invoke synchronous data transmission to share processing results and synchronize timing.

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

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

property system_attrs: Dict[str, Any]

> **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.

scikit-learn

optuna.integration.OptunaSearchCV	Hyperparameter search with cross-validation.

optuna.integration.OptunaSearchCV

class optuna.integration.**OptunaSearchCV**(*estimator*, *param_distributions*, ***, *cv=None*, *enable_pruning=False*, *error_score=nan*, *max_iter=1000*, *n_jobs=None*, *n_trials=10*, *random_state=None*, *refit=True*, *return_train_score=False*, *scoring=None*, *study=None*, *subsample=1.0*, *timeout=None*, *verbose=0*, *callbacks=None*)

Hyperparameter search with cross-validation.

Parameters

- **estimator** (*sklearn.base.BaseEstimator*) – Object to use to fit the data. This is assumed to implement the scikit-learn estimator interface. Either this needs to provide `score`, or `scoring` must be passed.
- **param_distributions** (*Mapping[str, BaseDistribution]*) – Dictionary where keys are parameters and values are distributions. Distributions are assumed to implement the optuna distribution interface.
- **cv** (*int | BaseCrossValidator | Iterable | None*) – Cross-validation strategy. Possible inputs for cv are:
  - None, to use the default 5-fold cross validation,
  - integer to specify the number of folds in a CV splitter,
  - CV splitter,
  - an iterable yielding (train, validation) splits as arrays of indices.

  For integer, if `estimator` is a classifier and `y` is either binary or multiclass, `sklearn.model_selection.StratifiedKFold` is used. otherwise, `sklearn.model_selection.KFold` is used.
- **enable_pruning** (*bool*) – If `True`, pruning is performed in the case where the underlying estimator supports `partial_fit`.

- **error_score** (*Number | float | str*) – Value to assign to the score if an error occurs in fitting. If 'raise', the error is raised. If numeric, `sklearn.exceptions.FitFailedWarning` is raised. This does not affect the refit step, which will always raise the error.
- **max_iter** (*int*) – Maximum number of epochs. This is only used if the underlying estimator supports `partial_fit`.
- **n_jobs** (*int | None*) – Number of `threading` based parallel jobs. `None` means `1`. `-1` means using 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.

- **n_trials** (*int*) – Number of trials. If `None`, there is no limitation on the number of trials. If `timeout` is also set to `None`, the study continues to create trials until it receives a termination signal such as Ctrl+C or SIGTERM. This trades off runtime vs quality of the solution.
- **random_state** (*int | RandomState | None*) – Seed of the pseudo random number generator. If int, this is the seed used by the random number generator. If `numpy.random.RandomState` object, this is the random number generator. If `None`, the global random state from `numpy.random` is used.
- **refit** (*bool*) – If `True`, refit the estimator with the best found hyperparameters. The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly.
- **return_train_score** (*bool*) – If `True`, training scores will be included. Computing training scores is used to get insights on how different hyperparameter settings impact the overfitting/underfitting trade-off. However computing training scores can be computationally expensive and is not strictly required to select the hyperparameters that yield the best generalization performance.
- **scoring** (*Callable[[...], float] | str | None*) – String or callable to evaluate the predictions on the validation data. If `None`, `score` on the estimator is used.
- **study** (`Study` *| None*) – Study corresponds to the optimization task. If `None`, a new study is created.
- **subsample** (*float | int*) – Proportion of samples that are used during hyperparameter search.
  - If int, then draw `subsample` samples.
  - If float, then draw `subsample` * `X.shape[0]` samples.
- **timeout** (*float | None*) – Time limit in seconds for the search of appropriate models. If `None`, the study is executed without time limitation. If `n_trials` is also set to `None`, the study continues to create trials until it receives a termination signal such as Ctrl+C or SIGTERM. This trades off runtime vs quality of the solution.
- **verbose** (*int*) – Verbosity level. The higher, the more messages.
- **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 *Callback for Study.optimize* for how to use and implement callback functions.

best_estimator_

Estimator that was chosen by the search. This is present only if `refit` is set to `True`.

n_splits_

Number of cross-validation splits.

refit_time_

Time for refitting the best estimator. This is present only if `refit` is set to `True`.

sample_indices_

Indices of samples that are used during hyperparameter search.

scorer_

Scorer function.

study_

Actual study.

Examples

```python
import optuna
from sklearn.datasets import load_iris
from sklearn.svm import SVC

clf = SVC(gamma="auto")
param_distributions = {
    "C": optuna.distributions.FloatDistribution(1e-10, 1e10, log=True)
}
optuna_search = optuna.integration.OptunaSearchCV(clf, param_distributions)
X, y = load_iris(return_X_y=True)
optuna_search.fit(X, y)
y_pred = optuna_search.predict(X)
```

Note: By following the scikit-learn convention for scorers, the direction of optimization is `maximize`. See https://scikit-learn.org/stable/modules/model_evaluation.html. For the minimization problem, please multiply `-1`.

Note: Added in v0.17.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v0.17.0.

Methods

fit(X[, y, groups])	Run fit with all sets of parameters.
get_params([deep])	Get parameters for this estimator.
score(X[, y])	Return the score on the given data.
set_params(**params)	Set the parameters of this estimator.

Attributes

*best_index_*	Trial number which corresponds to the best candidate parameter setting.
*best_params_*	Parameters of the best trial in the *Study*.
*best_score_*	Mean cross-validated score of the best estimator.
*best_trial_*	Best trial in the *Study*.
*classes_*	Class labels.
decision_function	Call `decision_function` on the best estimator.
inverse_transform	Call `inverse_transform` on the best estimator.
*n_trials_*	Actual number of trials.
predict	Call `predict` on the best estimator.
predict_log_proba	Call `predict_log_proba` on the best estimator.
predict_proba	Call `predict_proba` on the best estimator.
score_samples	Call `score_samples` on the best estimator.
set_user_attr	Call `set_user_attr` on the *Study*.
transform	Call `transform` on the best estimator.
*trials_*	All trials in the *Study*.
trials_dataframe	Call `trials_dataframe` on the *Study*.
*user_attrs_*	User attributes in the *Study*.

property best_index_: int

Trial number which corresponds to the best candidate parameter setting.

Retuned value is equivant to `optuna_search.best_trial_.number`.

property best_params_: Dict[str, Any]

Parameters of the best trial in the *Study*.

property best_score_: float

Mean cross-validated score of the best estimator.

property best_trial_: *FrozenTrial*

Best trial in the *Study*.

property classes_: List[float] | ndarray | Series

Class labels.

property decision_function: Callable[[...], List[float] | ndarray | Series | List[List[float]] | DataFrame | spmatrix]

Call `decision_function` on the best estimator.

This is available only if the underlying estimator supports `decision_function` and `refit` is set to `True`.

`fit`(*X*, *y=None*, *groups=None*, ***fit_params*)

Run fit with all sets of parameters.

Parameters

- **X** (*List[List[float]] | ndarray | DataFrame | spmatrix*) – Training data.
- **y** (*List[float] | ndarray | Series | List[List[float]] | DataFrame | spmatrix | None*) – Target variable.
- **groups** (*List[float] | ndarray | Series | None*) – Group labels for the samples used while splitting the dataset into train/validation set.
- ****fit_params** (*Any*) – Parameters passed to `fit` on the estimator.

Returns
self.

Return type
OptunaSearchCV

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

property inverse_transform: Callable[[...], List[List[float]] | ndarray | DataFrame | spmatrix]

Call `inverse_transform` on the best estimator.

This is available only if the underlying estimator supports `inverse_transform` and `refit` is set to `True`.

property n_trials_: int

Actual number of trials.

property predict: Callable[[...], List[float] | ndarray | Series | List[List[float]] | DataFrame | spmatrix]

Call `predict` on the best estimator.

This is available only if the underlying estimator supports `predict` and `refit` is set to `True`.

property predict_log_proba: Callable[[...], List[List[float]] | ndarray | DataFrame | spmatrix]

Call `predict_log_proba` on the best estimator.

This is available only if the underlying estimator supports `predict_log_proba` and `refit` is set to `True`.

property predict_proba: Callable[[...], List[List[float]] | ndarray | DataFrame | spmatrix]

Call `predict_proba` on the best estimator.

This is available only if the underlying estimator supports `predict_proba` and `refit` is set to `True`.

score(*X*, *y=None*)

Return the score on the given data.

Parameters

- **X** (*List[List[float]] | ndarray | DataFrame | spmatrix*) – Data.
- **y** (*List[float] | ndarray | Series | List[List[float]] | DataFrame | spmatrix | None*) – Target variable.

Returns
Scaler score.

Return type
float

property score_samples: Callable[[...], List[float] | ndarray | Series]

Call `score_samples` on the best estimator.

This is available only if the underlying estimator supports `score_samples` and `refit` is set to `True`.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

property set_user_attr: Callable[[...], None]

Call `set_user_attr` on the *Study*.

property transform: Callable[[...], List[List[float]] | ndarray | DataFrame | spmatrix]

Call `transform` on the best estimator.

This is available only if the underlying estimator supports `transform` and `refit` is set to `True`.

property trials_: List[FrozenTrial]

All trials in the *Study*.

property trials_dataframe: Callable[[...], DataFrame]

Call `trials_dataframe` on the *Study*.

property user_attrs_: Dict[str, Any]

User attributes in the *Study*.

scikit-optimize

optuna.integration.SkoptSampler	Sampler using Scikit-Optimize as the backend.

optuna.integration.SkoptSampler

class optuna.integration.**SkoptSampler**(*independent_sampler=None*, *warn_independent_sampling=True*, *skopt_kwargs=None*, *n_startup_trials=1*, ***, *consider_pruned_trials=False*, *seed=None*)

Sampler using Scikit-Optimize as the backend.

Example

Optimize a simple quadratic function by using *SkoptSampler*.

```
import optuna


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


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

Parameters

- **independent_sampler** (BaseSampler | *None*) – 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 *SkoptSampler* 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.

- **skopt_kwargs** (*Dict[str, Any] | None*) – Keyword arguments passed to the constructor of skopt.Optimizer class.

  Note that dimensions argument in skopt_kwargs will be ignored because it is added by *SkoptSampler* automatically.

- **n_startup_trials** (*int*) – The independent sampling is used until the given number of trials finish in the same study.
- **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: As the number of trials n increases, each sampling takes longer and longer on a scale of $O(n^3)$. And, if this is `True`, the number of trials will increase. So, it is suggested to set this flag `False` when each evaluation of the objective function is relatively faster than each sampling. On the other hand, it is suggested to set this flag `True` when each evaluation of the objective function is relatively slower than each sampling.

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

Methods

after_trial(study, trial, state, values)	Trial post-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

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*]

SHAP

optuna.integration.ShapleyImportanceEvaluator	Shapley (SHAP) parameter importance evaluator.

optuna.integration.ShapleyImportanceEvaluator

class optuna.integration.**ShapleyImportanceEvaluator**(*, *n_trees=64*, *max_depth=64*, *seed=None*)

Shapley (SHAP) parameter importance evaluator.

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

Note: This evaluator requires the sklearn Python package and SHAP. The model for the SHAP calculation is based on sklearn.ensemble.RandomForestClassifier.

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.

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

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

An collections.OrderedDict where the keys are parameter names and the values are assessed importances.

Return type

Dict[str, float]

skorch

optuna.integration.SkorchPruningCallback	Skorch callback to prune unpromising trials.

optuna.integration.SkorchPruningCallback

`class optuna.integration.SkorchPruningCallback(`*trial*, *monitor*`)`

Skorch callback to prune unpromising trials.

New in version 2.1.0.

Parameters

- **trial** (`Trial`) – A *Trial* corresponding to the current evaluation of the objective function.
- **monitor** (*str*) – An evaluation metric for pruning, e.g. `val_loss` or `val_acc`. The metrics are obtained from the returned dictionaries, i.e., `net.histroy`. The names thus depend on how this dictionary is formatted.

Methods

on_epoch_end(net, **kwargs)

TensorFlow

optuna.integration.TensorBoardCallback	Callback to track Optuna trials with TensorBoard.
optuna.integration.TensorFlowPruningHook	TensorFlow SessionRunHook to prune unpromising trials.
optuna.integration.TFKerasPruningCallback	tf.keras callback to prune unpromising trials.

optuna.integration.TensorBoardCallback

`class optuna.integration.TensorBoardCallback(`*dirname*, *metric_name*`)`

Callback to track Optuna trials with TensorBoard.

This callback adds relevant information that is tracked by Optuna to TensorBoard.

See the example.

Parameters

- **dirname** (*str*) – Directory to store TensorBoard logs.
- **metric_name** (*str*) – Name of the metric. Since the metric itself is just a number, *metric_name* can be used to give it a name. So you know later if it was roc-auc or accuracy.

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.

optuna.integration.TensorFlowPruningHook

`class optuna.integration.`**`TensorFlowPruningHook`**`(`*trial*, *estimator*, *metric*, *run_every_steps*`)`

TensorFlow SessionRunHook to prune unpromising trials.

See the example if you want to add a pruning hook to TensorFlow's estimator.

Parameters

- **`trial`** (`Trial`) – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`estimator`** (*`tf.estimator.Estimator`*) – An estimator which you will use.
- **`metric`** (*`str`*) – An evaluation metric for pruning, e.g., `accuracy` and `loss`.
- **`run_every_steps`** (*`int`*) – An interval to watch the summary file.

Methods

`after_run`(run_context, run_values)
`before_run`(run_context)
`begin`()

optuna.integration.TFKerasPruningCallback

`class optuna.integration.`**`TFKerasPruningCallback`**`(`*trial*, *monitor*`)`

tf.keras callback to prune unpromising trials.

This callback is intend to be compatible for TensorFlow v1 and v2, but only tested with TensorFlow v2.

See the example if you want to add a pruning callback which observes the validation accuracy.

Parameters

- **`trial`** (`Trial`) – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** (*`str`*) – An evaluation metric for pruning, e.g., `val_loss` or `val_acc`.

Methods

`on_epoch_end`(epoch[, logs])

XGBoost

optuna.integration.XGBoostPruningCallback	Callback for XGBoost to prune unpromising trials.

optuna.integration.XGBoostPruningCallback

class optuna.integration.**XGBoostPruningCallback**(*trial*, *observation_key*)

Callback for XGBoost to prune unpromising trials.

See the example if you want to add a pruning callback which observes validation accuracy of a XGBoost model.

Parameters

- **trial** (`Trial`) – A *Trial* corresponding to the current evaluation of the objective function.
- **observation_key** (*str*) – An evaluation metric for pruning, e.g., `validation-error` and `validation-merror`. When using the Scikit-Learn API, the index number of `eval_set` must be included in the `observation_key`, e.g., `validation_0-error` and `validation_0-merror`. Please refer to `eval_metric` in XGBoost reference for further details.

6.3.7 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
```

(continues on next page)

(continued from previous page)

```
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`.

```
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`.

```
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.

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.

```
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

6.3.8 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.

See also:

3. Efficient Optimization Algorithms 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.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.

```
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

```
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
```

(continues on next page)

(continued from previous page)

```
    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

```
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.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 modifying 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):
```

(continues on next page)

(continued from previous page)

```
        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`.

```
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():
```

(continues on next page)

(continued from previous page)

```
            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 $\mathsf{min\_resource} \times \mathsf{reduction\_factor}^{\mathsf{min\_early\_stopping\_rate}}$ 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 $\frac{1}{\mathsf{reduction\_factor}}$ 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., $\mathsf{min\_resource} \times \mathsf{reduction\_factor}^{(\mathsf{min\_early\_stopping\_rate}+\mathsf{rung})}$ 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 $\frac{1}{\mathsf{reduction\_factor}}$ 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 $\mathrm{The\ number\ of\ brackets} = \mathrm{floor}(\log_{\texttt{reduction\_factor}}(\frac{\texttt{max\_resource}}{\texttt{min\_resource}})) + 1$. 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.

```
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)


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

6.3.9 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:

3. Efficient Optimization Algorithms 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.

	RandomSampler	GridSampler	TPESampler	CmaEsSampler	NSGAIISampler	QMCSampler	BoTorchSampler	BruteForceSampler
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) (*)	$O(d)$	$O(dn)$	$O(dn$ lo	$O(d^3)$	$O(mp^2)$ (***)	$O(dn)$	$O(n^3)$	$O(d)$
Recommended budgets (#trials) (**)	as many as one likes	number of combinations	100 – 1000	1000 – 10000	100 – 10000	as many as one likes	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.

(**): The budget depends on the number of parameters and the number of objectives.

(***): 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: For float, integer, or categorical parameters, see *2. Pythonic Search Space* tutorial.

For pruning, see *3. Efficient Optimization Algorithms* 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 *2. Pythonic Search Space* tutorial and `TPESampler`. The `group` option of `TPESampler` allows `TPESampler` to handle the conditional search space.

For multi-objective optimization, see *Multi-objective Optimization with Optuna* 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 *4. Easy Parallelization* 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.PartialFixedSampler`	Sampler with partially fixed parameters.
`optuna.samplers.NSGAIISampler`	Multi-objective sampler using the NSGA-II algorithm.
`optuna.samplers.MOTPESampler`	Multi-objective sampler using the MOTPE 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.

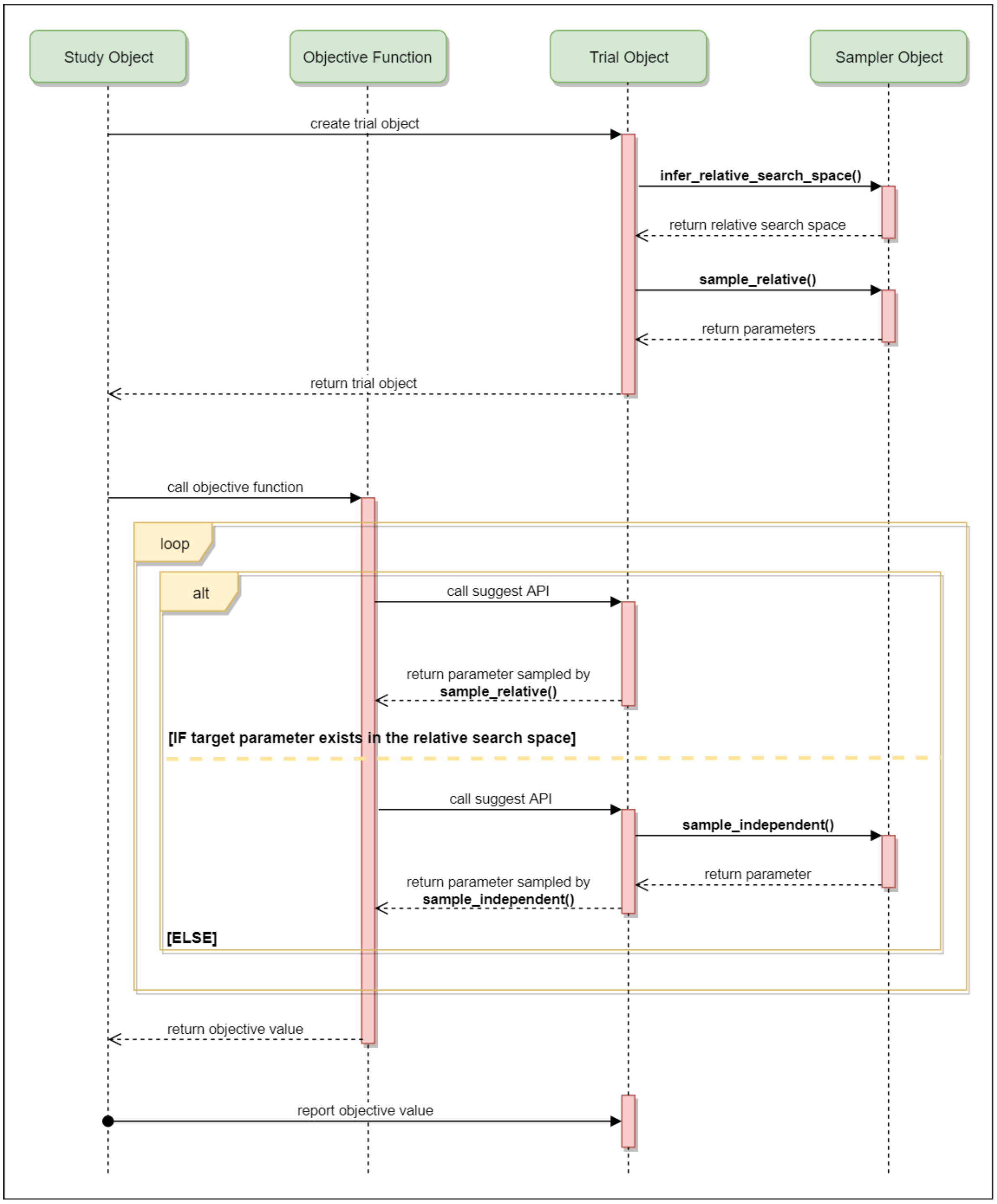
Optuna combines two types of sampling strategies, which are called *relative sampling* and *independent sampling*.

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.


Study Object
Objective Function
Trial Object
Sampler Object
create trial object
infer_relative_search_space()
return relative search space
sample_relative()
return parameters
return trial object
call objective function
loop
alt
call suggest API
return parameter sampled by sample_relative()
[IF target parameter exists in the relative search space]
call suggest API
sample_independent()
return parameter
return parameter sampled by sample_independent()
[ELSE]
return objective value
report objective value

Methods

after_trial(study, trial, state, values)	Trial post-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

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: *GridSampler* automatically stops the optimization if all combinations in the passed `search_space` have already been evaluated, internally invoking the *stop()* method.

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.

```
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.
`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

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

```
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.
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

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*)

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
- Multiobjective tree-structured parzen estimator for computationally expensive optimization problems
- Multiobjective Tree-Structured Parzen Estimator

Example

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


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


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

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], 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** (*int | None*) – 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
```

(continues on next page)

(continued from previous page)

```
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 algorithm 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: This feature can be used for only single-objective optimization; this argument is ignored for multi-objective optimization.

  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** (*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.

  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.
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

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.

```
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, 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.
- 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.

See also:

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

Parameters

- **x0** (*Dict[str, Any] | None*) – 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** (*float* | *None*) – 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** (*int* | *None*) – 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** (BaseSampler | *None*) – 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** (*str* | *None*) – Strategy for restarting CMA-ES optimization when converges to a local minimum. If given `None`, CMA-ES will not restart (default). If given 'ipop', CMA-ES will restart with increasing population size. 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** (*int* | *None*) – A population size of CMA-ES. When set `restart_strategy = 'ipop'`, 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 setting `restart_strategy = 'ipop'`.
- **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.

- **source_trials** (*List[FrozenTrial] | None*) – 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.
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

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.

New in version 2.4.0.

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
```

(continues on next page)

(continued from previous page)

```
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.
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

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*)

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.

Methods

`after_trial`(study, trial, state, values)	Trial post-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

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.MOTPESampler

class optuna.samplers.**MOTPESampler**(**, consider_prior=True, prior_weight=1.0, consider_magic_clip=True, consider_endpoints=True, n_startup_trials=10, n_ehvi_candidates=24, gamma=<function default_gamma>, weights_above=<function _default_weights_above>, seed=None*)

Multi-objective sampler using the MOTPE algorithm.

This sampler is a multiobjective version of *TPESampler*.

For further information about MOTPE algorithm, please refer to the following paper:

- Multiobjective tree-structured parzen estimator for computationally expensive optimization problems
- Multiobjective Tree-Structured Parzen Estimator

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 MOTPE algorithm until the given number of trials finish in the same study. 11 * number of variables - 1 is recommended in the original paper.
- **n_ehvi_candidates** (*int*) – Number of candidate samples used to calculate the expected hypervolume 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_above** (*Callable[[int], ndarray]*) – A function that takes the number of finished trials and returns a weight for them. As default, weights are automatically calculated by the MOTPE's default strategy.
- **seed** (*int | None*) – Seed for random number generator.

Note: Initialization with Latin hypercube sampling may improve optimization performance. However, the current implementation only supports initialization with random sampling.

Example

```python
import optuna

seed = 128
num_variables = 2
n_startup_trials = 11 * num_variables - 1


def objective(trial):
    x = []
    for i in range(1, num_variables + 1):
        x.append(trial.suggest_float(f"x{i}", 0.0, 2.0 * i))
    return x
```

(continues on next page)

(continued from previous page)

```
sampler = optuna.samplers.MOTPESampler(
    n_startup_trials=n_startup_trials, n_ehvi_candidates=24, seed=seed
)
study = optuna.create_study(directions=["minimize"] * num_variables,␣
↪sampler=sampler)
study.optimize(objective, n_trials=n_startup_trials + 10)
```

Warning: Deprecated in v2.9.0. 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/v2.9.0.

Methods

after_trial(study, trial, state, values)	Trial post-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

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.

```
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.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.
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

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

```
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.
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

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 an `collections.OrderedDict` 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 an `collections.OrderedDict` 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

6.3.10 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[, 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 an `collections.OrderedDict` sorted by keys, i.e. parameter names.

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

Return type
Dict[str, *BaseDistribution*]

optuna.search_space.intersection_search_space

optuna.search_space.**intersection_search_space**(*trials*, *ordered_dict=False*, *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*[optuna.trial.FrozenTrial]) – A list of 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 an collections.OrderedDict 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]

6.3.11 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.

```
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** (*Dict[str, Any] | None*) – 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** (*int | None*) – 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** (*int | None*) – 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** (*Callable[[optuna.study.Study, FrozenTrial], None] | 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.

continues on next page

Table 1 – continued from previous page

`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, 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

`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:

```
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*) – 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*]

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

6.3.12 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_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.
system_attrs	Return system attributes.
trials	Return all trials in the study.
user_attrs	Return user attributes.

Parameters

- **study_name** (*str*) –
- **storage** (*str | BaseStorage*) –
- **sampler** (samplers.BaseSampler | *None*) –
- **pruner** (BasePruner | *None*) –

add_trial(*trial*)

Add trial to study.

The trial is validated before being added.

Example

```
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
```

(continues on next page)

(continued from previous page)

```
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 *Second scenario: Have Optuna utilize already evaluated hyperparameters* 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

```
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 Interface* tutorial provides use-cases with examples.

Example

Getting the trial object with the *ask()* method.

```
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.

```
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`.
```

(continues on next page)

(continued from previous page)

```
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 *Re-use the 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

```
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 *First Scenario: Have Optuna evaluate your hyperparameters* 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

```
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]

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

```
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** (*None | float*) – 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 *Callback for Study.optimize* 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` not is `None`, and `n_jobs` $\neq 1$.

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"])
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 *User Attributes*.

Example

```
import optuna


def objective(trial):
    x = trial.suggest_float("x", 0, 1)
    y = trial.suggest_float("y", 0, 1)
    return x**2 + y**2
```

(continues on next page)

(continued from previous page)

```
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

```
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 Interface* tutorial provides use-cases with examples.

Example

```
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.
            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

```
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)
```

(continues on next page)

(continued from previous page)

```
# 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

```
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"])
```

(continues on next page)

(continued from previous page)

```
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

```
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 | 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** (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** (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 *Saving/Resuming Study with RDB Backend* 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

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

```
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

```
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_summaries

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

Get all history of studies stored in a specified storage.

Example

```
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*)

Direction of a *Study*.

NOT_SET

Direction has not been set.

MINIMIZE

Study minimizes the objective function.

MAXIMIZE

Study maximizes the objective function.

Attributes

NOT_SET
MINIMIZE
MAXIMIZE

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** (FrozenTrial | *None*) –
- **user_attrs** (*Dict[str, Any]*) –
- **system_attrs** (*Dict[str, Any]*) –
- **n_trials** (*int*) –
- **datetime_start** (*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

6.3.13 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.

```
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)


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 *User 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)
```

(continues on next page)

(continued from previous page)

```
    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)


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:

2. Pythonic Search Space 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 $[\mathsf{low}, \mathsf{high}]$, and the step of discretization is q. More specifically, this method returns one of the values in the sequence $\mathsf{low}, \mathsf{low} + q, \mathsf{low} + 2q, \ldots, \mathsf{low} + kq \leq \mathsf{high}$, 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.

New in version 1.3.0.

Example

Suggest a momentum, learning rate and scaling factor of learning rate for neural network training.

```
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)
```

(continues on next page)

(continued from previous page)

```
    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** (*None | float*) – 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:

2. Pythonic Search Space 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 $[\mathsf{low}, \mathsf{high}]$.

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 $\mathsf{low}, \mathsf{low} + \mathsf{step}, \mathsf{low} + 2 * \mathsf{step}, \dots, \mathsf{low} + k * \mathsf{step} \le \mathsf{high}$, where k denotes an integer.

  Note: The `step != 1` and `log` arguments cannot be used at the same time. To set the `step` argument $\mathsf{step} \ge 2$, 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:

2. Pythonic Search Space 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 $[\mathsf{low}, \mathsf{high})$ in the log domain. When $\mathsf{low} = \mathsf{high}$, 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 $[\mathsf{low}, \mathsf{high})$ in the linear domain. When $\mathsf{low} = \mathsf{high}$, 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.

```
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.

```
import copy
import datetime

import optuna


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

    # 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*)

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.

Attributes

RUNNING
COMPLETE
PRUNED
FAIL
WAITING

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

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

```
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** (*None | float*) – 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

6.3.14 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_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 objective value.
optuna.visualization.plot_slice	Plot the parameter relationship as slice plot in a study.
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_objs.Figure` object.

Return type

go.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.

```
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


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_objs.Figure object.

Return type
go.Figure

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
```

(continues on next page)

(continued from previous page)

```
    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_objs.Figure` object.

Return type
: go.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.

```
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_objs.Figure` object.

Return type
go.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_objs.Figure` object.

Return type
go.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.

```
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*.

- **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.

  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.

- **target_name** (*str*) – Target's name to display on the axis label.

Returns
A plotly.graph_objs.Figure object.

Return type
go.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 Optimization with Optuna* 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()
```

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.

  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_objs.Figure` object.

Return type
go.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 objective value.

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 a rank plot.

```
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_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_objs.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.

```
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_objs.Figure` object.

Return type
go.Figure

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_objs.Figure object.

Return type

go.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_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_slice`	Plot the parameter relationship as slice plot in a study with Matplotlib.
`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()*.

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)

optuna.visualization.matplotlib.plot_contour(study, params=["x", "y"])
```

Parameters

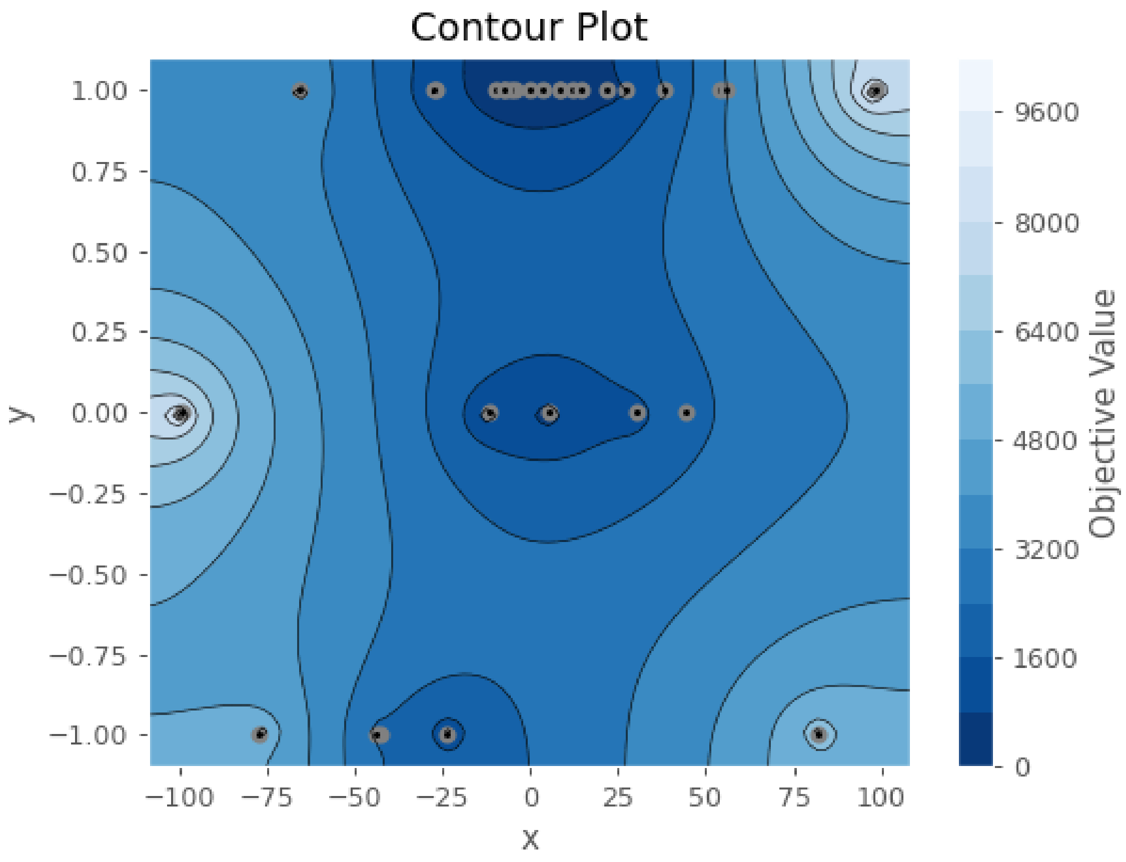
- **study** (Study) – A *Study* object whose trials are plotted for their target values.
- **params** (*list[str] | None*) – Parameter list to visualize. The default is all parameters.
- **target** (*Callable[[FrozenTrial], float] | None*) – A function to specify the value to display. If it is None and 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


Contour Plot
y
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
x
−100
−75
−50
−25
0
25
50
75
100
Objective Value
9600
8000
6400
4800
3200
1600
0

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

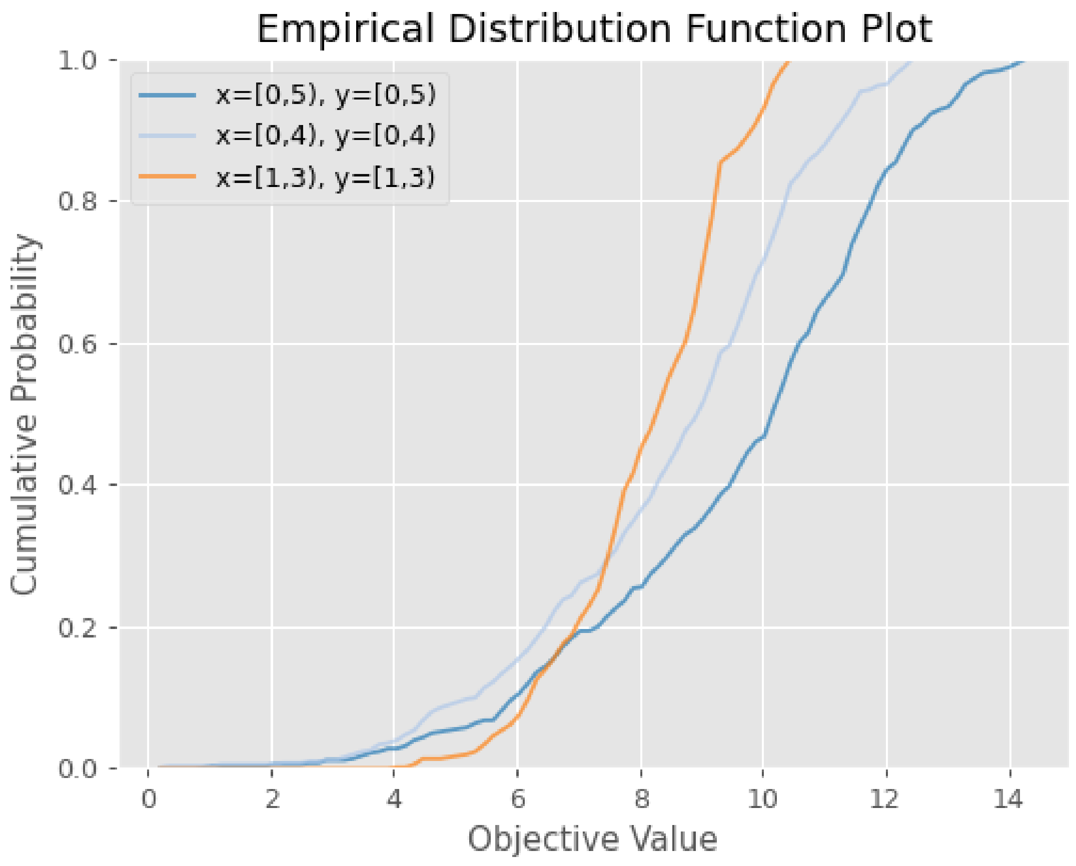
The following code snippet shows how to plot EDF.

```
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


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)
```

(continues on next page)

(continued from previous page)

```
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_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

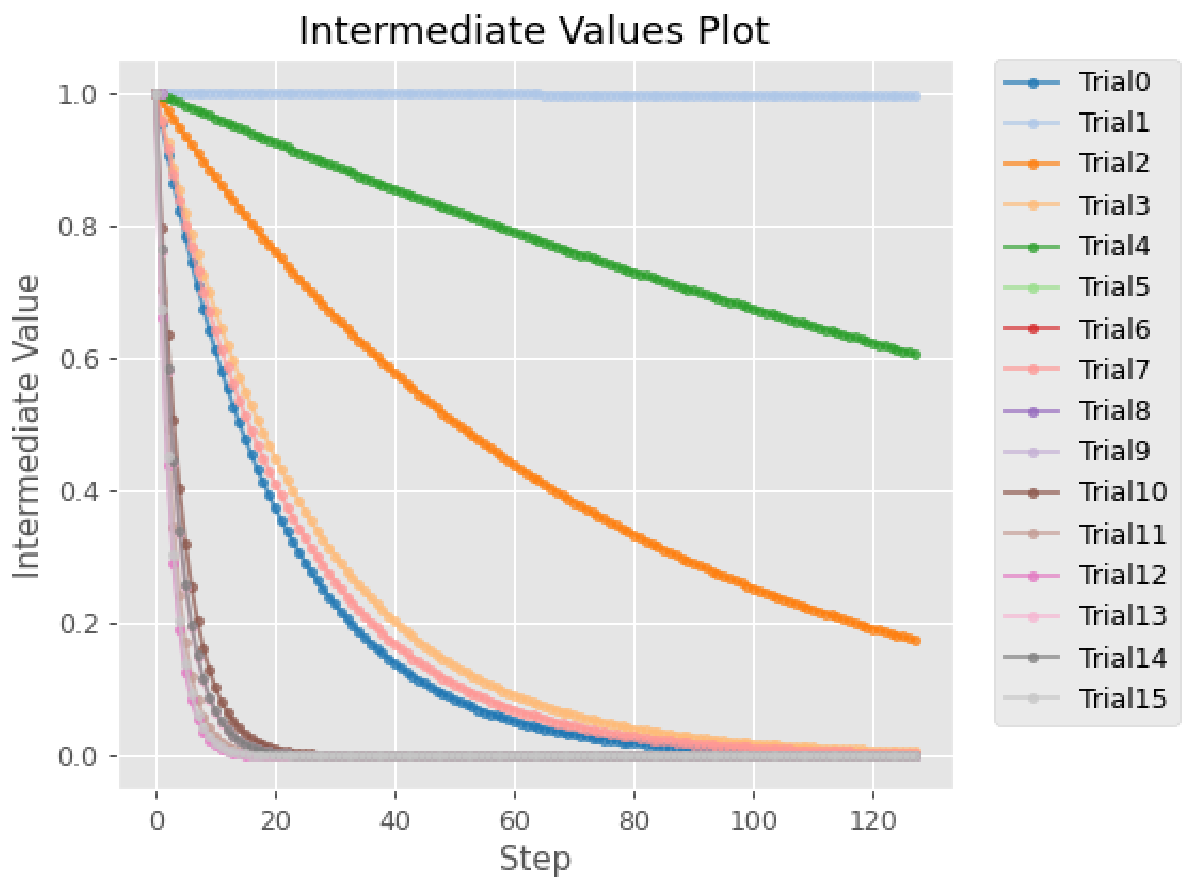
The following code snippet shows how to plot intermediate values.

```
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)
```

(continues on next page)

(continued from previous page)

```
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.

```
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

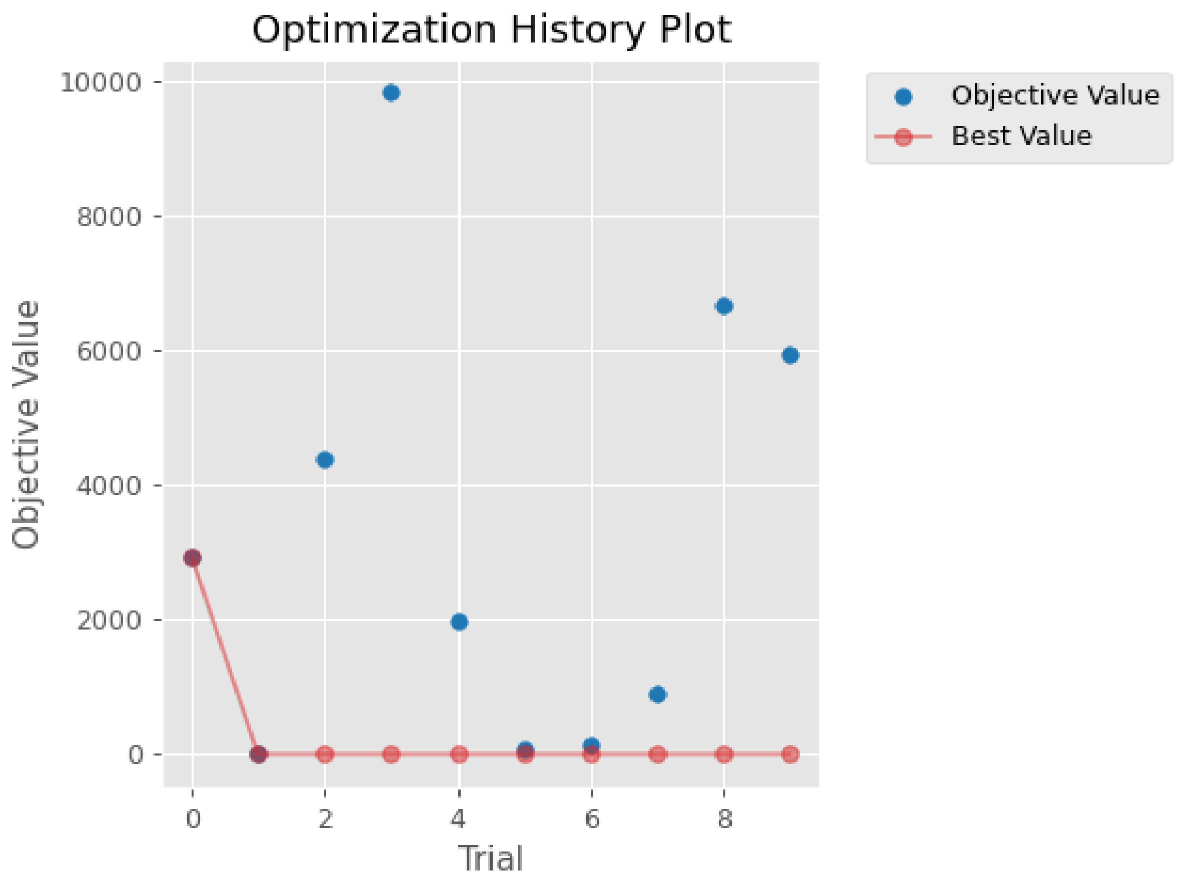
- **`study`** (`Study` | *`Sequence[Study]`*) – A *`Study`* object whose trials are plotted for their target values. You can pass multiple studies if you want to compare those optimization histories.
- **`target`** (*`Callable[[FrozenTrial], float] | None`*) – A function to specify the value to display. 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.


Optimization History Plot
Objective Value
Best Value
Objective Value
Trial
0
2000
4000
6000
8000
10000
0
2
4
6
8

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.

```
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"])
```

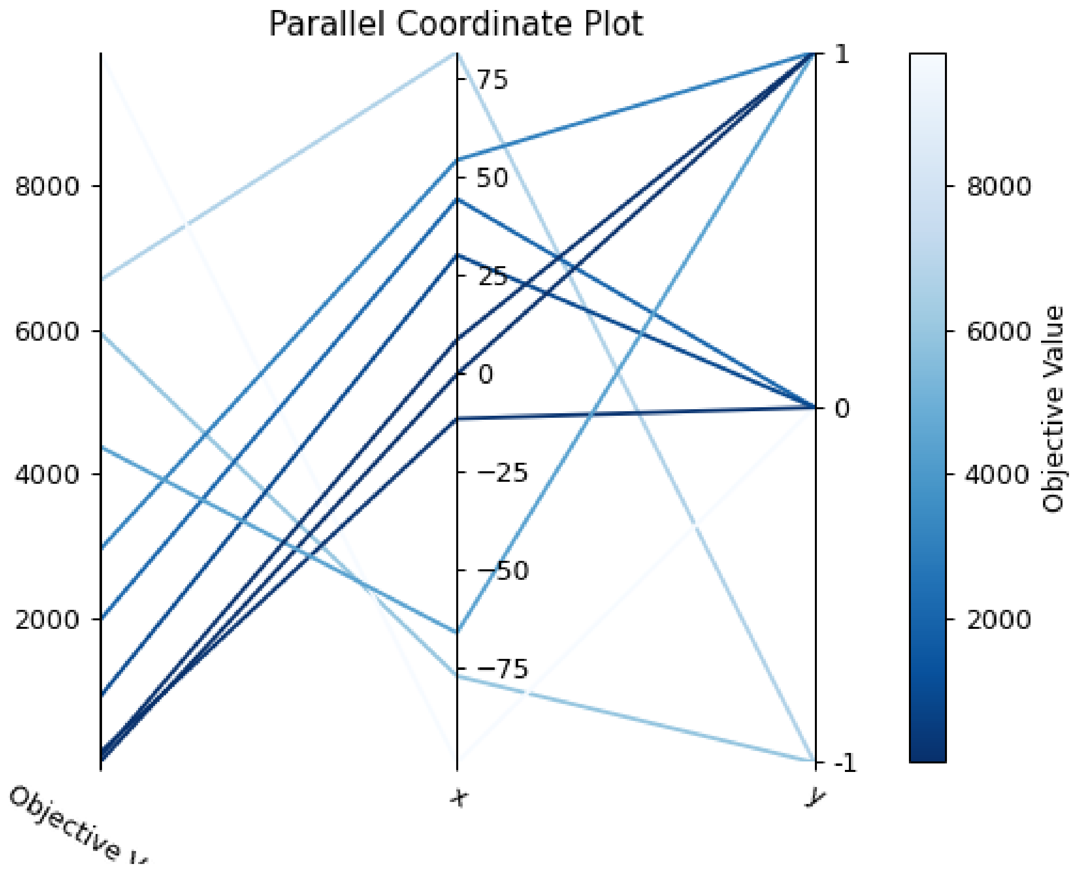
Parameters

- **study** (Study) – A *Study* object whose trials are plotted for their target values.
- **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.


Parallel Coordinate Plot
8000
6000
4000
2000
75
50
25
0
−25
−50
−75
1
0
-1
8000
6000
4000
2000
Objective Value
Objective V
x
y

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.

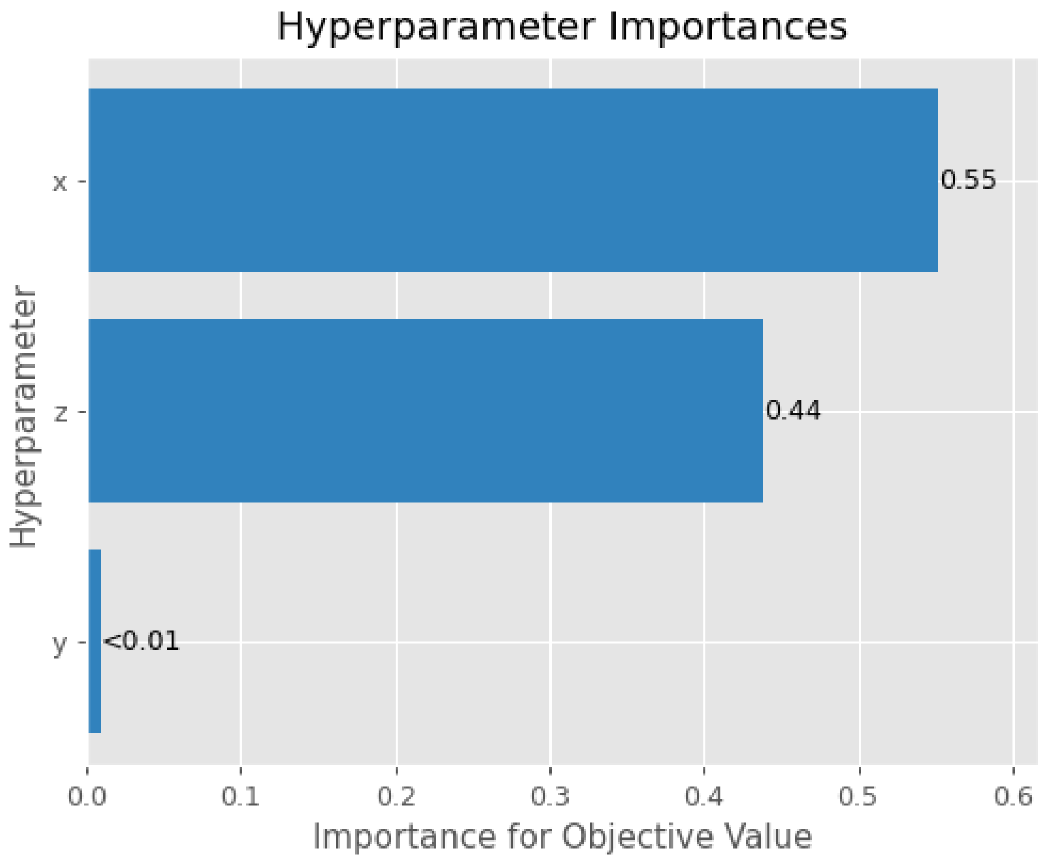
Example

The following code snippet shows how to plot hyperparameter importances.

```
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)

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.


Hyperparameter Importances
x
z
y
0.55
0.44
<0.01
Hyperparameter
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Importance for Objective Value

- **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. For example, to get 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.

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.

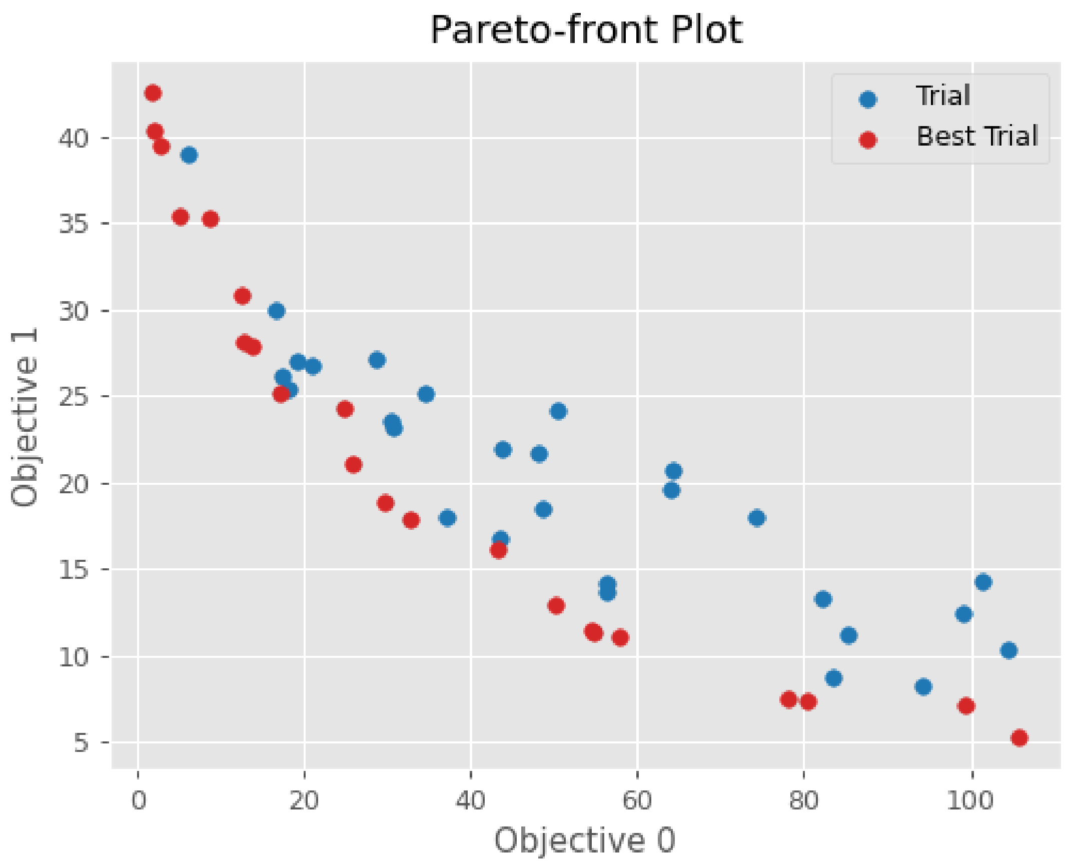
See also:

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)
```


Pareto-front Plot
Trial
Best Trial
Objective 1
Objective 0
40
35
30
25
20
15
10
5
0
20
40
60
80
100

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 `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 `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_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.

```
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.

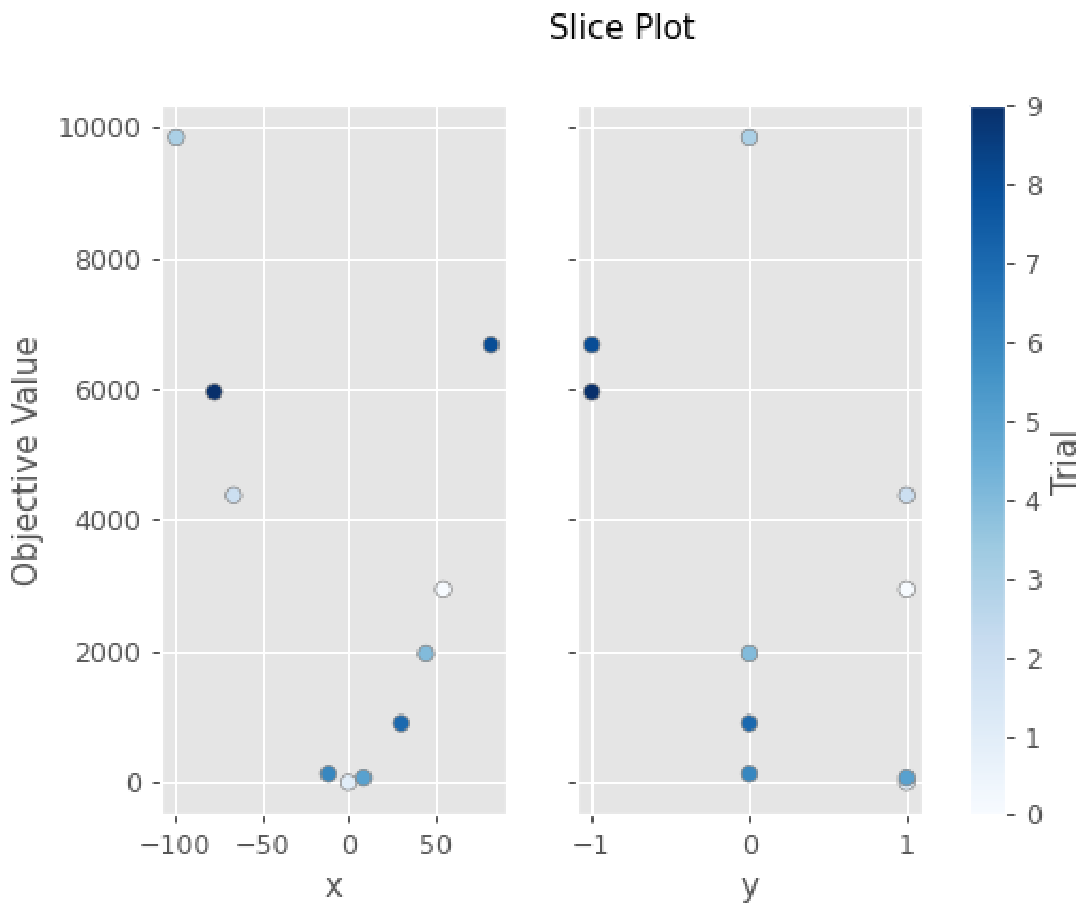
Returns

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.


Slice Plot
Objective Value
10000
8000
6000
4000
2000
0
−100
−50
0
50
x
−1
0
1
y
Trial
9
8
7
6
5
4
3
2
1
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 *5. Quick Visualization for Hyperparameter Optimization Analysis* tutorial provides use-cases with examples.

6.4 FAQ

6.4.1 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.

6.4.2 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)
    return (x - 2) ** 2
```

(continues on next page)

(continued from previous page)

```
# 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_addtitional_args.py example, which reuses the dataset instead of loading it in each trial execution.

6.4.3 Can I use Optuna without remote RDB servers?

Yes, it's possible.

In the simplest form, Optuna works with in-memory storage:

```
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:

```
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 *Saving/Resuming Study with RDB Backend* for more details.

6.4.4 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`:

```
study = optuna.create_study()
joblib.dump(study, "study.pkl")
```

And to resume the study:

```
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 *Saving/Resuming Study with RDB Backend* for more details.

6.4.5 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.

6.4.6 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:

```
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)))
```

6.4.7 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:

```
sampler = TPESampler(seed=10)  # Make the sampler behave in a deterministic way.
study = optuna.create_study(sampler=sampler)
study.optimize(objective)
```

To make the pruning by *HyperbandPruner* reproducible, you can specify `study_name` of *Study* and hash seed.

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.

6.4.8 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()
```

number	state	value	…	param	system_attrs
0	TrialState.FAIL		…	0	Setting status of trial#0 as TrialState.FAIL because of the following error: ValueError('A test error in objective.')
1	TrialState.COMPLE	1269	…	1	

See also:

The `catch` argument in *optimize()*.

6.4.9 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.
```

6.4.10 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

6.4.11 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.

6.4.12 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:

```
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:

```
# 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}))
```

6.4.13 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.

6.4.14 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)

(continued from previous 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.

6.4.15 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)
```

(continues on next page)

(continued from previous page)

```
        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.

6.4.16 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.

```
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"],
```

(continues on next page)

(continued from previous page)

```
    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.

6.4.17 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.

6.4.18 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.

```python
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.

6.4.19 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)
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX