Optuna Documentation

Release 2.8.0.dev0

Optuna Contributors.

Apr 05, 2021

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 or 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

```
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.load_boston(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 Issues for bug reports, feature requests and questions.
- Gitter for interactive chat with developers.
- Stack Overflow for questions.

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).

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 3.6 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.

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))
```

Out:

```
Found x: 2.01204497628707, (x - 2)^2: 0.00014508145375607373
```

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
```

Out:

```
{'x': 2.01204497628707}
```

To get the best observed value of the objective function:

```
study.best_value
```

Out:

```
0.00014508145375607373
```

To get the best trial:

```
study.best_trial
```

Out:

```
FrozenTrial(number=81, values=[0.00014508145375607373], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 19, 616846), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 19, 619989), params={'x': 2.01204497628707}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=81, state=TrialState.COMPLETE, value=None)
```

To get all trials:

```
study.trials
```

Out:

```
[FrozenTrial(number=0, values=[69.34000849259169], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 19, 372801), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 19, 373148), params={'x': -6.327064818565525}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=0, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=1, values=[38.54750055245735], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 19, 373469), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 19, 373782), params={'x': -4.208663346683998}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=1, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=2, values=[22.35552357251056], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 19, 374036), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 19, 374310), params={'x': 6.728162811548536}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=2, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=3, values=[21.736594736998725], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 19, 374558), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 19, 374826), params={'x': -2.662252109978473}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=3, state=TrialState.COMPLETE, value=None),␣
```

(continues on next page)

(continued from previous page)

```
```

To get the number of trials:

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

Out:

```
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)
```

Out:

```
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))
```

Out:

```
Found x: 1.9968718971934352, (x - 2)^2: 9.785027168438806e-06
```

Total running time of the script: (0 minutes 0.709 seconds)

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"])

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

(continues on next page)

(continued from previous page)

```
    # 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:

```
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:

```
import torch
import torch.nn as nn


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

    layers = []
    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))
```

(continues on next page)

(continued from previous page)

```
    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)

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 `optuna.samplers.BaseSampler`.

Optuna provides the following sampling algorithms:

- Tree-structured Parzen Estimator algorithm implemented in `optuna.samplers.TPESampler`
- CMA-ES based algorithm implemented in `optuna.samplers.CmaEsSampler`
- Grid Search implemented in `optuna.samplers.GridSampler`
- Random Search implemented in `optuna.samplers.RandomSampler`

The default sampler is `optuna.samplers.TPESampler`.

Switching Samplers

```
import optuna
```

By default, Optuna uses `TPESampler` as follows.

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

Out:

```
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__}")
```

Out:

```
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:

- Asynchronous Successive Halving algorithm implemted in *`optuna.pruners.SuccessiveHalvingPruner`*
- Hyperband algorithm implemented in *`optuna.pruners.HyperbandPruner`*
- Median pruning algorithm implemented in *`optuna.pruners.MedianPruner`*
- Threshold pruning algorithm implemented in *`optuna.pruners.ThresholdPruner`*

We use *`optuna.pruners.MedianPruner`* in most examples, though basically it is outperformed by *`optuna.pruners.SuccessiveHalvingPruner`* and *`optuna.pruners.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 *`optuna.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)
```

(continues on next page)

(continued from previous page)

```
    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)
```

Out:

```
A new study created in memory with name: no-name-c34d0e34-14ad-4131-b518-8b4a3736f89f
Trial 0 finished with value: 0.23684210526315785 and parameters: {'alpha': 0.
↪02760600748111276}. Best is trial 0 with value: 0.23684210526315785.
Trial 1 finished with value: 0.07894736842105265 and parameters: {'alpha': 3.
↪764350743416773e-05}. Best is trial 1 with value: 0.07894736842105265.
Trial 2 finished with value: 0.1578947368421053 and parameters: {'alpha': 0.
↪03133884470809709}. Best is trial 1 with value: 0.07894736842105265.
Trial 3 finished with value: 0.07894736842105265 and parameters: {'alpha': 5.
↪034108008965986e-05}. Best is trial 1 with value: 0.07894736842105265.
Trial 4 finished with value: 0.23684210526315785 and parameters: {'alpha': 0.
↪013203749930564852}. Best is trial 1 with value: 0.07894736842105265.
Trial 5 pruned.
Trial 6 pruned.
Trial 7 pruned.
Trial 8 finished with value: 0.052631578947368474 and parameters: {'alpha': 9.
↪321886293035837e-05}. Best is trial 8 with value: 0.052631578947368474.
Trial 9 pruned.
Trial 10 pruned.
Trial 11 pruned.
Trial 12 pruned.
Trial 13 finished with value: 0.10526315789473684 and parameters: {'alpha': 0.
↪0008551010713908288}. Best is trial 8 with value: 0.052631578947368474.
Trial 14 pruned.
Trial 15 pruned.
Trial 16 finished with value: 0.10526315789473684 and parameters: {'alpha': 0.
↪0003945128222658016}. Best is trial 8 with value: 0.052631578947368474.
Trial 17 pruned.
Trial 18 finished with value: 0.39473684210526316 and parameters: {'alpha': 2.
↪9285769426129593e-05}. Best is trial 8 with value: 0.052631578947368474.
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 `optuna.samplers.RandomSampler`, `optuna.pruners.MedianPruner` is the best.
- For `optuna.samplers.TPESampler`, `optuna.pruners.Hyperband` 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 `optuna.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 1.863 seconds)

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: 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)

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.

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

import optuna
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.

```
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": 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], verbose_eval=False, callbacks=[pruning_
↪callback]
```

(continues on next page)

(continued from previous page)

```
    )

    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)
```

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

```
plot_edf(study)
```

Total running time of the script: (0 minutes 4.288 seconds)

6.2.2 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 *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

# 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)
```

Out:

```
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)
```

Out:

```
Trial 0 finished with value: 18.04918112894684 and parameters: {'x': -2.
↪2484327850334225}. Best is trial 0 with value: 18.04918112894684.
Trial 1 finished with value: 0.0034351527707770814 and parameters: {'x': 1.
↪9413898236585396}. Best is trial 1 with value: 0.0034351527707770814.
Trial 2 finished with value: 49.844546165184354 and parameters: {'x': -5.
↪060067008547748}. Best is trial 1 with value: 0.0034351527707770814.
```

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)
```

Out:

```
Using an existing study with name 'example-study' instead of creating a new one.
Trial 3 finished with value: 8.737523671655262 and parameters: {'x': 4.
↪955930254869905}. Best is trial 1 with value: 0.0034351527707770814.
Trial 4 finished with value: 35.539802267067486 and parameters: {'x': -3.
↪961526840253886}. Best is trial 1 with value: 0.0034351527707770814.
Trial 5 finished with value: 67.33071639846814 and parameters: {'x': -6.
↪205529623276497}. Best is trial 1 with value: 0.0034351527707770814.
```

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"))
```

Out:

```
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)
```

Out:

```
   number      value  params_x     state
0       0  18.049181 -2.248433  COMPLETE
1       1   0.003435  1.941390  COMPLETE
2       2  49.844546 -5.060067  COMPLETE
3       3   8.737524  4.955930  COMPLETE
4       4  35.539802 -3.961527  COMPLETE
5       5  67.330716 -6.205530  COMPLETE
```

A `Study` object also provides properties such as `trials`, `best_value`, `best_params` (see also *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)
```

Out:

```
Best params:  {'x': 1.9413898236585396}
Best value:  0.0034351527707770814
Best Trial:  FrozenTrial(number=1, values=[0.0034351527707770814], datetime_
↪start=datetime.datetime(2021, 4, 5, 1, 58, 26, 946805), datetime_complete=datetime.
↪datetime(2021, 4, 5, 1, 58, 26, 960379), params={'x': 1.9413898236585396},␣
↪distributions={'x': UniformDistribution(high=10.0, low=-10.0)}, user_attrs={},␣
↪system_attrs={}, intermediate_values={}, trial_id=2, state=TrialState.COMPLETE,␣
↪value=None)
Trials:  [FrozenTrial(number=0, values=[18.04918112894684], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 26, 900412), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 26, 917126), params={'x': -2.2484327850334225}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=1, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=1, values=[0.0034351527707770814], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 26, 946805), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 26, 960379), params={'x': 1.9413898236585396}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=2, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=2, values=[49.844546165184354], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 26, 979536), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 26, 994053), params={'x': -5.060067008547748}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=3, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=3, values=[8.737523671655262], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 27, 51408), datetime_complete=datetime.datetime(2021, 4,
↪ 5, 1, 58, 27, 67158), params={'x': 4.955930254869905}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=4, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=4, values=[35.539802267067486], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 27, 91609), datetime_complete=datetime.datetime(2021, 4,
↪ 5, 1, 58, 27, 104881), params={'x': -3.961526840253886}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=5, state=TrialState.COMPLETE, value=None),␣
↪FrozenTrial(number=5, values=[67.33071639846814], datetime_start=datetime.
↪datetime(2021, 4, 5, 1, 58, 27, 124626), datetime_complete=datetime.datetime(2021,␣
↪4, 5, 1, 58, 27, 137693), params={'x': -6.205529623276497}, distributions={'x':␣
↪UniformDistribution(high=10.0, low=-10.0)}, user_attrs={}, system_attrs={},␣
↪intermediate_values={}, trial_id=6, state=TrialState.COMPLETE, value=None)]
```

Total running time of the script: (0 minutes 0.508 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 thop to measure FLOPS.

```
import thop
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision

import optuna


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

(continues on next page)

(continued from previous page)

```
            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, _ = thop.profile(model, inputs=(torch.randn(1, 28 * 28).to(DEVICE),),␣
↪verbose=False)
    return flops, accuracy
```

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

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

Out:

```
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%|          | 40960/26421880 [00:00<01:22, 318353.24it/s]
  0%|          | 104448/26421880 [00:00<01:13, 356690.10it/s]
  1%|1         | 293888/26421880 [00:00<00:28, 904174.54it/s]
  2%|2         | 612352/26421880 [00:00<00:15, 1672987.43it/s]
  5%|4         | 1208320/26421880 [00:00<00:08, 3066429.61it/s]
  8%|8         | 2226176/26421880 [00:00<00:04, 5321601.90it/s]
 15%|#4        | 3880960/26421880 [00:00<00:02, 8821319.44it/s]
 21%|##1       | 5550080/26421880 [00:00<00:01, 11246240.86it/s]
 27%|##7       | 7153664/26421880 [00:00<00:01, 12710085.13it/s]
 33%|###3      | 8811520/26421880 [00:01<00:01, 13885785.09it/s]
 40%|###9      | 10536960/26421880 [00:01<00:01, 14905548.89it/s]
 46%|####6     | 12235776/26421880 [00:01<00:00, 15534228.81it/s]
 53%|#####2    | 13947904/26421880 [00:01<00:00, 16012455.29it/s]
 59%|#####9    | 15603712/26421880 [00:01<00:00, 16176366.18it/s]
 66%|######5   | 17314816/26421880 [00:01<00:00, 16457367.55it/s]
 72%|#######1  | 18971648/26421880 [00:01<00:00, 16490649.29it/s]
 78%|#######8  | 20624384/26421880 [00:01<00:00, 16497768.89it/s]
 84%|########4 | 22277120/26421880 [00:01<00:00, 15531149.24it/s]
 91%|######### | 23917568/26421880 [00:02<00:00, 13208069.84it/s]
 97%|#########6| 25528320/26421880 [00:02<00:00, 13949677.52it/s]
26422272it [00:02, 12054313.31it/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 to ../FashionMNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/29515 [00:00<?, ?it/s]
29696it [00:00, 298596.97it/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 to ../FashionMNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/4422102 [00:00<?, ?it/s]
  1%|          | 39936/4422102 [00:00<00:15, 289713.62it/s]
  2%|2         | 96256/4422102 [00:00<00:10, 428896.38it/s]
  4%|3         | 161792/4422102 [00:00<00:08, 524196.82it/s]
  8%|7         | 336896/4422102 [00:00<00:04, 984908.77it/s]
 15%|#5        | 677888/4422102 [00:00<00:02, 1831572.43it/s]
 31%|###       | 1361920/4422102 [00:00<00:00, 3498368.50it/s]
 60%|######    | 2674688/4422102 [00:00<00:00, 6600922.09it/s]
 97%|#########7| 4310016/4422102 [00:00<00:00, 9674946.77it/s]
4422656it [00:00, 5272669.68it/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 to ../FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/5148 [00:00<?, ?it/s]
6144it [00:00, 37401747.13it/s]
Extracting ../FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../FashionMNIST/raw
Processing...
/home/docs/checkouts/readthedocs.org/user_builds/optuna/envs/v2.7.0/lib/python3.8/
↪site-packages/torchvision/datasets/mnist.py:479: UserWarning:

The given NumPy array is not writeable, and PyTorch does not support non-writeable␣
↪tensors. This means you can write to the underlying (supposedly non-writeable)␣
↪NumPy array using the tensor. You may want to copy the array to protect its data or␣
↪make it writeable before converting it to a tensor. This type of warning will be␣
↪suppressed for the rest of this program. (Triggered internally at  /pytorch/torch/
↪csrc/utils/tensor_numpy.cpp:143.)
```

(continues on next page)

(continued from previous page)

```

Done!
Number of finished trials:  30
```

Check trials on pareto front visually

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

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/002_multi_objective.py:123: ExperimentalWarning:

plot_pareto_front is experimental (supported from v2.4.0). The interface can change␣
↪in the future.
```

Total running time of the script: (1 minutes 37.125 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_attr` property.

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

Out:

```
{'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
↪"}
```

Out:

```
{'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
```

Out:

```
{'accuracy': 0.9733333333333334}
```

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 0.240 seconds)

Command-Line Interface

Command	Description
create-study	Create a new study.
delete-study	Delete a specified study.
dashboard	Launch web dashboard (beta).
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.

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


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))
```

Out:

```
Best value: 0.00021709814480694095 (params: {'x': 2.0147342507378876})
```

However, we can reduce boilerplate codes by using our `optuna` command. Let us assume that `foo.py` contains only the following code.

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

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*.)

```
$ cat foo.py
def objective(trial):
    x = trial.suggest_float('x', -10, 10)
    return (x - 2) ** 2

$ STUDY_NAME=`optuna create-study --storage sqlite:///example.db`
$ optuna study optimize foo.py objective --n-trials=100 --storage sqlite:///example.
↪db --study-name $STUDY_NAME
[I 2018-05-09 10:40:25,196] Finished a trial resulted in value: 54.353767789264026.␣
↪Current best value is 54.353767789264026 with parameters: {'x': -5.372500782588228}.
[I 2018-05-09 10:40:25,197] Finished a trial resulted in value: 15.784266965526376.␣
↪Current best value is 15.784266965526376 with parameters: {'x': 5.972941852774387}.
...
[I 2018-05-09 10:40:26,204] Finished a trial resulted in value: 14.704254135013741.␣
↪Current best value is 2.280758099793617e-06 with parameters: {'x': 1.
↪9984897821018828}.
```

Please note that `foo.py` only contains the definition of the objective function. By giving the script file name and the method name of objective function to `optuna study optimize` command, we can invoke the optimization.

Total running time of the script: (0 minutes 0.318 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., `UniformDistribution`) 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):

```
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.
        # 1. Calculate transition probability.
        prev_trial = study.trials[-2]
```

(continues on next page)

(continued from previous page)

```
        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.
↪UniformDistribution):
                raise NotImplementedError("Only suggest_float() is supported")

            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.samplers.intersection_search_space(study)

    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
```

(continues on next page)

(continued from previous page)

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

Out:

```
Best value:  -4.909533060293563
Parameters that achieve the best value:  {'x': 0.12068467686711326, 'y': -4.
↪924097851524082}
```

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 0.303 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_trial()` 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_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", "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)
```

Total running time of the script: (0 minutes 0.713 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:
```

(continues on next page)

(continued from previous page)

```
        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])
```

Out:

```
A new study created in memory with name: no-name-735e1409-ac50-4e3a-acfe-e3e2831e8c99
Trial 0 finished with value: 0.7835633957417836 and parameters: {'x': 0.
↪7835633957417836}. Best is trial 0 with value: 0.7835633957417836.
Trial 1 finished with value: 0.6571821231067395 and parameters: {'x': 0.
↪6571821231067395}. Best is trial 1 with value: 0.6571821231067395.
Trial 2 finished with value: 0.750899809430996 and parameters: {'x': 0.
↪750899809430996}. Best is trial 1 with value: 0.6571821231067395.
Trial 3 finished with value: 0.4328722703743789 and parameters: {'x': 0.
↪4328722703743789}. Best is trial 3 with value: 0.4328722703743789.
Trial 4 finished with value: 0.5477946556257093 and parameters: {'x': 0.
↪5477946556257093}. Best is trial 3 with value: 0.4328722703743789.
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 also 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.

```
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.

```
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-
↪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], verbose_eval=False, callbacks=[pruning_
↪callback]
    )

    preds = gbm.predict(valid_x)
```

(continues on next page)

(continued from previous page)

```
    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)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/008_specify_params.py:77: ExperimentalWarning:

enqueue_trial is experimental (supported from v1.2.0). The interface can change in␣
↪the future.

/home/docs/checkouts/readthedocs.org/user_builds/optuna/envs/v2.7.0/lib/python3.8/
↪site-packages/optuna/study.py:857: ExperimentalWarning:

create_trial is experimental (supported from v2.0.0). The interface can change in the␣
↪future.

/home/docs/checkouts/readthedocs.org/user_builds/optuna/envs/v2.7.0/lib/python3.8/
↪site-packages/optuna/study.py:856: ExperimentalWarning:

add_trial is experimental (supported from v2.0.0). The interface can change in the␣
↪future.

/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/008_specify_params.py:85: ExperimentalWarning:
```

(continues on next page)

(continued from previous page)

```
enqueue_trial is experimental (supported from v1.2.0). The interface can change in␣
↪the future.

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: 0.965034965034965 and parameters: {'bagging_fraction': 0.
↪75, 'bagging_freq': 5, 'min_child_samples': 20}. Best is trial 0 with value: 0.
↪972027972027972.
Trial 2 finished with value: 0.958041958041958 and parameters: {'bagging_fraction': 0.
↪41290140944743037, 'bagging_freq': 1, 'min_child_samples': 70}. Best is trial 0␣
↪with value: 0.972027972027972.
Trial 3 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 0.
↪9427895053283064, 'bagging_freq': 1, 'min_child_samples': 24}. Best is trial 0 with␣
↪value: 0.972027972027972.
Trial 4 finished with value: 0.9370629370629371 and parameters: {'bagging_fraction':␣
↪0.5001041104010937, 'bagging_freq': 1, 'min_child_samples': 100}. Best is trial 0␣
↪with value: 0.972027972027972.
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 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 pruned. Trial was pruned at iteration 0.
Trial 20 finished with value: 0.972027972027972 and parameters: {'bagging_fraction':␣
↪0.8595082743229087, 'bagging_freq': 7, 'min_child_samples': 6}. Best is trial 0␣
↪with value: 0.972027972027972.
Trial 21 pruned. Trial was pruned at iteration 0.
Trial 22 pruned. Trial was pruned at iteration 33.
Trial 23 pruned. Trial was pruned at iteration 17.
Trial 24 finished with value: 0.965034965034965 and parameters: {'bagging_fraction':␣
↪0.8895937677023859, 'bagging_freq': 4, 'min_child_samples': 28}. Best is trial 0␣
↪with value: 0.972027972027972.
Trial 25 pruned. Trial was pruned at iteration 2.
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 1.
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 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 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction':␣
↪0.5625447412457188, 'bagging_freq': 1, 'min_child_samples': 14}. Best is trial 37␣
↪with value: 0.9790209790209791.
```

(continues on next page)

(continued from previous page)

```
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 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction':␣
↪0.7226495098407482, 'bagging_freq': 1, 'min_child_samples': 16}. Best is trial 37␣
↪with value: 0.9790209790209791.
Trial 48 pruned. Trial was pruned at iteration 0.
Trial 49 pruned. Trial was pruned at iteration 55.
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 2.
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 1.
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 0.
Trial 63 pruned. Trial was pruned at iteration 0.
Trial 64 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction':␣
↪0.7515579258213221, 'bagging_freq': 5, 'min_child_samples': 40}. Best is trial 37␣
↪with value: 0.9790209790209791.
Trial 65 pruned. Trial was pruned at iteration 2.
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.
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.
```

(continues on next page)

(continued from previous page)

```
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.

```
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.UniformDistribution(0.4, 1.0 +␣
↪1e-12),
            "bagging_freq": optuna.distributions.IntUniformDistribution(0, 7),
            "min_child_samples": optuna.distributions.IntUniformDistribution(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.UniformDistribution(0.4, 1.0 +␣
↪1e-12),
            "bagging_freq": optuna.distributions.IntUniformDistribution(0, 7),
            "min_child_samples": optuna.distributions.IntUniformDistribution(5, 100),
        },
        value=0.95,
    )
)
study.optimize(objective, n_trials=100, timeout=600)
```

Out:

```
A new study created in memory with name: no-name-53bf1362-d063-448b-8bf7-def65048ff77
/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/008_specify_params.py:115: ExperimentalWarning:

create_trial is experimental (supported from v2.0.0). The interface can change in the␣
↪future.

/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/008_specify_params.py:114: ExperimentalWarning:

add_trial is experimental (supported from v2.0.0). The interface can change in the␣
↪future.

/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/008_specify_params.py:130: ExperimentalWarning:

create_trial is experimental (supported from v2.0.0). The interface can change in the␣
↪future.

/home/docs/checkouts/readthedocs.org/user_builds/optuna/checkouts/v2.7.0/tutorial/20_
↪recipes/008_specify_params.py:129: ExperimentalWarning:

add_trial is experimental (supported from v2.0.0). The interface can change in the␣
↪future.

Trial 2 finished with value: 0.9440559440559441 and parameters: {'bagging_fraction':␣
↪0.5727495629948887, 'bagging_freq': 3, 'min_child_samples': 84}. Best is trial 1␣
↪with value: 0.95.
Trial 3 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction':␣
↪0.4615153505478339, 'bagging_freq': 3, 'min_child_samples': 93}. Best is trial 3␣
↪with value: 0.9790209790209791.
Trial 4 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 0.
↪7153239356278835, 'bagging_freq': 3, 'min_child_samples': 24}. Best is trial 3 with␣
↪value: 0.9790209790209791.
Trial 5 finished with value: 0.972027972027972 and parameters: {'bagging_fraction': 0.
↪754908250224628, 'bagging_freq': 4, 'min_child_samples': 5}. Best is trial 3 with␣
↪value: 0.9790209790209791.
Trial 6 pruned. Trial was pruned at iteration 1.
Trial 7 pruned. Trial was pruned at iteration 1.
Trial 8 pruned. Trial was pruned at iteration 15.
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 1.
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 9.
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 65.
Trial 19 pruned. Trial was pruned at iteration 1.
Trial 20 finished with value: 0.972027972027972 and parameters: {'bagging_fraction':␣
↪0.6944101829849761, 'bagging_freq': 2, 'min_child_samples': 47}. Best is trial 3␣
↪with value: 0.9790209790209791.
Trial 21 pruned. Trial was pruned at iteration 9.
Trial 22 pruned. Trial was pruned at iteration 0.
```

(continues on next page)

(continued from previous page)

```
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 1.
Trial 26 finished with value: 0.986013986013986 and parameters: {'bagging_fraction':␣
↪0.7034498107027325, 'bagging_freq': 3, 'min_child_samples': 24}. Best is trial 26␣
↪with value: 0.986013986013986.
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':␣
↪0.784473572818712, 'bagging_freq': 5, 'min_child_samples': 26}. Best is trial 26␣
↪with 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 2.
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 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction':␣
↪0.6831898452426733, 'bagging_freq': 2, 'min_child_samples': 46}. Best is trial 26␣
↪with value: 0.986013986013986.
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 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 0.
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 finished with value: 0.965034965034965 and parameters: {'bagging_fraction':␣
↪0.7673974059823012, 'bagging_freq': 6, 'min_child_samples': 21}. Best is trial 26␣
↪with value: 0.986013986013986.
```

(continues on next page)

(continued from previous page)

```
Trial 72 pruned. Trial was pruned at iteration 0.
Trial 73 pruned. Trial was pruned at iteration 1.
Trial 74 finished with value: 0.993006993006993 and parameters: {'bagging_fraction':␣
↪0.8007167664060137, 'bagging_freq': 6, 'min_child_samples': 10}. Best is trial 74␣
↪with value: 0.993006993006993.
Trial 75 pruned. Trial was pruned at iteration 0.
Trial 76 finished with value: 0.986013986013986 and parameters: {'bagging_fraction':␣
↪0.8108681697315372, 'bagging_freq': 3, 'min_child_samples': 10}. Best is trial 74␣
↪with value: 0.993006993006993.
Trial 77 finished with value: 0.9790209790209791 and parameters: {'bagging_fraction':␣
↪0.829171796994586, 'bagging_freq': 3, 'min_child_samples': 9}. Best is trial 74␣
↪with value: 0.993006993006993.
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 2.
Trial 82 pruned. Trial was pruned at iteration 0.
Trial 83 finished with value: 0.986013986013986 and parameters: {'bagging_fraction':␣
↪0.8299363069819753, 'bagging_freq': 4, 'min_child_samples': 12}. Best is trial 74␣
↪with value: 0.993006993006993.
Trial 84 pruned. Trial was pruned at iteration 0.
Trial 85 pruned. Trial was pruned at iteration 1.
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 2.
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.
Trial 100 pruned. Trial was pruned at iteration 0.
Trial 101 pruned. Trial was pruned at iteration 2.
```

Total running time of the script: (0 minutes 6.784 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)

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

    clf = LogisticRegression(C=C, solver=solver)
```

(continues on next page)

(continued from previous page)

```
    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:

```
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_uniform("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.LogUniformDistribution(1e-7, 10.0),
    "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` instead of a single hyperparameter and calculates the objective over the full vector.

```
def batched_objective(xs: np.ndarray):
    return xs ** 2 + 1
```

In the following example, the number 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_ids` or `trial` to call `optuna.study.Study.tell()` method after the batched evaluations.

```
batch_size = 10
study = optuna.create_study()

```

(continues on next page)

(continued from previous page)

```
for _ in range(3):

    # create batch
    trial_ids = []
    samples = []
    for _ in range(batch_size):
        trial = study.ask()
        trial_ids.append(trial.number)
        x = trial.suggest_int("x", -10, 10)
        samples.append(x)

    # evaluate batched objective
    samples = np.array(samples)
    objectives = batched_objective(samples)

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

Total running time of the script: (0 minutes 0.155 seconds)

Re-use the best values

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_loguniform("C", 1e-7, 10.0)

    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.
```

Out:

```
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_loguniform("C", 1e-7, 10.0)

    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
```

Out:

```
(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`.

Total running time of the script: (0 minutes 0.056 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.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

```
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`** (*`Optional[Union[str, optuna.storages._base.BaseStorage]]`*) – 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`** (*`Optional[optuna.samplers._base.BaseSampler]`*) – 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`** (*`Optional[optuna.pruners._base.BasePruner]`*) – 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`** (*`Optional[str]`*) – Study's name. If this argument is set to None, a unique name is generated automatically.
- **`direction`** (*`Optional[Union[str, optuna._study_direction.StudyDirection]]`*) – Direction of optimization. Set `minimize` for minimization and `maximize` for maximization. You can also pass the corresponding *`StudyDirection`* object.

  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`** (*`Optional[Sequence[Union[str, optuna._study_direction.StudyDirection]]]`*) – A sequence of directions during multi-objective optimization.

Returns A *`Study`* object.

Raises **`ValueError`** – If the length of `directions` is zero. Or, if `direction` is neither 'minimize' nor 'maximize' when it is a string. Or, if the element of `directions` is neither *minimize* nor *maximize*. Or, if both `direction` and `directions` are specified.

Return type *optuna.study.Study*

See also:

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

optuna.load_study

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

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

Example

```
import optuna


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


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

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

Parameters

- **study_name** (*str*) – Study's name. Each study has a unique name as an identifier.
- **storage** (*Union[str, optuna.storages._base.BaseStorage]*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of `create_study()` for further details.
- **sampler** (*Optional[optuna.samplers._base.BaseSampler]*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, `TPESampler` is used as the default. See also `samplers`.
- **pruner** (*Optional[optuna.pruners._base.BasePruner]*) – 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 *optuna.study.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")
```

(continues on next page)

(continued from previous page)

```
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** (*Union[str, optuna.storages._base.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.get_all_study_summaries

optuna.**get_all_study_summaries**(*storage*)

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** (*Union[str, optuna.storages._base.BaseStorage]*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *create_study()* for further details.

Returns List of study history summarized as *StudySummary* objects.

Return type List[optuna._study_summary.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 using the cliff framework.

```
optuna
    [--version]
    [-v | -q]
    [--log-file LOG_FILE]
    [--debug]
    [--storage STORAGE]
```

`--version`
show program's version number and exit

`-v, --verbose`
Increase verbosity of output. Can be repeated.

`-q, --quiet`
Suppress output except warnings and errors.

`--log-file` `<LOG_FILE>`
Specify a file to log output. Disabled by default.

`--debug`
Show tracebacks on errors.

`--storage` `<STORAGE>`
DB URL. (e.g. sqlite:///example.db)

create-study

Create a new study.

```
optuna create-study
    [--study-name STUDY_NAME]
    [--direction {minimize,maximize}]
    [--skip-if-exists]
```

`--study-name` `<STUDY_NAME>`
A human-readable name of a study to distinguish it from others.

`--direction` `<DIRECTION>`
Set direction of optimization to a new study. Set 'minimize' for minimization and 'maximize' for maximization.

`--skip-if-exists`
If specified, the creation of the study is skipped without any error when the study name is duplicated.

This command is provided by the optuna plugin.

dashboard

Launch web dashboard (beta).

This feature is deprecated since version 2.7.0. Please use optuna-dashboard instead.

```
optuna dashboard
    [--study STUDY]
    [--study-name STUDY_NAME]
    [--out OUT]
    [--allow-websocket-origin BOKEH_ALLOW_WEBSOCKET_ORIGINS]
```

`--study` `<STUDY>`
This argument is deprecated. Use –study-name instead.

`--study-name` `<STUDY_NAME>`
The name of the study to show on the dashboard.

`--out` `<OUT>`**`, -o`** `<OUT>`
Output HTML file path. If it is not given, a HTTP server starts and the dashboard is served.

`--allow-websocket-origin` `<BOKEH_ALLOW_WEBSOCKET_ORIGINS>`
Allow websocket access from the specified host(s).Internally, it is used as the value of bokeh's –allow-websocket-origin option. Please refer to https://bokeh.pydata.org/en/latest/docs/reference/command/subcommands/serve.html for more details.

This command is provided by the optuna plugin.

delete-study

Delete a specified study.

```
optuna delete-study [--study-name STUDY_NAME]
```

`--study-name` `<STUDY_NAME>`
The name of the study to delete.

This command is provided by the optuna plugin.

storage upgrade

Upgrade the schema of a storage.

```
optuna storage upgrade
```

This command is provided by the optuna plugin.

studies

Show a list of studies.

```
optuna studies
    [-f {csv,json,table,value,yaml}]
    [-c COLUMN]
    [--quote {all,minimal,none,nonnumeric}]
    [--noindent]
    [--max-width <integer>]
    [--fit-width]
    [--print-empty]
    [--sort-column SORT_COLUMN]
    [--sort-ascending | --sort-descending]
```

`-f` `<FORMATTER>`, **`--format`** `<FORMATTER>`
    the output format, defaults to table

`-c` `COLUMN`, **`--column`** `COLUMN`
    specify the column(s) to include, can be repeated to show multiple columns

`--quote` `<QUOTE_MODE>`
    when to include quotes, defaults to nonnumeric

`--noindent`
    whether to disable indenting the JSON

`--max-width` `<integer>`
    Maximum display width, <1 to disable. You can also use the CLIFF_MAX_TERM_WIDTH environment variable, but the parameter takes precedence.

`--fit-width`
    Fit the table to the display width. Implied if –max-width greater than 0. Set the environment variable CLIFF_FIT_WIDTH=1 to always enable

`--print-empty`
    Print empty table if there is no data to show.

`--sort-column` `SORT_COLUMN`
    specify the column(s) to sort the data (columns specified first have a priority, non-existing columns are ignored), can be repeated

`--sort-ascending`
    sort the column(s) in ascending order

`--sort-descending`
    sort the column(s) in descending order

This command is provided by the optuna plugin.

study optimize

Start optimization of a study. Deprecated since version 2.0.0.

```
optuna study optimize
    [--n-trials N_TRIALS]
    [--timeout TIMEOUT]
    [--n-jobs N_JOBS]
    [--study STUDY]
    [--study-name STUDY_NAME]
    file
    method
```

--n-trials `<N_TRIALS>`
The number of trials. If this argument is not given, as many trials run as possible.

--timeout `<TIMEOUT>`
Stop study after the given number of second(s). If this argument is not given, as many trials run as possible.

--n-jobs `<N_JOBS>`
The number of parallel jobs. If this argument is set to -1, the number is set to CPU counts.

--study `<STUDY>`
This argument is deprecated. Use –study-name instead.

--study-name `<STUDY_NAME>`
The name of the study to start optimization on.

file
Python script file where the objective function resides.

method
The method name of the objective function.

This command is provided by the optuna plugin.

study set-user-attr

Set a user attribute to a study.

```
optuna study set-user-attr
    [--study STUDY]
    [--study-name STUDY_NAME]
    --key KEY
    --value VALUE
```

--study `<STUDY>`
This argument is deprecated. Use –study-name instead.

--study-name `<STUDY_NAME>`
The name of the study to set the user attribute to.

--key `<KEY>`, **-k** `<KEY>`
Key of the user attribute.

--value `<VALUE>`, **-v** `<VALUE>`
Value to be set.

This command is provided by the optuna plugin.

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 *UniformDistribution*.

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

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.UniformDistribution

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

A uniform distribution in the linear domain.

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

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 from the range.

Raises **ValueError** – If `low` value is larger than `high` value.

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** (*Any*) – 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 *suggest_loguniform()*, and passed to *samplers* in general.

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 from the range.

Raises **ValueError** – If `low` value is larger than `high` value, or `low` value is smaller than or equal to 0.

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`** (*`Any`*) – 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_uniform()` with `step` argument and `suggest_discrete_uniform()`, and passed to `samplers` in general.

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

`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.

`q`

A discretization step.

Raises **`ValueError`** – If `low` value is larger than `high` value.

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`** (*`Any`*) – 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 $[\mathsf{low}, \mathsf{high}]$ is not divisible by step, high will be replaced with the maximum of $k \times \mathsf{step} + \mathsf{low} < \mathsf{high}$, where k is an integer.

`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.

`step`

A step for spacing between values.

Raises **`ValueError`** – If `low` value is larger than `high` value, or `step` value is smaller or equal to 0.

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.

`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.

`step`

A step for spacing between values.

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.

Raises `ValueError` – If `low` value is larger than `high` value, or `low` value is smaller than 1.

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

`step`

`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`** – Parameter value candidates.

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.

Raises `ValueError` – If `choices` do not contain any elements.

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 Union[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** (*Union[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** (*optuna.distributions.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.

Raises **`ValueError`** – If the unknown class is specified.

Return type optuna.distributions.BaseDistribution

optuna.distributions.check_distribution_compatibility

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

A function to check compatibility of two distributions.

Note that this method is not supposed to be called by library users.

Parameters

- **`dist_old`** (*`optuna.distributions.BaseDistribution`*) – A distribution previously recorded in storage.
- **`dist_new`** (*`optuna.distributions.BaseDistribution`*) – A distribution newly added to storage.

Returns True denotes given distributions are compatible. Otherwise, they are not.

Raises **`ValueError`** – If different distribution kinds are set to `dist_old` and `dist_new`, or `dist_old.choices` doesn't match `dist_new.choices` for *`CategoricalDistribution`*.

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()

    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*`)`

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 floating point numbers that sum to 1.0 over the entire dictionary. The higher the value, the more important. The returned dictionary is of type `collections.OrderedDict` and is ordered by its values in a descending order.

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`** (`optuna.study.Study`) – An optimized study.
- **`evaluator`** (*`Optional[optuna.importance._base.BaseImportanceEvaluator]`*) – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *`FanovaImportanceEvaluator`*.
- **`params`** (*`Optional[List[str]]`*) – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

  Note: Specify this argument if `study` is being used for multi-objective optimization.

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

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

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.

Given a study, fANOVA fits a random forest regression model that predicts the objective value given a parameter configuration. The more accurate this model is, the more reliable the importances assessed by this class are.

Note: Requires the sklearn Python package.

Note: Pairwise and higher order importances are not supported through this class. They can be computed using `_Fanova` directly but is not recommended as interfaces may change without prior notice.

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`** – The number of trees in the forest.
- **`max_depth`** – The maximum depth of the trees in the forest.
- **`seed`** – 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`** (`optuna.study.Study`) – An optimized study.
- **`params`** (*`Optional[List[str]]`*) – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

  Note: Specify this argument if `study` is being used for multi-objective optimization.

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

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

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 a random forest that predicts objective values given hyperparameter configurations. Feature importances are then computed using MDI.

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

Parameters

- **`n_trees`** – Number of trees in the random forest.
- **`max_depth`** – The maximum depth of each tree in the random forest.
- **`seed`** – 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`** (`optuna.study.Study`) – An optimized study.
- **`params`** (*`Optional[List[str]]`*) – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

  Note: Specify this argument if `study` is being used for multi-objective optimization.

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

Raises `ValueError` – If `target` is `None` and `study` is being used for multi-objective optimization.

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.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.

AllenNLP

`optuna.integration.AllenNLPExecutor`	AllenNLP extension to use optuna with Jsonnet config file.
`optuna.integration.allennlp.dump_best_config`	Save JSON config file after updating with parameters from the best trial in the study.
`optuna.integration.AllenNLPPruningCallback`	AllenNLP callback to prune unpromising trials.

optuna.integration.AllenNLPExecutor

class `optuna.integration.`**`AllenNLPExecutor`**(*trial, config_file, serialization_dir, metrics='best_validation_accuracy', *, include_package=None, force=False, file_friendly_logging=False*)

AllenNLP extension to use optuna with Jsonnet config file.

This feature is experimental since AllenNLP major release will come soon. The interface may change without prior notice to correspond to the update.

See the examples of objective function.

You can also see the tutorial of our AllenNLP integration on AllenNLP Guide.

Note: From Optuna v2.1.0, users have to cast their parameters by using methods in Jsonnet. Call `std.parseInt` for integer, or `std.parseJson` for floating point. Please see the example configuration.

Note: In *`AllenNLPExecutor`*, you can pass parameters to AllenNLP by either defining a search space using Optuna suggest methods or setting environment variables just like AllenNLP CLI. If a value is set in both a search space in Optuna and the environment variables, the executor will use the value specified in the search space in Optuna.

Parameters

- **`trial`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`config_file`** – Config file for AllenNLP. Hyperparameters should be masked with `std.extVar`. Please refer to the config example.
- **`serialization_dir`** – A path which model weights and logs are saved.
- **`metrics`** – An evaluation metric for the result of `objective`.
- **`force`** – If `True`, an executor overwrites the output directory if it exists.
- **`file_friendly_logging`** – If `True`, tqdm status is printed on separate lines and slows tqdm refresh rate.
- **`include_package`** – Additional packages to include. For more information, please see AllenNLP documentation.

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

`run`()	Train a model using AllenNLP.

`run` ()

Train a model using AllenNLP.

Return type float

optuna.integration.allennlp.dump_best_config

`optuna.integration.allennlp.`**`dump_best_config`** (*input_config_file*, *output_config_file*, *study*)

Save JSON config file after updating with parameters from the best trial in the study.

Parameters

- **`input_config_file`** (*`str`*) – Input Jsonnet config file used with *`AllenNLPExecutor`*.
- **`output_config_file`** (*`str`*) – Output JSON config file.
- **`study`** (`optuna.study.Study`) – Instance of *`Study`*. Note that *`optimize()`* must have been called.

Return type None

optuna.integration.AllenNLPPruningCallback

class `optuna.integration.`**`AllenNLPPruningCallback`**(*trial=None*, *monitor=None*)

AllenNLP callback to prune unpromising trials.

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

You can also see the tutorial of our AllenNLP integration on AllenNLP Guide.

Note: When `AllenNLPPruningCallback` is instantiated in Python script, trial and monitor are mandatory.

On the other hand, when `AllenNLPPruningCallback` is used with `AllenNLPExecutor`, `trial` and `monitor` would be `None`. `AllenNLPExecutor` sets environment variables for a study name, trial id, monitor, and storage. Then `AllenNLPPruningCallback` loads them to restore `trial` and `monitor`.

Parameters

- **trial** – A `Trial` corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g. `validation_loss` or `validation_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.

Methods

`on_epoch`(trainer, metrics, epoch[, is_primary])	Check if a training reaches saturation.
`register`(*args, **kwargs)	Stub method for *TrainerCallback.register*.

on_epoch(*trainer*, *metrics*, *epoch*, *is_primary=True*, ***kwargs*)

Check if a training reaches saturation.

Parameters

- **trainer** (`GradientDescentTrainer`) – AllenNLP's trainer
- **metrics** (`Dict[str, Any]`) – Dictionary of metrics.
- **epoch** (`int`) – Number of current epoch.
- **is_primary** (`bool`) – A flag for AllenNLP internal.
- **kwargs** (`Any`) –

Return type None

classmethod register(**args*, ***kwargs*)

Stub method for *TrainerCallback.register*.

This method has the same signature as Registrable.register in AllenNLP.

Parameters

- **args** (`Any`) –

- **kwargs** (*Any*) –

Return type Callable

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.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*)

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 be not 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** – 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, 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 larger than one but smaller than four, 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** – 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 omitted, no constraints will be passed to `candidates_func` nor taken into account during suggestion if `candidates_func` is omitted.
- **n_startup_trials** – Number of initial trials, that is the number of trials to resort to independent sampling.
- **independent_sampler** – An independent sampler to use for the initial trials and for parameters that are conditional.

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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.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, optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **param_name** (*str*) – Name of the sampled parameter.
- **param_distribution** (*optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`search_space`** (*`Dict[str, optuna.distributions.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`** (*`torch.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`** (*`torch.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`** (*`Optional[torch.Tensor]`*) – 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`** (*`torch.Tensor`*) – Search space bounds. A `torch.Tensor` of shape `(n_params, 2)`. `n_params` is identical to that of `train_x`. The first and the second column 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 torch.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`** (*`torch.Tensor`*) –
- **`train_obj`** (*`torch.Tensor`*) –
- **`train_con`** (*`Optional[torch.Tensor]`*) –
- **`bounds`** (*`torch.Tensor`*) –

Return type torch.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`** (*`torch.Tensor`*) –
- **`train_obj`** (*`torch.Tensor`*) –
- **`train_con`** (*`Optional[torch.Tensor]`*) –

- **`bounds`** (*`torch.Tensor`*) –

Return type torch.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.

Chainer

`optuna.integration.ChainerPruningExtension`	Chainer extension to prune unpromising trials.
`optuna.integration.ChainerMNStudy`	A wrapper of *`Study`* to incorporate Optuna with ChainerMN.

optuna.integration.ChainerPruningExtension

class `optuna.integration.`**`ChainerPruningExtension`**(*trial*, *observation_key*, *pruner_trigger*)

Chainer extension to prune unpromising trials.

See the example if you want to add a pruning extension which observes validation accuracy of a Chainer Trainer.

Parameters

- **`trial`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`observation_key`** – An evaluation metric for pruning, e.g., `main/loss` and `validation/main/accuracy`. Please refer to chainer.Reporter reference for further details.
- **`pruner_trigger`** – A trigger to execute pruning. `pruner_trigger` is an instance of IntervalTrigger or ManualScheduleTrigger. IntervalTrigger can be specified by a tuple of the interval length and its unit like `(1, 'epoch')`.

optuna.integration.ChainerMNStudy

class optuna.integration.**ChainerMNStudy**(*study*, *comm*)

A wrapper of `Study` to incorporate Optuna with ChainerMN.

See also:

`ChainerMNStudy` provides the same interface as `Study`. Please refer to `optuna.study.Study` for further details.

See the example if you want to optimize an objective function that trains neural network written with ChainerMN.

Parameters

- **study** – A `Study` object.
- **comm** – A ChainerMN communicator.

Methods

`optimize`(func[, n_trials, timeout, catch])	Optimize an objective function.

optimize(*func*, *n_trials=None*, *timeout=None*, *catch=()*)

Optimize an objective function.

This method provides the same interface as `optuna.study.Study.optimize()` except the absence of `n_jobs` argument.

Parameters

- **func** (*Callable[[ChainerMNTrial, CommunicatorBase], float]*) –
- **n_trials** (*Optional[int]*) –
- **timeout** (*Optional[float]*) –
- **catch** (*Tuple[Type[Exception], ..]*) –

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 `optuna.integration.fastaiv2.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`** – fastai.basic_train.Learner.
- **`trial`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** – 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`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** – 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 `optuna.integration.fastaiv2.FastAIV2PruningCallback`

Keras

`optuna.integration.KerasPruningCallback`	Keras callback to prune unpromising trials.

optuna.integration.KerasPruningCallback

`class` `optuna.integration.`**`KerasPruningCallback`**(*trial*, *monitor*, *interval=1*)
Keras callback to prune unpromising trials.

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

Parameters

- **`trial`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** – An evaluation metric for pruning, e.g., `val_loss` and `val_accuracy`. Please refer to keras.Callback reference for further details.
- **`interval`** – Check if trial should be pruned every n-th epoch. By default `interval=1`

and pruning is performed after every epoch. Increase `interval` to run several epochs faster before applying pruning.

> **Warning:** Deprecated in v2.1.0. 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/v2.1.0.
>
> Recent Keras release (2.4.0) simply redirects all APIs in the standalone keras package to point to tf.keras. There is now only one Keras: tf.keras. There may be some breaking changes for some workflows by upgrading to keras 2.4.0. Test before upgrading. REF:https://github.com/keras-team/keras/releases/tag/2.4.0

Methods

`on_epoch_end`(epoch[, logs])	

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'*)

Callback for LightGBM to prune unpromising trials.

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

Parameters

- **`trial`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`metric`** – An evaluation metric for pruning, e.g., `binary_error` and `multi_error`. Please refer to LightGBM reference for further details.
- **`valid_name`** – 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.

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=True*, *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*)

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** – A time budget for parameter tuning in seconds.
- **study** – 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 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`** – 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`** – 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`** – 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.

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])	

continues on next page

Table 28 – continued from previous page

`tune_regularization_factors([n_trials])`	

Attributes

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

property best_booster

Return the best booster.

> **Warning:** Deprecated in v1.4.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v3.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v1.4.0.
>
> Please get the best booster via *`get_best_booster`* instead.

property best_params

Return parameters of the best booster.

property best_score

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 lightgbm.basic.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=True*, *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=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** – A time budget for parameter tuning in seconds.
- **study** – 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 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** – 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** – 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`** – 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`** – Flag to enable `get_best_booster()`.

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`

Return parameters of the best booster.

`property best_score`

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 lightgbm.engine.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'*, *nest_trials=False*, *tag_study_user_attrs=False*)

Callback to track Optuna trials with MLflow.

This callback adds relevant information that is tracked by Optuna to MLflow. The MLflow experiment will be named after the Optuna study name.

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** – The URI of the MLflow tracking server.

  Please refer to mlflow.set_tracking_uri for more details.
- **metric_name** – 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.
- **nest_trials** – Flag indicating whether or not trials should be logged as nested runs. This is often helpful for aggregating trials to a particular study, under a given experiment.

- **`tag_study_user_attrs`** – 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 study.user_attrs will supersede existing tags.

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.

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`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`eval_metric`** – 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)
    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** – 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** – 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** – 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** – A random seed for CMA-ES.
- **cma_opts** – Options passed to the constructor of cma.CMAEvolutionStrategy class.

  Note that `BoundaryHandler`, `bounds`, `CMA_stds` and `seed` arguments in `cma_opts` will be ignored because it is added by *PyCmaSampler* automatically.
- **n_startup_trials** – The independent sampling is used instead of the CMA-ES algorithm until the given number of trials finish in the same study.
- **independent_sampler** – 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`** – 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 trials 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`state`** (*`optuna.trial._state.TrialState`*) – Resulting trial state.
- **`values`** (*`Optional[Sequence[float]]`*) – 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.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, optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **param_name** (*str*) – Name of the sampled parameter.
- **param_distribution** (*optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.

- **search_space** (*Dict[str, optuna.distributions.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.

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

Trial post-processing.

This method is called after the objective function returns and right before the trials 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** (optuna.study.Study) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (`optuna.trial._frozen.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, optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (`optuna.trial._frozen.FrozenTrial`) – Target trial object. Take a copy before modifying this object.
- **param_name** (`str`) – Name of the sampled parameter.
- **param_distribution** (`optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **search_space** (*Dict[str, optuna.distributions.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** – A *Trial* corresponding to the current evaluation of the objective function.
- **metric** – A name of metric for pruning, e.g., `accuracy` and `loss`.
- **trainer** – 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`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** – 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.

Methods

`on_validation_end`(trainer, pl_module)

optuna.integration.TorchDistributedTrial

class `optuna.integration.`**`TorchDistributedTrial`**(*trial*, *device=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`** – A *`Trial`* object or `None`. Please set trial object in rank-0 node and set `None` in the other rank node.
- **`device`** – A *torch.device* to communicate with the other nodes. Please set a CUDA device assigned to the current node if you use “nccl” as *torch.distributed* backend.

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`(name, choices)
`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`

scikit-learn

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

optuna.integration.OptunaSearchCV

class `optuna.integration.`**`OptunaSearchCV`**(*estimator*, *param_distributions*, *cv=5*, *enable_pruning=False*, *error_score=nan*, *max_iter=1000*, *n_jobs=1*, *n_trials=10*, *random_state=None*, *refit=True*, *return_train_score=False*, *scoring=None*, *study=None*, *subsample=1.0*, *timeout=None*, *verbose=0*)

Hyperparameter search with cross-validation.

Parameters

- **`estimator`** – 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`** – Dictionary where keys are parameters and values are distributions. Distributions are assumed to implement the optuna distribution interface.
- **`cv`** – Cross-validation strategy. Possible inputs for cv are:
  - integer to specify the number of folds in a CV splitter,
  - a 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`** – If `True`, pruning is performed in the case where the underlying estimator supports `partial_fit`.
- **`error_score`** – 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`** – Maximum number of epochs. This is only used if the underlying estimator supports `partial_fit`.
- **`n_jobs`** – Number of `threading` based parallel jobs. `-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.

  Warning: Deprecated in v2.7.0. This feature will be removed in the future. It is recommended to use *process-based parallelization*. 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.
- **`n_trials`** – 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`** – 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`** – 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`** – 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`** – String or callable to evaluate the predictions on the validation data. If `None`, `score` on the estimator is used.
- **`study`** – Study corresponds to the optimization task. If `None`, a new study is created.
- **`subsample`** – 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`** – 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`** – Verbosity level. The higher, the more messages.

`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

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

clf = SVC(gamma="auto")
param_distributions = {"C": optuna.distributions.LogUniformDistribution(1e-10,␣
↪1e10)}
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: 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_*	Index 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_

Index which corresponds to the best candidate parameter setting.

property best_params_

Parameters of the best trial in the `Study`.

property best_score_

Mean cross-validated score of the best estimator.

property best_trial_

Best trial in the `Study`.

property classes_

Class labels.

property decision_function

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** (*Union[List[List[float]], numpy.ndarray, pandas.core.frame.DataFrame, scipy.sparse.base.spmatrix]*) – Training data.
- **y** (*Optional[Union[List[float], numpy.ndarray, pandas.core.series.Series, List[List[float]], pandas.core.frame.DataFrame, scipy.sparse.base.spmatrix]]*) – Target variable.
- **groups** (*Optional[Union[List[float], numpy.ndarray, pandas.core.series.Series]]*) – Group labels for the samples used while splitting the dataset into train/validation set.
- ****fit_params** – Parameters passed to `fit` on the estimator.
- **fit_params** (*Any*) –

Returns Return self.

Return type self

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

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_

Actual number of trials.

property predict

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

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

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** (*Union[List[List[float]], numpy.ndarray, pandas.core.frame.DataFrame, scipy.sparse.base.spmatrix]*) – Data.
- **y** (*Optional[Union[List[float], numpy.ndarray, pandas.core.series.Series, List[List[float]], pandas.core.frame.DataFrame, scipy.sparse.base.spmatrix]]*) – Target variable.

Returns Scaler score.

Return type score

property score_samples

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

Call `set_user_attr` on the *Study*.

property transform

Call `transform` on the best estimator.

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

property trials_

All trials in the *Study*.

property trials_dataframe

Call `trials_dataframe` on the *Study*.

property user_attrs_

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*)

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** – 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** – 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** – 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** – The independent sampling is used until the given number of trials finish in the same study.

- **`consider_pruned_trials`** – 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.

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 trials 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`state`** (*`optuna.trial._state.TrialState`*) – Resulting trial state.
- **`values`** (*`Optional[Sequence[float]]`*) – 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.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, optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`search_space`** (*`Dict[str, optuna.distributions.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]

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`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** – 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`** – Directory to store TensorBoard logs.
- **`metric_name`** – 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`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`estimator`** – An estimator which you will use.
- **`metric`** – An evaluation metric for pruning, e.g., `accuracy` and `loss`.
- **`run_every_steps`** – 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`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`monitor`** – 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 AUC of a XGBoost model.

Parameters

- **`trial`** – A *`Trial`* corresponding to the current evaluation of the objective function.
- **`observation_key`** – 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.

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.

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.

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.multi_objective

This module is deprecated, with former functionality moved to *`optuna.samplers`*, *`optuna.study`*, *`optuna.trial`* and *`optuna.visualization`*.

optuna.multi_objective.samplers

`optuna.multi_objective.samplers.BaseMultiObjectiveSampler`	Base class for multi-objective samplers.
`optuna.multi_objective.samplers.NSGAIIMultiObjectiveSampler`	Multi-objective sampler using the NSGA-II algorithm.
`optuna.multi_objective.samplers.RandomMultiObjectiveSampler`	Multi-objective sampler using random sampling.
`optuna.multi_objective.samplers.MOTPEMultiObjectiveSampler`	Multi-objective sampler using the MOTPE algorithm.

optuna.multi_objective.samplers.BaseMultiObjectiveSampler

class `optuna.multi_objective.samplers.`**`BaseMultiObjectiveSampler`**(**args*, ***kwargs*)

Base class for multi-objective samplers.

The abstract methods of this class are the same as ones defined by *`BaseSampler`* except for taking multi-objective versions of study and trial instances as the arguments.

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

`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.

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** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **trial** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.

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

Return type Dict[str, optuna.distributions.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 *`MultiObjectiveStudy`* 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 the relationship between parameters such as random sampling.

Parameters

- **study** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **trial** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.
- **param_name** (*str*) – Name of the sampled parameter.
- **param_distribution** (*optuna.distributions.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 the relationship between parameters.

Parameters

- **study** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **trial** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.
- **search_space** (*Dict[str, optuna.distributions.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.multi_objective.samplers.NSGAIIMultiObjectiveSampler

class optuna.multi_objective.samplers.**NSGAIIMultiObjectiveSampler**(*population_size=50*, *mutation_prob=None*, *crossover_prob=0.9*, *swapping_prob=0.5*, *seed=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`** – Number of individuals (trials) in a generation.
- **`mutation_prob`** – 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_prob`** – Probability that a crossover (parameters swapping between parents) will occur when creating a new individual.
- **`swapping_prob`** – Probability of swapping each parameter of the parents during crossover.
- **`seed`** – Seed for random number generator.

> **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

`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.

`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** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **trial** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.

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

Return type Dict[str, optuna.distributions.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 *`MultiObjectiveStudy`* 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 the relationship between parameters such as random sampling.

Parameters

- **`study`** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **`trial`** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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 the relationship between parameters.

Parameters

- **`study`** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **`trial`** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.
- **`search_space`** (*`Dict[str, optuna.distributions.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.multi_objective.samplers.RandomMultiObjectiveSampler

class `optuna.multi_objective.samplers.`**`RandomMultiObjectiveSampler`**(*seed=None*)

Multi-objective sampler using random sampling.

This sampler is based on *independent sampling*. See also *`BaseMultiObjectiveSampler`* for more details of ‘independent sampling’.

Example

```
import optuna
from optuna.multi_objective.samplers import RandomMultiObjectiveSampler


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


study = optuna.multi_objective.create_study(
    ["minimize", "minimize"], sampler=RandomMultiObjectiveSampler()
)
study.optimize(objective, n_trials=10)
```

Args: seed: Seed for random number generator.

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

`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.

`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`** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **`trial`** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.

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

Return type Dict[str, optuna.distributions.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 *`MultiObjectiveStudy`* 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 the relationship between parameters such as random sampling.

Parameters

- **`study`** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **`trial`** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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 the relationship between parameters.

Parameters

- **`study`** (`optuna.multi_objective.study.MultiObjectiveStudy`) – Target study object.
- **`trial`** (`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`) – Target trial object.
- **`search_space`** (*`Dict[str, optuna.distributions.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.multi_objective.samplers.MOTPEMultiObjectiveSampler

class `optuna.multi_objective.samplers.`**`MOTPEMultiObjectiveSampler`**(*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

Parameters

- **`consider_prior`** – Enhance the stability of Parzen estimator by imposing a Gaussian prior when `True`. The prior is only effective if the sampling distribution is either *`UniformDistribution`*, *`DiscreteUniformDistribution`*, *`LogUniformDistribution`*, *`IntUniformDistribution`*, or *`IntLogUniformDistribution`*.
- **`prior_weight`** – The weight of the prior. This argument is used in *`UniformDistribution`*, *`DiscreteUniformDistribution`*, *`LogUniformDistribution`*, *`IntUniformDistribution`*, *`IntLogUniformDistribution`*, and *`CategoricalDistribution`*.
- **`consider_magic_clip`** – Enable a heuristic to limit the smallest variances of Gaussians used in the Parzen estimator.
- **`consider_endpoints`** – 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`** – 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`** – Number of candidate samples used to calculate the expected hypervolume improvement.
- **`gamma`** – 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`** – 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`** – 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 = 9
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


sampler = optuna.multi_objective.samplers.MOTPEMultiObjectiveSampler(
    n_startup_trials=n_startup_trials, n_ehvi_candidates=24, seed=seed
)
study = optuna.multi_objective.create_study(
    ["minimize"] * num_variables, sampler=sampler
)
study.optimize(objective, n_trials=250)
```

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

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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (*Union[*`optuna.study.Study`*,* `optuna.multi_objective.study.MultiObjectiveStudy`*]*) – Target study object.

- **trial** (*Union[optuna.trial._frozen.FrozenTrial, optuna.multi_objective.trial.FrozenMultiObjectiveTrial]*) – 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, optuna.distributions.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** (*Union[optuna.study.Study, optuna.multi_objective.study.MultiObjectiveStudy]*) – Target study object.
- **trial** (*Union[optuna.trial._frozen.FrozenTrial, optuna.multi_objective.trial.FrozenMultiObjectiveTrial]*) – Target trial object. Take a copy before modifying this object.
- **param_name** (*str*) – Name of the sampled parameter.
- **param_distribution** (*optuna.distributions.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** (*Union[*`optuna.study.Study`*,* `optuna.multi_objective.study.MultiObjectiveStudy`*]*) – Target study object.
- **trial** (*Union[optuna.trial._frozen.FrozenTrial,* `optuna.multi_objective.trial.FrozenMultiObjectiveTrial`*]*) – Target trial object. Take a copy before modifying this object.
- **search_space** (*Dict[*str*, optuna.distributions.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.multi_objective.study

`optuna.multi_objective.study.MultiObjectiveStudy`	A study corresponds to a multi-objective optimization task, i.e., a set of trials.
`optuna.multi_objective.study.create_study`	Create a new `MultiObjectiveStudy`.
`optuna.multi_objective.study.load_study`	Load the existing `MultiObjectiveStudy` that has the specified name.

optuna.multi_objective.study.MultiObjectiveStudy

class `optuna.multi_objective.study.`**`MultiObjectiveStudy`**(*study*)

A study corresponds to a multi-objective 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.

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

enqueue_trial(params)	Enqueue a trial with given parameter values.
get_pareto_front_trials()	Return trials located at the pareto front in the study.
get_trials([deepcopy, states])	Return all trials in the study.
optimize(objective[, timeout, n_trials, . . .])	Optimize an objective function.
set_system_attr(key, value)	Set a system attribute to the study.
set_user_attr(key, value)	Set a user attribute to the study.

Attributes

directions	Return the optimization direction list.
n_objectives	Return the number of objectives.
sampler	Return the sampler.
system_attrs	Return system attributes.
trials	Return all trials in the study.
user_attrs	Return user attributes.

property directions

Return the optimization direction list.

Returns A list that contains the optimization direction for each objective value.

enqueue_trial(*params*)

Enqueue a trial with given parameter values.

You can fix the next sampling parameters which will be evaluated in your objective function.

Please refer to the documentation of `optuna.study.Study.enqueue_trial()` for further details.

Parameters **params** (`Dict[str, Any]`) – Parameter values to pass your objective function.

Return type None

get_pareto_front_trials()

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 `FrozenMultiObjectiveTrial` objects.

Return type List[*optuna.multi_objective.trial.FrozenMultiObjectiveTrial*]

get_trials(*deepcopy=True*, *states=None*)

Return all trials in the study.

The returned trials are ordered by trial number.

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`** (*`Optional[Tuple[optuna.trial._state.TrialState, ..]]`*) – Trial states to filter on. If `None`, include all states.

Returns A list of `FrozenMultiObjectiveTrial` objects.

Return type List[*optuna.multi_objective.trial.FrozenMultiObjectiveTrial*]

`property n_objectives`

Return the number of objectives.

Returns Number of objectives.

`optimize`(*objective*, *timeout=None*, *n_trials=None*, *n_jobs=1*, *catch=()*, *callbacks=None*, *gc_after_trial=True*, *show_progress_bar=False*)

Optimize an objective function.

This method is the same as `optuna.study.Study.optimize()` except for taking an objective function that returns multi-objective values as the argument.

Please refer to the documentation of `optuna.study.Study.optimize()` for further details.

Example

```
import optuna


def objective(trial):
    # Binh and Korn function.
    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.multi_objective.create_study(["minimize", "minimize"])
study.optimize(objective, n_trials=3)
```

Parameters

- **`objective`** (*`Callable[[optuna.multi_objective.trial.MultiObjectiveTrial], Sequence[float]]`*) –
- **`timeout`** (*`Optional[int]`*) –
- **`n_trials`** (*`Optional[int]`*) –
- **`n_jobs`** (*`int`*) –
- **`catch`** (*`Tuple[Type[Exception], ..]`*) –
- **`callbacks`** (*`Optional[List[Callable[[optuna.multi_objective.study.MultiObjectiveStudy, optuna.multi_objective.trial.FrozenMultiObjectiveTrial], None]]]`*) –
- **`gc_after_trial`** (*`bool`*) –
- **`show_progress_bar`** (*`bool`*) –

Return type None

property sampler
Return the sampler.

Returns A `BaseMultiObjectiveSampler` object.

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

set_user_attr(*key*, *value*)
Set a user attribute to the study.

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

property system_attrs
Return system attributes.

Returns A dictionary containing all system attributes.

property trials
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 *`FrozenMultiObjectiveTrial`* objects.

property user_attrs
Return user attributes.

Returns A dictionary containing all user attributes.

optuna.multi_objective.study.create_study

`optuna.multi_objective.study.`**create_study**(*directions*, *study_name=None*, *storage=None*, *sampler=None*, *load_if_exists=False*)
Create a new *`MultiObjectiveStudy`*.

Example

```
import optuna


def objective(trial):
    # Binh and Korn function.
    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.multi_objective.create_study(["minimize", "minimize"])
study.optimize(objective, n_trials=3)
```

Parameters

- **directions** (*List[str]*) – Optimization direction for each objective value. Set `minimize` for minimization and `maximize` for maximization.
- **study_name** (*Optional[str]*) – Study's name. If this argument is set to None, a unique name is generated automatically.
- **storage** (*Optional[Union[str, optuna.storages._base.BaseStorage]]*) – 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** (*Optional[optuna.multi_objective.samplers._base.BaseMultiObjectiveSampler]*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *NSGAIIMultiObjectiveSampler* is used as the default. See also *samplers*.
- **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.

Returns A *MultiObjectiveStudy* object.

Return type *optuna.multi_objective.study.MultiObjectiveStudy*

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.

optuna.multi_objective.study.load_study

`optuna.multi_objective.study.`**`load_study`**(*study_name*, *storage*, *sampler=None*)

Load the existing *`MultiObjectiveStudy`* that has the specified name.

Example

```
import optuna


def objective(trial):
    # Binh and Korn function.
    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.multi_objective.create_study(
    directions=["minimize", "minimize"],
    study_name="my_study",
    storage="sqlite:///example.db",
)
study.optimize(objective, n_trials=3)

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

Parameters

- **`study_name`** (*`str`*) – Study's name. Each study has a unique name as an identifier.
- **`storage`** (*`Union[str, optuna.storages._base.BaseStorage]`*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *`create_study()`* for further details.
- **`sampler`** (*`Optional[optuna.multi_objective.samplers._base.BaseMultiObjectiveSampler]`*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *`RandomMultiObjectiveSampler`* is used as the default. See also *`samplers`*.

Returns A *`MultiObjectiveStudy`* object.

Return type *optuna.multi_objective.study.MultiObjectiveStudy*

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.

optuna.multi_objective.trial

`optuna.multi_objective.trial.MultiObjectiveTrial`	A trial is a process of evaluating an objective function.
`optuna.multi_objective.trial.FrozenMultiObjectiveTrial`	Status and results of a `MultiObjectiveTrial`.

optuna.multi_objective.trial.MultiObjectiveTrial

class `optuna.multi_objective.trial.`**`MultiObjectiveTrial`**(*trial*)

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.multi_objective.study.MultiObjectiveStudy.optimize()` method; hence library users do not care about instantiation of this object.

Parameters **`trial`** – A `Trial` object.

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

`report`(values, step)	Report intermediate objective function values 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.
`suggest_categorical`(name, choices)	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.
`system_attrs`	Return system attributes.

continues on next page

Table 63 – continued from previous page

user_attrs	Return user attributes.

property datetime_start

Return start datetime.

Returns Datetime where the *Trial* started.

property distributions

Return distributions of parameters to be optimized.

Returns A dictionary containing all distributions.

property number

Return trial's number which is consecutive and unique in a study.

Returns A trial number.

property params

Return parameters to be optimized.

Returns A dictionary containing all parameters.

report(*values*, *step*)

Report intermediate objective function values 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 values are 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.

Parameters

- **values** (*Sequence[float]*) – Intermediate objective function values for a given step.
- **step** (*int*) – Step of the trial (e.g., Epoch of neural network training).

Return type None

set_system_attr(*key*, *value*)

Set system attributes to the trial.

Please refer to the documentation of *optuna.trial.Trial.set_system_attr()* for further details.

Parameters

- **key** (*str*) –
- **value** (*Any*) –

Return type None

set_user_attr(*key*, *value*)

Set user attributes to the trial.

Please refer to the documentation of *optuna.trial.Trial.set_user_attr()* for further details.

Parameters

- **key** (*str*) –
- **value** (*Any*) –

Return type None

suggest_categorical (*name*, *choices*)

Suggest a value for the categorical parameter.

Please refer to the documentation of `optuna.trial.Trial.suggest_categorical()` for further details.

Parameters

- **name** (*str*) –
- **choices** (*Sequence[Optional[Union[str, float, int, bool]]]*) –

Return type Union[None, bool, int, float, str]

suggest_discrete_uniform (*name*, *low*, *high*, *q*)

Suggest a value for the discrete parameter.

Please refer to the documentation of `optuna.trial.Trial.suggest_discrete_uniform()` for further details.

Parameters

- **name** (*str*) –
- **low** (*float*) –
- **high** (*float*) –
- **q** (*float*) –

Return type float

suggest_float (*name*, *low*, *high*, ***, *step=None*, *log=False*)

Suggest a value for the floating point parameter.

Please refer to the documentation of `optuna.trial.Trial.suggest_float()` for further details.

Parameters

- **name** (*str*) –
- **low** (*float*) –
- **high** (*float*) –
- **step** (*Optional[float]*) –
- **log** (*bool*) –

Return type float

suggest_int (*name*, *low*, *high*, *step=1*, *log=False*)

Suggest a value for the integer parameter.

Please refer to the documentation of `optuna.trial.Trial.suggest_int()` for further details.

Parameters

- **name** (*str*) –
- **low** (*int*) –

- **high** (*int*) –
- **step** (*int*) –
- **log** (*bool*) –

Return type int

suggest_loguniform(*name*, *low*, *high*)

Suggest a value for the continuous parameter.

Please refer to the documentation of *optuna.trial.Trial.suggest_loguniform()* for further details.

Parameters

- **name** (*str*) –
- **low** (*float*) –
- **high** (*float*) –

Return type float

suggest_uniform(*name*, *low*, *high*)

Suggest a value for the continuous parameter.

Please refer to the documentation of *optuna.trial.Trial.suggest_uniform()* for further details.

Parameters

- **name** (*str*) –
- **low** (*float*) –
- **high** (*float*) –

Return type float

property system_attrs

Return system attributes.

Returns A dictionary containing all system attributes.

property user_attrs

Return user attributes.

Returns A dictionary containing all user attributes.

optuna.multi_objective.trial.FrozenMultiObjectiveTrial

class optuna.multi_objective.trial.**FrozenMultiObjectiveTrial**(*n_objectives*, *trial*)

Status and results of a *MultiObjectiveTrial*.

number

Unique and consecutive number of *MultiObjectiveTrial* for each *MultiObjectiveStudy*. Note that this field uses zero-based numbering.

state

TrialState of the *MultiObjectiveTrial*.

values

Objective values of the *MultiObjectiveTrial*.

datetime_start
Datetime where the *MultiObjectiveTrial* started.

datetime_complete
Datetime where the *MultiObjectiveTrial* finished.

params
Dictionary that contains suggested parameters.

distributions
Dictionary that contains the distributions of *params*.

user_attrs
Dictionary that contains the attributes of the *MultiObjectiveTrial* set with *optuna.multi_objective.trial.MultiObjectiveTrial.set_user_attr()*.

intermediate_values
Intermediate objective values set with *optuna.multi_objective.trial.MultiObjectiveTrial.report()*.

> **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.

Attributes

datetime_complete
datetime_start
distributions
`last_step`
number
params
state
`system_attrs`
user_attrs

optuna.multi_objective.visualization

Note: *`optuna.multi_objective.visualization`* module uses plotly to create figures, but JupyterLab cannot render them by default. Please follow this installation guide to show figures in JupyterLab.

`optuna.multi_objective.visualization.plot_pareto_front`	Plot the pareto front of a study.

optuna.multi_objective.visualization.plot_pareto_front

`optuna.multi_objective.visualization.`**`plot_pareto_front`**(*study*, *names=None*, *include_dominated_trials=False*, *axis_order=None*)

Plot the pareto front of a study.

Example

The following code snippet shows how to plot the pareto front of a study.

```
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.multi_objective.create_study(["minimize", "minimize"])
study.optimize(objective, n_trials=50)

fig = optuna.multi_objective.visualization.plot_pareto_front(study)
fig.show()
```

Parameters

- **`study`** (`optuna.multi_objective.study.MultiObjectiveStudy`) – A *`MultiObjectiveStudy`* object whose trials are plotted for their objective values.
- **`names`** (*`Optional[List[str]]`*) – Objective name list used as the axis titles. If `None` is specified, “Objective {objective_index}” is used instead.
- **`include_dominated_trials`** (*`bool`*) – A flag to include all dominated trial’s objective values.
- **`axis_order`** (*`Optional[List[int]]`*) – A list of indices indicating the axis order. If `None` is specified, default order is used.

Returns A `plotly.graph_objs.Figure` object.

Raises **`ValueError`** – If the number of objectives of `study` isn’t 2 or 3.

Return type plotly.graph_objs._figure.Figure

> **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.

6.3.9 optuna.pruners

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

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

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`** – Pruning is disabled until the given number of trials finish in the same study.
- **`n_warmup_steps`** – Pruning is disabled until the trial exceeds the given number of step. Note that this feature assumes that `step` starts at zero.

- **`interval_steps`** – 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.

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

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

(continues on next page)

(continued from previous page)

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

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

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

import optuna

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


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

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

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

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

    return clf.score(X_valid, y_valid)


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

Parameters

- **`percentile`** – 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`** – Pruning is disabled until the given number of trials finish in the same study.
- **`n_warmup_steps`** – Pruning is disabled until the trial exceeds the given number of step. Note that this feature assumes that `step` starts at zero.
- **`interval_steps`** – 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** (`optuna.study.Study`) – Study object of the target study.
- **trial** (*optuna.trial._frozen.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)

```

(continues on next page)

(continued from previous page)

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

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

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

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

    return clf.score(X_valid, y_valid)


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

Parameters

- **`min_resource`** – 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`** – 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`** – A parameter for specifying the minimum early-stopping rate (in the paper this parameter is referred to as s).
- **`bootstrap_count`** – 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** (`optuna.study.Study`) – Study object of the target study.
- **trial** (*optuna.trial._frozen.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 to set 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 collect 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*. Each *SuccessiveHalvingPruner* is referred as "bracket" in the original paper. The number of brackets is an important factor to control the early stopping behavior of Hyperband and is automatically determined by `min_resource`, `max_resource` and `reduction_factor` as *The number of brackets = floor(log_{reduction_factor}(max_resource /*

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.

See also:

Please refer to *`report()`*.

Example

We minimize an objective function with Hyperband pruning algorithm.

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

import optuna

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


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

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

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

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

    return clf.score(X_valid, y_valid)


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`** – 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`** – A parameter for specifying the maximum resource allocated to a trial. R in the paper corresponds to `max_resource / min_resource`. This value repre-

sents 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`** – A parameter for specifying reduction factor of promotable trials noted as η in the paper. See the details for `SuccessiveHalvingPruner`.
- **`bootstrap_count`** – Parameter specifying the number of trials required in a rung before any trial can be promoted. Incompatible with `max_resouce` 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`** (`optuna.study.Study`) – Study object of the target study.
- **`trial`** (`optuna.trial._frozen.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

```
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)
```

Args

lower: A minimum value which determines whether pruner prunes or not. If an intermediate value is smaller than lower, it prunes.

upper: A maximum value which determines whether pruner prunes or not. If an intermediate value is larger than upper, it prunes.

n_warmup_steps: Pruning is disabled if the step is less than the given number of warmup steps.

interval_steps: 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** (optuna.study.Study) – Study object of the target study.
- **trial** (*optuna.trial._frozen.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

6.3.10 optuna.samplers

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

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.IntersectionSearchSpace	A class to calculate the intersection search space of a `BaseStudy`.
optuna.samplers.intersection_search_space	Return the intersection search space of the `BaseStudy`.

optuna.samplers.BaseSampler

class optuna.samplers.**BaseSampler**

Base class for samplers.

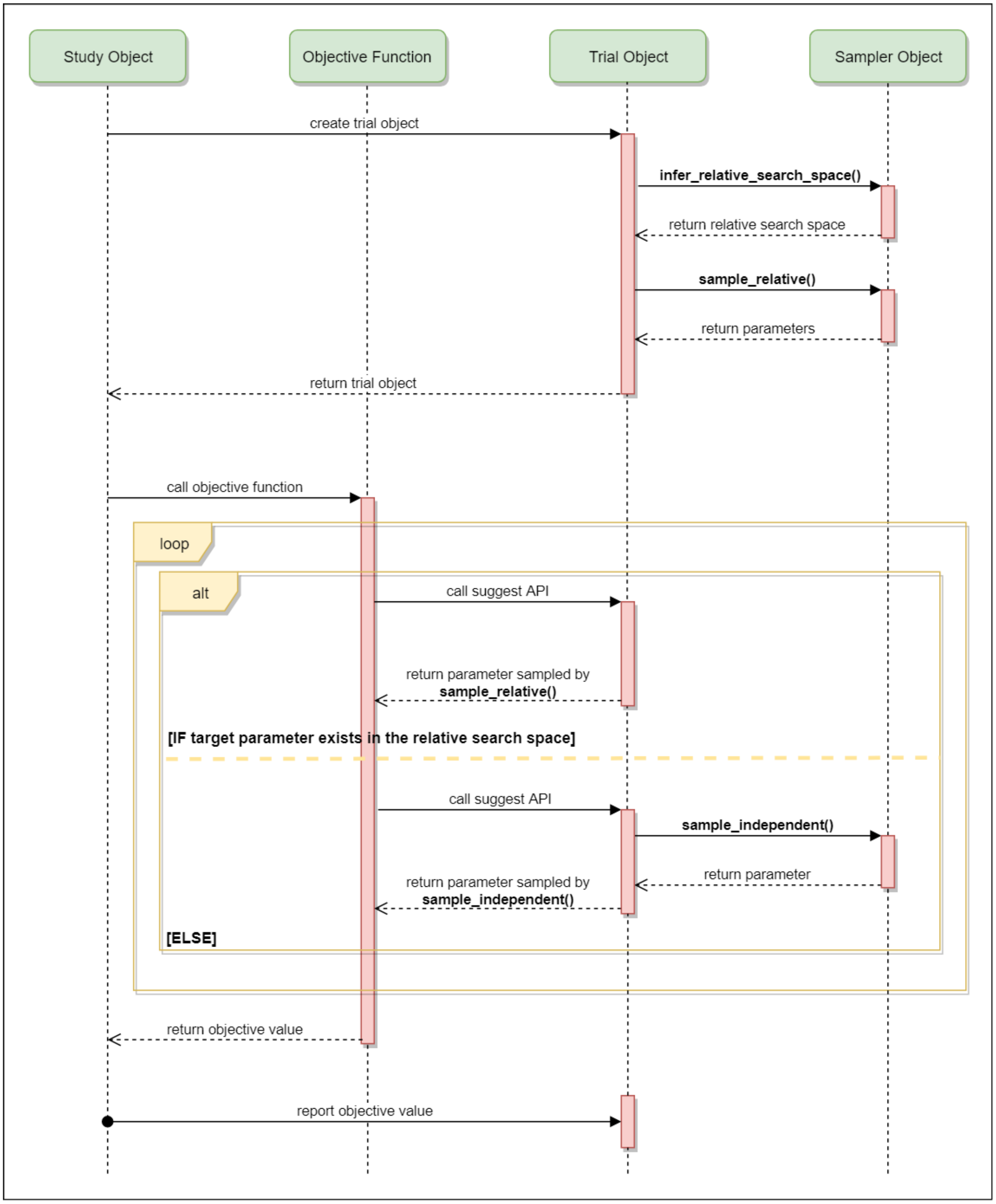
Optuna combines two types of sampling strategies, which are called *relative sampling* and *independent sampling*.

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. Then, `sample_relative()` is invoked to sample parameters from the relative 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.

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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.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, optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **param_name** (*str*) – Name of the sampled parameter.
- **param_distribution** (*optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **search_space** (*Dict[str, optuna.distributions.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*)

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: The grid is randomly shuffled and the order in which parameter configurations are suggested may vary.

This is to reduce duplicate suggestions during distributed optimization.

Parameters **search_space** – A dictionary whose key and value are a parameter name and the corresponding candidates of values, respectively.

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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.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, optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.

- **search_space** (*Dict[str, optuna.distributions.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** – 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 trials 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`state`** (*`optuna.trial._state.TrialState`*) – Resulting trial state.
- **`values`** (*`Optional[Sequence[float]]`*) – 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.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, optuna.distributions.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`** (`optuna.study.Study`) – Target study object.

- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`search_space`** (*`Dict[str, optuna.distributions.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*, *warn_independent_sampling=True*)

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

Example

```
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`** – Enhance the stability of Parzen estimator by imposing a Gaussian prior when `True`. The prior is only effective if the sampling distribution is either *`UniformDistribution`*, *`DiscreteUniformDistribution`*, *`LogUniformDistribution`*, *`IntUniformDistribution`*, or *`IntLogUniformDistribution`*.
- **`prior_weight`** – The weight of the prior. This argument is used in *`UniformDistribution`*, *`DiscreteUniformDistribution`*, *`LogUniformDistribution`*, *`IntUniformDistribution`*, *`IntLogUniformDistribution`*, and *`CategoricalDistribution`*.
- **`consider_magic_clip`** – Enable a heuristic to limit the smallest variances of Gaussians used in the Parzen estimator.
- **`consider_endpoints`** – 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`** – The random sampling is used instead of the TPE algorithm until the given number of trials finish in the same study.
- **`n_ei_candidates`** – Number of candidate samples used to calculate the expected improvement.
- **`gamma`** – 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`** – 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.
- **`seed`** – Seed for random number generator.
- **`multivariate`** – 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.

- **`warn_independent_sampling`** – 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.

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 trials 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`state`** (*`optuna.trial._state.TrialState`*) – Resulting trial state.
- **`values`** (*`Optional[Sequence[float]]`*) – 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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.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, optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`search_space`** (*`Dict[str, optuna.distributions.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*, *inc_popsize=2*, *use_separable_cma=False*, *source_trials=None*)

A sampler using cmaes as the backend.

Example

Optimize a simple quadratic function by using *`CmaEsSampler`*.

```
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. 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.

See also:

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

Parameters

- **`x0`** – 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`** – 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`** – A random seed for CMA-ES.
- **`n_startup_trials`** – The independent sampling is used instead of the CMA-ES algorithm until the given number of trials finish in the same study.
- **`independent_sampler`** – 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`** – 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`** – 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.

- **`inc_popsize`** – Multiplier for increasing population size before each restart. This argument will be used when setting `restart_strategy = 'ipop'`.

- **`consider_pruned_trials`** – 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`** – 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.

- **`source_trials`** – 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.

Raises **`ValueError`** – If `restart_strategy` is not 'ipop' or `None`.

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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.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, optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`search_space`** (*`Dict[str, optuna.distributions.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


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`** – A dictionary of parameters to be fixed.
- **`base_sampler`** – 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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **state** (*`optuna.trial._state.TrialState`*) – Resulting trial state.
- **values** (*`Optional[Sequence[float]]`*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.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, optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`param_name`** (*`str`*) – Name of the sampled parameter.
- **`param_distribution`** (*`optuna.distributions.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`** (`optuna.study.Study`) – Target study object.
- **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **`search_space`** (*`Dict[str, optuna.distributions.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_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`** – Number of individuals (trials) in a generation.
- **`mutation_prob`** – 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_prob`** – Probability that a crossover (parameters swapping between parents) will occur when creating a new individual.
- **`swapping_prob`** – Probability of swapping each parameter of the parents during crossover.
- **`seed`** – Seed for random number generator.
- **`constraints_func`** – 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 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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **state** (*`optuna.trial._state.TrialState`*) – Resulting trial state.
- **values** (*`Optional[Sequence[float]]`*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.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, optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.

- **param_name** (`str`) – Name of the sampled parameter.
- **param_distribution** (*`optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **search_space** (*`Dict[str, optuna.distributions.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

Parameters

- **consider_prior** – Enhance the stability of Parzen estimator by imposing a Gaussian prior when `True`. The prior is only effective if the sampling distribution is either *`UniformDistribution`*, *`DiscreteUniformDistribution`*, *`LogUniformDistribution`*, *`IntUniformDistribution`*, or *`IntLogUniformDistribution`*.

- **`prior_weight`** – The weight of the prior. This argument is used in `UniformDistribution`, `DiscreteUniformDistribution`, `LogUniformDistribution`, `IntUniformDistribution`, `IntLogUniformDistribution`, and `CategoricalDistribution`.
- **`consider_magic_clip`** – Enable a heuristic to limit the smallest variances of Gaussians used in the Parzen estimator.
- **`consider_endpoints`** – 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`** – 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`** – Number of candidate samples used to calculate the expected hypervolume improvement.
- **`gamma`** – 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`** – 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`** – 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

```
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


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)
```

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.
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 trials 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** (`optuna.study.Study`) – Target study object.
- **trial** (*optuna.trial._frozen.FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*optuna.trial._state.TrialState*) – Resulting trial state.
- **values** (*Optional[Sequence[float]]*) – 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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.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, optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **param_name** (*`str`*) – Name of the sampled parameter.
- **param_distribution** (*`optuna.distributions.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** (`optuna.study.Study`) – Target study object.
- **trial** (*`optuna.trial._frozen.FrozenTrial`*) – Target trial object. Take a copy before modifying this object.
- **search_space** (*`Dict[str, optuna.distributions.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 `BaseStudy`.

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`** – Whether pruned trials should be included in the search space.

Methods

`calculate`(study[, ordered_dict])	Returns the intersection search space of the `BaseStudy`.

`calculate` (*study*, *ordered_dict=False*)

Returns the intersection search space of the `BaseStudy`.

Parameters

- **`study`** (*`optuna.study.BaseStudy`*) – 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.

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

Raises **`ValueError`** – If different studies are passed into this method.

Return type Dict[str, optuna.distributions.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 `BaseStudy`.

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`** (*`optuna.study.BaseStudy`*) – 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, optuna.distributions.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.RedisStorage	Storage class for Redis 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*)

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** – URL of the storage.
- **engine_kwargs** – A dictionary of keyword arguments that is passed to sqlalchemy.engine.create_engine function.
- **skip_compatibility_check** – Flag to skip schema compatibility check if set to True.
- **heartbeat_interval** – Interval to record the heartbeat. It is recorded every `interval` seconds.
- **grace_period** – Grace period before a running trial is failed from the last heartbeat. If it is `None`, the grace period will be *2 * heartbeat_interval*.
- **failed_trial_callback** – 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.

> **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.

Raises

- **`ValueError`** – If the given *heartbeat_interval* or *grace_period* is not a positive integer.
- **`RuntimeError`** – If the a process that was failed by heartbeat but was actually running.

Methods

`check_trial_is_updatable`(trial_id, trial_state)	Check whether a trial state is updatable.
`create_new_study`([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.
`fail_stale_trials`(study_id)	Fail stale trials.
`get_all_study_summaries`()	Read a list of `StudySummary` 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_id_from_trial_id`(trial_id)	Read the ID of a study to which a trial belongs.
`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.
`is_heartbeat_enabled`()	Check whether the storage enables the heartbeat.
`read_trials_from_remote_storage`(study_id)	Make an internal cache of trials up-to-date.

continues on next page

Table 85 – continued from previous page

record_heartbeat(trial_id)	Record the heartbeat of the trial.
remove_session()	Removes the current session.
set_study_directions(study_id, directions)	Register optimization problem directions to a study.
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(trial_id, state)	Update the state 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.
set_trial_values(trial_id, values)	Set return values of an objective function.
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** (*optuna.trial._state.TrialState*) – Trial state to check.

Raises **RuntimeError** – If the trial is already finished.

Return type None

create_new_study(*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 **study_name** (*Optional[str]*) – 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** (*Optional[optuna.trial._frozen.FrozenTrial]*) – Template `FronzenTrial` 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

fail_stale_trials(*study_id*)
Fail stale trials.

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*. The grace period is `2 * heartbeat_interval`.

Parameters **study_id** (*int*) – ID of the related study.

Returns List of trial IDs of the failed trials.

Return type List[int]

get_all_study_summaries()
Read a list of *StudySummary* objects.

Returns A list of *StudySummary* objects.

Return type List[optuna._study_summary.StudySummary]

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** (*Optional[Tuple[optuna.trial._state.TrialState, ..]]*) – Trial states to filter on. If `None`, include all states.

Returns List of trials in the study.

Raises **KeyError** – If no study with the matching `study_id` exists.

Return type List[optuna.trial._frozen.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 optuna.trial._frozen.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 Optional[Callable[[*optuna.study.Study*, optuna.trial._frozen.FrozenTrial], 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 Optional[int]

get_n_trials(*study_id*, *state=None*)

Count the number of trials in a study.

Parameters

- **study_id** (*int*) – ID of the study.
- **state** (*Optional[Union[Tuple[optuna.trial._state.TrialState, ...], optuna.trial._state.TrialState]]*) – 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[optuna._study_direction.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_id_from_trial_id(*trial_id*)

Read the ID of a study to which a trial belongs.

Parameters **`trial_id`** (*`int`*) – ID of the trial.

Returns ID of the study.

Raises **`KeyError`** – If no trial with the matching `trial_id` 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 optuna.trial._frozen.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]

is_heartbeat_enabled()

Check whether the storage enables the heartbeat.

Returns `True` if the storage supports the heartbeat and the return value of `get_heartbeat_interval()` is an integer, otherwise `False`.

Return type bool

`read_trials_from_remote_storage`(*study_id*)

Make an internal cache of trials up-to-date.

Parameters **`study_id`** (*`int`*) – ID of the study.

Raises **`KeyError`** – If no study with the matching `study_id` exists.

Return type None

`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_directions`(*study_id*, *directions*)

Register optimization problem directions to a study.

Parameters

- **`study_id`** (*`int`*) – ID of the study.
- **`directions`** (*`Sequence[optuna._study_direction.StudyDirection]`*) – A sequence of direction whose element is either *`MAXIMIZE`* or *`MINIMIZE`*.

Raises

- **`KeyError`** – If no study with the matching `study_id` exists.
- **`ValueError`** – If the directions are already set and the each coordinate of passed `directions` is the opposite direction or *`NOT_SET`*.

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`** (*`optuna.distributions.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`(*trial_id*, *state*)

Update the state of a trial.

Parameters

- **`trial_id`** (*`int`*) – ID of the trial.
- **`state`** (*`optuna.trial._state.TrialState`*) – New state of the trial.

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** (*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

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

set_trial_values(*trial_id*, *values*)
Set return values of an objective function.

This method overwrites any existing trial values.

Parameters

- **trial_id** (*int*) – ID of the trial.
- **values** (*Sequence[float]*) – Values of the objective function.

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.RedisStorage

`class` `optuna.storages.`**`RedisStorage`**`(`*url*`)`

Storage class for Redis 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

We create an `RedisStorage` instance using the given redis database URL.

```
import optuna


def objective(trial):
    ...


storage = optuna.storages.RedisStorage(
    url="redis://passwd@localhost:port/db",
)

study = optuna.create_study(storage=storage)
study.optimize(objective)
```

Parameters **`url`** – URL of the redis storage, password and db are optional. (ie: redis://localhost:6379)

Note: If you use plan to use Redis as a storage mechanism for optuna, make sure Redis in installed and running. Please execute `$ pip install -U redis` to install redis python library.

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

`check_trial_is_updatable`(trial_id, trial_state)	Check whether a trial state is updatable.
`create_new_study`([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.
`fail_stale_trials`(study_id)	Fail stale trials.
`get_all_study_summaries`()	Read a list of `StudySummary` 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_failed_trial_callback`()	Get the failed trial callback function.
`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_id_from_trial_id`(trial_id)	Read the ID of a study to which a trial belongs.
`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.
`is_heartbeat_enabled`()	Check whether the storage enables the heartbeat.
`read_trials_from_remote_storage`(study_id)	Make an internal cache of trials up-to-date.
`record_heartbeat`(trial_id)	Record the heartbeat of the trial.
`remove_session`()	Clean up all connections to a database.
`set_study_directions`(study_id, directions)	Register optimization problem directions to a study.
`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`(trial_id, state)	Update the state 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.
`set_trial_values`(trial_id, values)	Set return values of an objective function.

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** (*optuna.trial._state.TrialState*) – Trial state to check.

Raises `RuntimeError` – If the trial is already finished.

Return type None

`create_new_study`(*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 **`study_name`** (*`Optional[str]`*) – 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`** (*`Optional[optuna.trial._frozen.FrozenTrial]`*) – Template `FronzenTrial` 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

`fail_stale_trials`(*study_id*)

Fail stale trials.

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`*. The grace period is `2 * heartbeat_interval`.

Parameters **`study_id`** (*`int`*) – ID of the related study.

Returns List of trial IDs of the failed trials.

Return type List[int]

`get_all_study_summaries`()

Read a list of *`StudySummary`* objects.

Returns A list of *`StudySummary`* objects.

Return type List[optuna._study_summary.StudySummary]

`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** (*Optional[Tuple[optuna.trial._state.TrialState, ..]]*) – Trial states to filter on. If `None`, include all states.

Returns List of trials in the study.

Raises **KeyError** – If no study with the matching `study_id` exists.

Return type List[optuna.trial._frozen.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 optuna.trial._frozen.FrozenTrial

get_failed_trial_callback()

Get the failed trial callback function.

Returns The failed trial callback function if it is set, otherwise `None`.

Return type Optional[Callable[[*optuna.study.Study*, optuna.trial._frozen.FrozenTrial], None]]

get_heartbeat_interval()

Get the heartbeat interval if it is set.

Returns The heartbeat interval if it is set, otherwise `None`.

Return type Optional[int]

get_n_trials(*study_id*, *state=None*)

Count the number of trials in a study.

Parameters

- **study_id** (*int*) – ID of the study.
- **state** (*Optional[Union[Tuple[optuna.trial._state.TrialState, ..], optuna.trial._state.TrialState]]*) – 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[optuna._study_direction.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_id_from_trial_id`(*trial_id*)
Read the ID of a study to which a trial belongs.

Parameters **`trial_id`** (*`int`*) – ID of the trial.

Returns ID of the study.

Raises **`KeyError`** – If no trial with the matching `trial_id` 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 optuna.trial._frozen.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]

`is_heartbeat_enabled`()
Check whether the storage enables the heartbeat.

Returns `True` if the storage supports the heartbeat and the return value of `get_heartbeat_interval()` is an integer, otherwise `False`.

Return type bool

`read_trials_from_remote_storage`(*study_id*)
Make an internal cache of trials up-to-date.

Parameters **`study_id`** (*`int`*) – ID of the study.

Raises `KeyError` – If no study with the matching `study_id` exists.

Return type None

`record_heartbeat`(*trial_id*)
Record the heartbeat of the trial.

Parameters **`trial_id`** (*`int`*) – ID of the trial.

Return type None

`remove_session`()
Clean up all connections to a database.

Return type None

`set_study_directions`(*study_id*, *directions*)
Register optimization problem directions to a study.

Parameters

- **`study_id`** (*`int`*) – ID of the study.
- **`directions`** (*`Sequence[optuna._study_direction.StudyDirection]`*) – A sequence of direction whose element is either *`MAXIMIZE`* or *`MINIMIZE`*.

Raises

- `KeyError` – If no study with the matching `study_id` exists.
- `ValueError` – If the directions are already set and the each coordinate of passed `directions` is the opposite direction or *`NOT_SET`*.

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`** (*`optuna.distributions.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`(*trial_id*, *state*)

Update the state of a trial.

Parameters

- **`trial_id`** (*`int`*) – ID of the trial.
- **`state`** (*`optuna.trial._state.TrialState`*) – New state of the trial.

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`** (*`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

`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

`set_trial_values`(*trial_id*, *values*)

Set return values of an objective function.

This method overwrites any existing trial values.

Parameters

- **`trial_id`** (*`int`*) – ID of the trial.
- **`values`** (*`Sequence[float]`*) – Values of the objective function.

Raises

- **`KeyError`** – If no trial with the matching `trial_id` exists.
- **`RuntimeError`** – If the trial is already finished.

Return type None

6.3.12 optuna.structs

This module is deprecated, with former functionality moved to *`optuna.trial`* and *`optuna.study`*.

`class` `optuna.structs.`**`TrialState`**(*value*)

State of a *`Trial`*.

`RUNNING`

The *`Trial`* is running.

`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.

Deprecated since version 1.4.0: This class is deprecated. Please use *`TrialState`* instead.

`class` `optuna.structs.`**`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.

Deprecated since version 1.4.0: This class is deprecated. Please use *`StudyDirection`* instead.

`class` `optuna.structs.`**`FrozenTrial`**(*number*, *state*, *value*, *datetime_start*, *datetime_complete*, *params*, *distributions*, *user_attrs*, *system_attrs*, *intermediate_values*, *trial_id*, ***, *values=None*)

> **Warning:** Deprecated in v1.4.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v3.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v1.4.0.
>
> This class was moved to *`trial`*. Please use *`FrozenTrial`* instead.

`property distributions`

Dictionary that contains the distributions of `params`.

property duration
Return the elapsed time taken to complete the trial.

Returns The duration.

property last_step
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

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

class optuna.structs.**StudySummary**(*study_name*, *direction*, *best_trial*, *user_attrs*, *system_attrs*, *n_trials*, *datetime_start*, *study_id*, ***, *directions=None*)

Warning: Deprecated in v1.4.0. This feature will be removed in the future. The removal of this feature is currently scheduled for v3.0.0, but this schedule is subject to change. See https://github.com/optuna/optuna/releases/tag/v1.4.0.

This class was moved to `study`. Please use `StudySummary` instead.

6.3.13 optuna.study

The `study` module implements the `Study` object and related functions. A public constructor is available for the `Study` class, but direct use of this constructor is not recommended. Instead, library users should create and load a `Study` using `create_study()` and `load_study()` respectively.

`optuna.study.Study`	A study corresponds to an optimization task, i.e., a set of trials.
`optuna.study.create_study`	Create a new `Study`.
`optuna.study.load_study`	Load the existing `Study` that has the specified name.
`optuna.study.delete_study`	Delete a `Study` object.
`optuna.study.get_all_study_summaries`	Get all history of studies stored in a specified storage.
`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)	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_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])	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.

add_trial(*trial*)

Add trial to study.

The trial is validated before being added.

Example

```
import optuna
from optuna.distributions import UniformDistribution


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": UniformDistribution(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

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.

Parameters **`trial`** (*`optuna.trial._frozen.FrozenTrial`*) – Trial to add.

Raises **`ValueError`** – If trial is an invalid state.

Return type None

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.

`add_trials`(*trials*)

Add trials to study.

The trials are validated before being added.

Example

```
import optuna
from optuna.distributions import UniformDistribution


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[optuna.trial._frozen.FrozenTrial]`*) – Trials to add.

Raises **`ValueError`** – If `trials` include invalid trial.

Return type None

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.

---

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.LogUniformDistribution(0.0001, 0.1),
}

# You can pass the distributions previously defined.
trial = study.ask(fixed_distributions=distributions)

# `optimizer` and `lr` are already suggested and accessible with `trial.
↪params`.
assert "optimizer" in trial.params
assert "lr" in trial.params
```

Parameters **fixed_distributions** (*Optional[Dict[str, optuna.distributions.BaseDistribution]]*) – 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 optuna.trial._trial.Trial

property best_params
Return parameters of the best trial in the study.

Returns A dictionary containing parameters of the best trial.

Raises `RuntimeError` – If the study has more than one direction.

property best_trial
Return the best trial in the study.

Returns A `FrozenTrial` object of the best trial.

Raises `RuntimeError` – If the study has more than one direction.

property best_trials
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
Return the best objective value in the study.

Returns A float representing the best objective value.

Raises `RuntimeError` – If the study has more than one direction.

property direction
Return the direction of the study.

Returns A `StudyDirection` object.

Raises `RuntimeError` – If the study has more than one direction.

property directions
Return the directions of the study.

Returns A list of `StudyDirection` objects.

enqueue_trial(*params*)
Enqueue a trial with given parameter values.

You can fix the next sampling parameters which will be evaluated in your objective function.

Example

```python
import optuna


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


study = optuna.create_study()
study.enqueue_trial({"x": 5})
study.enqueue_trial({"x": 0})
```

(continues on next page)

(continued from previous page)

```
study.optimize(objective, n_trials=2)

assert study.trials[0].params == {"x": 5}
assert study.trials[1].params == {"x": 0}
```

Parameters **`params`** (*`Dict[str, Any]`*) – Parameter values to pass your objective function.

Return type None

Note: Added in v1.2.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v1.2.0.

`get_trials`(*deepcopy=True*, *states=None*)

Return all trials in the study.

The returned trials are ordered by trial number.

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`** (*`Optional[Tuple[optuna.trial._state.TrialState, ...]]`*) – Trial states to filter on. If `None`, include all states.

Returns A list of `FrozenTrial` objects.

Return type List[optuna.trial._frozen.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'.

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[[optuna.trial._trial.Trial], Union[float, Sequence[float]]]`*) – A callable that implements objective function.
- **`n_trials`** (*`Optional[int]`*) – The number of trials. If this argument is set to `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.
- **`timeout`** (*`Optional[float]`*) – Stop study after the given number of second(s). If this argument is set to `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.
- **`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.

  Warning: Deprecated in v2.7.0. This feature will be removed in the future. It is recommended to use *process-based parallelization*. 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.

- **`catch`** (*`Tuple[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`** (*`Optional[List[Callable[[optuna.study.Study, optuna.trial._frozen.FrozenTrial], 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`.
- **`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_jobs` $\neq 1$.

Raises **`RuntimeError`** – If nested invocation of this method occurs.

Return type None

`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

`set_user_attr`(*key*, *value*)

Set a user attribute to the study.

See also:

See *`user_attrs`* for related attribute.

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"])

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.

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
```

Raises `RuntimeError` – If this method is called outside an objective function or callback.

Return type None

property system_attrs

Return system attributes.

Returns A dictionary containing all system attributes.

tell(*trial*, *values=None*, *state=TrialState.COMPLETE*)

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):
```

(continues on next page)

(continued from previous page)

```
    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** (*Union[optuna.trial._trial.Trial, int]*) – A `Trial` object or a trial number.
- **values** (*Optional[Union[float, Sequence[float]]]*) – 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** (*optuna.trial._state.TrialState*) – State to be reported. Must be `COMPLETE`, `FAIL` or `PRUNED`.

Raises

- **TypeError** – If `trial` is not a `Trial` or an `int`.
- **ValueError** – If any of the following. `values` is a sequence but its length does not match the number of objectives for its associated study. `state` is `COMPLETE` but `values` is `None`. `state` is `FAIL` or `PRUNED` but `values` is not `None`. `state` is not `COMPLETE`, `FAIL` or `PRUNED`. `trial` is a trial number but no trial exists with that number.

Return type None

property trials

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.

`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)

# 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 pandas.core.frame.DataFrame

Note: If `value` is in `attrs` during multi-objective optimization, it is implicitly replaced with `values`.

`property user_attrs`

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"])

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`** (*`Optional[Union[str, optuna.storages._base.BaseStorage]]`*) – 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** (*Optional[optuna.samplers._base.BaseSampler]*) – 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** (*Optional[optuna.pruners._base.BasePruner]*) – 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** (*Optional[str]*) – Study's name. If this argument is set to None, a unique name is generated automatically.
- **direction** (*Optional[Union[str, optuna._study_direction.StudyDirection]]*) – Direction of optimization. Set `minimize` for minimization and `maximize` for maximization. You can also pass the corresponding *StudyDirection* object.

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** (*Optional[Sequence[Union[str, optuna._study_direction.StudyDirection]]]*) – A sequence of directions during multi-objective optimization.

Returns A *Study* object.

Raises **ValueError** – If the length of `directions` is zero. Or, if `direction` is neither 'minimize' nor 'maximize' when it is a string. Or, if the element of `directions` is neither *minimize* nor *maximize*. Or, if both `direction` and `directions` are specified.

Return type *optuna.study.Study*

See also:

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

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`*) – Study's name. Each study has a unique name as an identifier.
- **`storage`** (*`Union[str, optuna.storages._base.BaseStorage]`*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *`create_study()`* for further details.
- **`sampler`** (*`Optional[optuna.samplers._base.BaseSampler]`*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *`TPESampler`* is used as the default. See also *`samplers`*.
- **`pruner`** (*`Optional[optuna.pruners._base.BasePruner]`*) – 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 *optuna.study.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")
```

(continues on next page)

(continued from previous page)

```
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`** (*`Union[str, optuna.storages._base.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.get_all_study_summaries

`optuna.study.`**`get_all_study_summaries`**`(`*storage*`)`

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`** (*`Union[str, optuna.storages._base.BaseStorage]`*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of *`create_study()`* for further details.

Returns List of study history summarized as *`StudySummary`* objects.

Return type List[optuna._study_summary.StudySummary]

See also:

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

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

MAXIMIZE
MINIMIZE
NOT_SET

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()`.

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

`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.

n_trials
The number of trials ran in the `Study`.

datetime_start
Datetime where the `Study` started.

Attributes

`direction`
`directions`

6.3.14 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** – A `Study` object.
- **trial_id** – 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(name, choices)	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.
system_attrs	Return system attributes.
user_attrs	Return user attributes.

property datetime_start

Return start datetime.

Returns Datetime where the *Trial* started.

property distributions

Return distributions of parameters to be optimized.

Returns A dictionary containing all distributions.

property number

Return trial's number which is consecutive and unique in a study.

Returns A trial number.

property params

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.

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.

Raises **`NotImplementedError`** – If trial is being used for multi-objective optimization.

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

`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()`*.

Example

Save fixed hyperparameters of 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):
    trial.set_user_attr("BATCHSIZE", 128)
    momentum = trial.suggest_uniform("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()`*.

Returns A boolean value. If `True`, the trial should be pruned according to the configured pruning algorithm. Otherwise, the trial should continue.

Raises **`NotImplementedError`** – If trial is being used for multi-objective optimization.

Return type bool

`suggest_categorical`(*name*, *choices*)

Suggest a value for the categorical parameter.

The value is sampled from `choices`.

Example

Suggest a kernel function of SVC.

```
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`** (*`str`*) – A parameter name.
- **`choices`** (*`Sequence[Union[None, bool, int, float, str]]`*) – Parameter value candidates.

Return type Union[None, bool, int, float, str]

See also:

`CategoricalDistribution`.

Returns A suggested value.

Parameters

- **`name`** (*`str`*) –
- **`choices`** (*`Sequence[Union[None, bool, int, float, str]]`*) –

Return type Union[None, bool, int, float, str]

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.

Example

Suggest a fraction of samples used for fitting the individual learners of GradientBoostingClassifier.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import GradientBoostingClassifier
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):
    subsample = trial.suggest_discrete_uniform("subsample", 0.1, 1.0, 0.1)
    clf = GradientBoostingClassifier(subsample=subsample, 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** (*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

suggest_float(*name*, *low*, *high*, ***, *step=None*, *log=False*)

Suggest a value for the floating point parameter.

Note that this is a wrapper method for *suggest_uniform()*, *suggest_loguniform()* and *suggest_discrete_uniform()*.

New in version 1.3.0.

See also:

Please see also *suggest_uniform()*, *suggest_loguniform()* and *suggest_discrete_uniform()*.

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)
    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.
- **high** (*float*) – Upper endpoint of the range of suggested values. `high` is excluded from the range.

  Note: If `step` is specified, `high` is included as well as `low` because this method falls back to *suggest_discrete_uniform()*.

- **step** (*Optional[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. See also *`suggest_uniform()`* and *`suggest_loguniform()`*.

> **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`.

Raises **`ValueError`** – If `step is not None` and `log = True` are specified.

Returns A suggested float value.

Return type float

`suggest_int`(*name*, *low*, *high*, *step=1*, *log=False*)

Suggest a value for the integer parameter.

The value is sampled from the integers in $[\text{low}, \text{high}]$.

Example

Suggest the number of trees in RandomForestClassifier.

```
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.

- **high** (*int*) – Upper endpoint of the range of suggested values. `high` is included in the range.
- **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.

Raises **ValueError** – If `step != 1` and `log = True` are specified.

Return type int

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.

Example

Suggest penalty parameter `C` of SVC.

```
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)
```

(continues on next page)

(continued from previous page)

```
X_train, X_valid, y_train, y_valid = train_test_split(X, y)


def objective(trial):
    c = trial.suggest_loguniform("c", 1e-5, 1e2)
    clf = SVC(C=c, 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`** (*`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 excluded from the range.

Returns A suggested float value.

Return type float

`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.

Example

Suggest a momentum 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)


def objective(trial):
    momentum = trial.suggest_uniform("momentum", 0.0, 1.0)
    clf = MLPClassifier(
        hidden_layer_sizes=(100, 50),
        momentum=momentum,
        solver="sgd",
        random_state=0,
    )
```

(continues on next page)

(continued from previous page)

```
    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.
- **high** (*float*) – Upper endpoint of the range of suggested values. `high` is excluded from the range.

Returns A suggested float value.

Return type float

property system_attrs
Return system attributes.

Returns A dictionary containing all system attributes.

property user_attrs
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.

```
import optuna


def objective(trial):
    x = trial.suggest_uniform("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`** – A dictionary containing all parameters.
- **`number`** – A trial number. Defaults to `0`.

Methods

`report`(value, step)
`set_system_attr`(key, value)
`set_user_attr`(key, value)
`should_prune`()
`suggest_categorical`(name, choices)
`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`

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`.

This object has the same methods as `Trial`, and it suggests best parameter values among performed trials. In contrast to `Trial`, `FrozenTrial` does not depend on `Study`, and it 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_uniform("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: Attributes are set in `optuna.Study.optimize()`, but several attributes can be updated after the optimization. That means such attributes are overwritten by the re-evaluation if your objective updates attributes of `Trial`.

Example:

Overwritten attributes.

```
import copy
import datetime

import optuna


def objective(trial):
    x = trial.suggest_uniform("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)
```

(continues on next page)

(continued from previous page)

```
# 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.

`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`.

`values`
Sequence of objective values of the `Trial`. The length is greater than 1 if the problem is multi-objective optimization.

`datetime_start`
Datetime where the `Trial` started.

`datetime_complete`
Datetime where the `Trial` finished.

`params`
Dictionary that contains suggested parameters.

`user_attrs`
Dictionary that contains the attributes of the `Trial` set with `optuna.trial.Trial.set_user_attr()`.

`intermediate_values`
Intermediate objective values set with `optuna.trial.Trial.report()`.

Raises **`ValueError`** – If both `value` and `values` are specified.

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`(name, choices)	

continues on next page

Table 97 – continued from previous page

`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`	Dictionary that contains the distributions of *`params`*.
`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 distributions`

Dictionary that contains the distributions of *`params`*.

`property duration`

Return the elapsed time taken to complete the trial.

Returns The duration.

`property last_step`

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

`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

optuna.trial.TrialState

`class` `optuna.trial.`**`TrialState`**`(`*value*`)`

State of a *`Trial`*.

`RUNNING`

The *`Trial`* is running.

`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.

Attributes

`COMPLETE`
`FAIL`
`PRUNED`
`RUNNING`
`WAITING`

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 UniformDistribution

trial = optuna.trial.create_trial(
    params={"x": 1.0, "y": 0},
    distributions={
        "x": UniformDistribution(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`** (*`optuna.trial._state.TrialState`*) – Trial state.
- **`value`** (*`Optional[float]`*) – Trial objective value. Must be specified if `state` is `None` or *`TrialState.COMPLETE`*.
- **`values`** (*`Optional[Sequence[float]]`*) – 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 `None` or *`TrialState.COMPLETE`*.
- **`params`** (*`Optional[Dict[str, Any]]`*) – Dictionary with suggested parameters of the trial.
- **`distributions`** (*`Optional[Dict[str, optuna.distributions.BaseDistribution]]`*) – Dictionary with parameter distributions of the trial.
- **`user_attrs`** (*`Optional[Dict[str, Any]]`*) – Dictionary with user attributes.

- **system_attrs** (*Optional[Dict[str, Any]]*) – Dictionary with system attributes. Should not have to be used for most users.
- **intermediate_values** (*Optional[Dict[int, float]]*) – Dictionary with intermediate objective values of the trial.

Returns Created trial.

Raises **ValueError** – If both `value` and `values` are specified.

Return type optuna.trial._frozen.FrozenTrial

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.

6.3.15 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.

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_slice	Plot the parameter relationship as slice plot in 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.

```
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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`params`** (*`Optional[List[str]]`*) – Parameter list to visualize. The default is all parameters.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type plotly.graph_objs._figure.Figure

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** (*Union*[`optuna.study.Study`, *Sequence*[`optuna.study.Study`]]) – A target *Study* object. You can pass multiple studies if you want to compare those EDFs.
- **target** (*Optional*[*Callable*[[`optuna.trial._frozen.FrozenTrial`], *float*]]) – 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.

Raises **ValueError** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type plotly.graph_objs._figure.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.

```
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
```

(continues on next page)

(continued from previous page)

```
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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their intermediate values.

Returns A `plotly.graph_objs.Figure` object.

Return type plotly.graph_objs._figure.Figure

optuna.visualization.plot_optimization_history

`optuna.visualization.`**`plot_optimization_history`**(*study*, ***, *target=None*, *target_name='Objective Value'*)

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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type plotly.graph_objs._figure.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.

```
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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`params`** (*`Optional[List[str]]`*) – Parameter list to visualize. The default is all parameters.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type plotly.graph_objs._figure.Figure

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`** (`optuna.study.Study`) – An optimized study.
- **`evaluator`** (*`Optional[optuna.importance._base.BaseImportanceEvaluator]`*) – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *`FanovaImportanceEvaluator`*.
- **`params`** (*`Optional[List[str]]`*) – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type plotly.graph_objs._figure.Figure

optuna.visualization.plot_pareto_front

`optuna.visualization.`**`plot_pareto_front`**(*study*, ***, *target_names=None*, *include_dominated_trials=True*, *axis_order=None*)

Plot the Pareto front of a study.

Example

The following code snippet shows how to plot the Pareto front of a study.

```
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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their objective values.
- **`target_names`** (*`Optional[List[str]]`*) – Objective name list used as the axis titles. If `None` is specified, "Objective {objective_index}" is used instead.
- **`include_dominated_trials`** (*`bool`*) – A flag to include all dominated trial's objective values.
- **`axis_order`** (*`Optional[List[int]]`*) – A list of indices indicating the axis order. If `None` is specified, default order is used.

Returns A `plotly.graph_objs.Figure` object.

Raises **`ValueError`** – If the number of objectives of `study` isn't 2 or 3.

Return type plotly.graph_objs._figure.Figure

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.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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`params`** (*`Optional[List[str]]`*) – Parameter list to visualize. The default is all parameters.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type plotly.graph_objs._figure.Figure

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_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

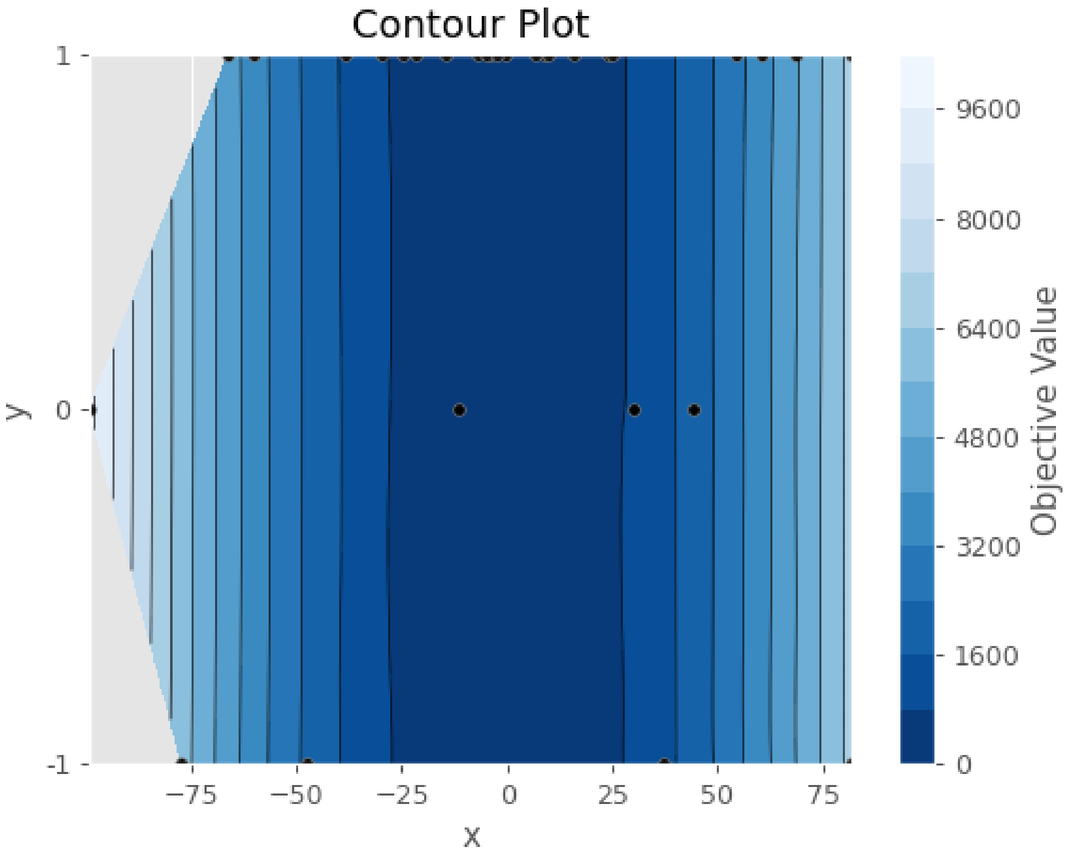
The following code snippet shows how to plot the parameter relationship as contour plot.

```
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** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **params** (*`Optional[List[str]]`*) – Parameter list to visualize. The default is all parameters.
- **target** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type matplotlib.axes._axes.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_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.

See also:

Please refer to *`optuna.visualization.plot_edf()`* for an example, where this function can be replaced with it.

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)
```

(continues on next page)

(continued from previous page)

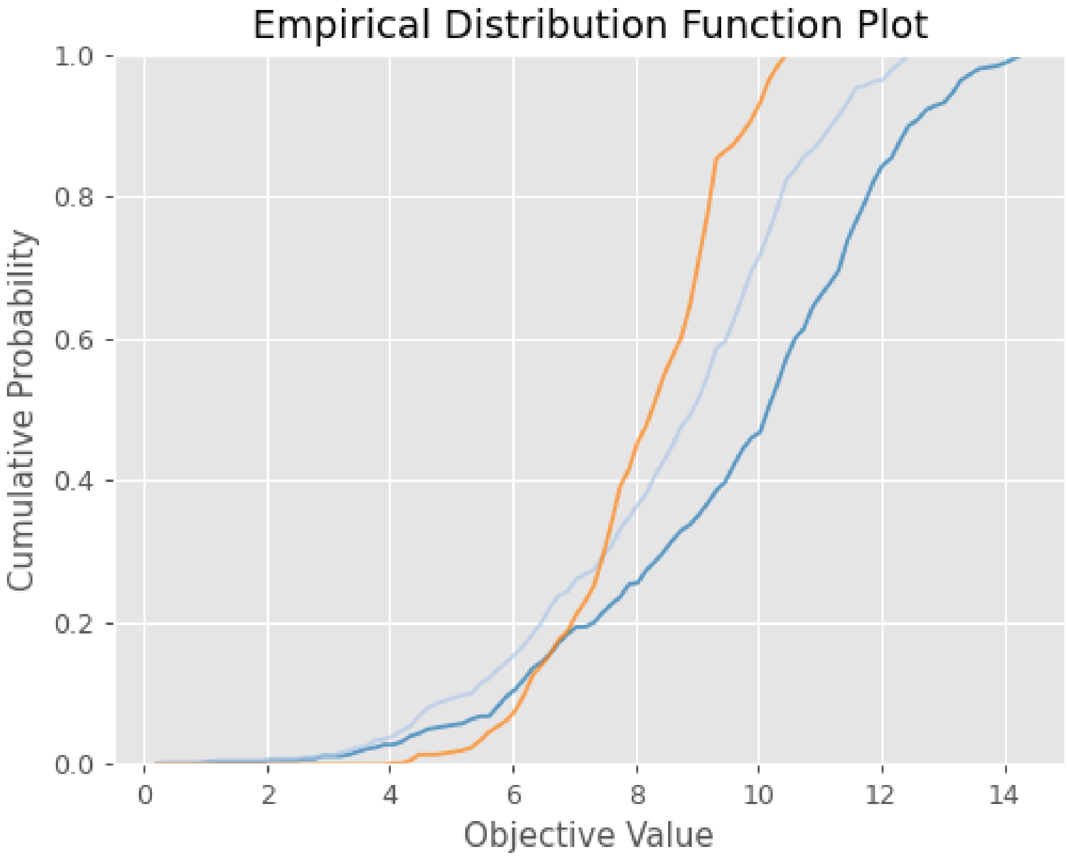
```
sampler = optuna.samplers.RandomSampler(seed=10)

# Widest search space.
study0 = optuna.create_study(study_name="x=[0,5), y=[0,5)", sampler=sampler)
study0.optimize(lambda t: objective(t, 0, 5), n_trials=500)

# Narrower search space.
study1 = optuna.create_study(study_name="x=[0,4), y=[0,4)", sampler=sampler)
study1.optimize(lambda t: objective(t, 0, 4), n_trials=500)

# Narrowest search space but it doesn't include the global optimum point.
study2 = optuna.create_study(study_name="x=[1,3), y=[1,3)", sampler=sampler)
study2.optimize(lambda t: objective(t, 1, 3), n_trials=500)

optuna.visualization.matplotlib.plot_edf([study0, study1, study2])
```



Parameters

- **study** (*Union*[`optuna.study.Study`, *Sequence*[`optuna.study.Study`]]) – A target *Study* object. You can pass multiple studies if you want to compare those EDFs.

- **target** (*Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]*) – 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.

Raises `ValueError` – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type matplotlib.axes._axes.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.

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
```

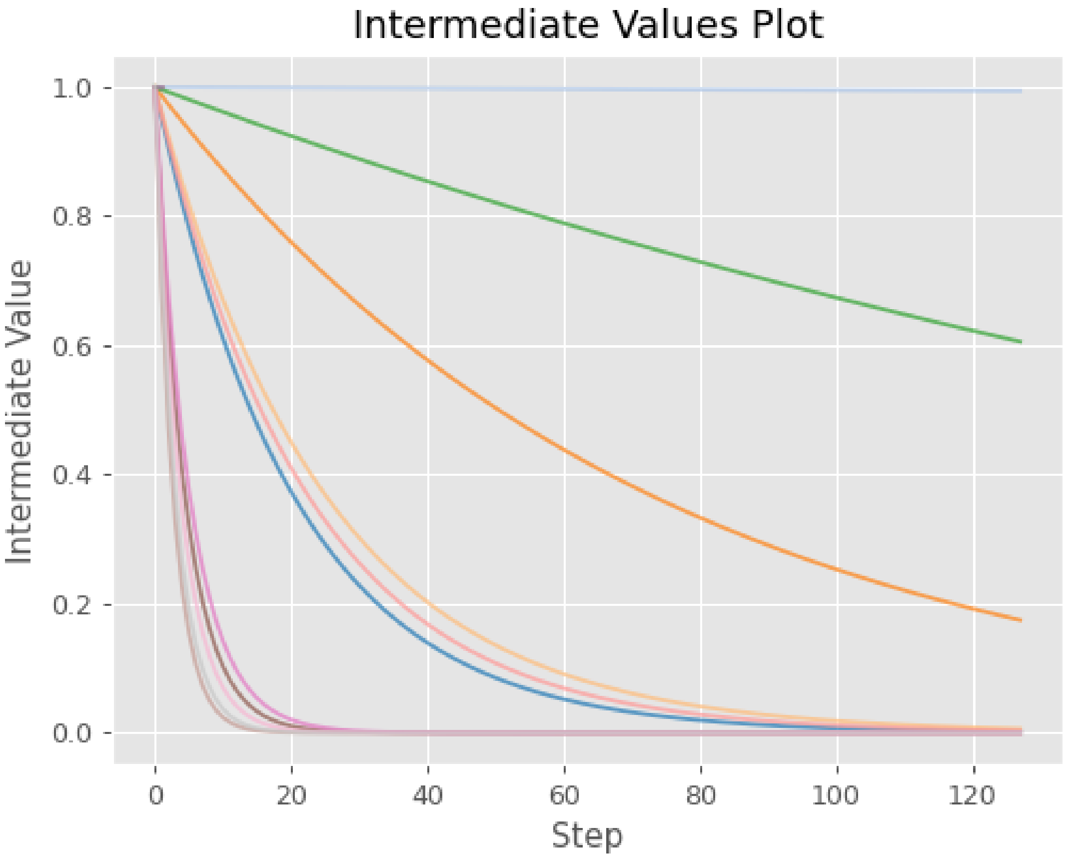
(continues on next page)

(continued from previous page)

```
    return y


sampler = optuna.samplers.TPESampler(seed=10)
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=16)

optuna.visualization.matplotlib.plot_intermediate_values(study)
```



See also:

Please refer to *`optuna.visualization.plot_intermediate_values()`* for an example.

Parameters **`study`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their intermediate values.

Returns A `matplotlib.axes.Axes` object.

Return type matplotlib.axes._axes.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'*)

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


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)
```

Parameters

- **`study`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

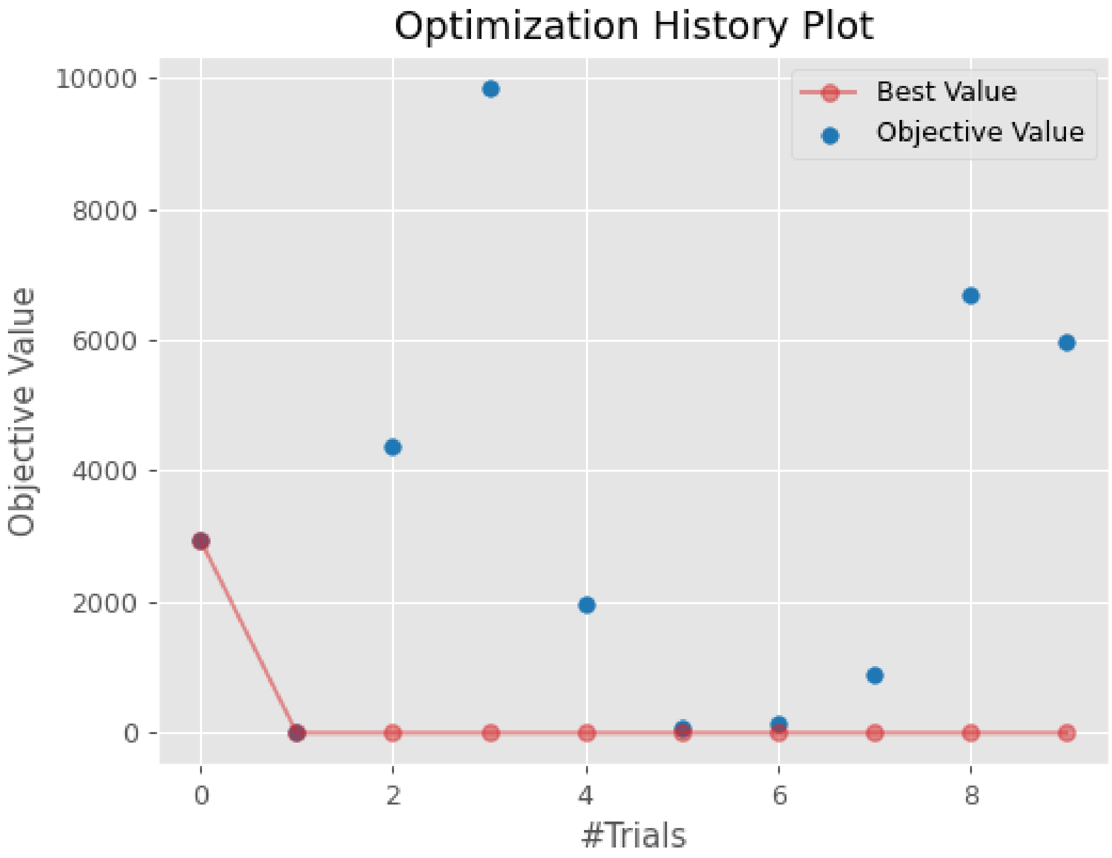
- **`target_name`** (*`str`*) – Target's name to display on the axis label and the legend.

Returns A `matplotlib.axes.Axes` object.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type matplotlib.axes._axes.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.


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

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.

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`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`params`** (*`Optional[List[str]]`*) – Parameter list to visualize. The default is all parameters.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

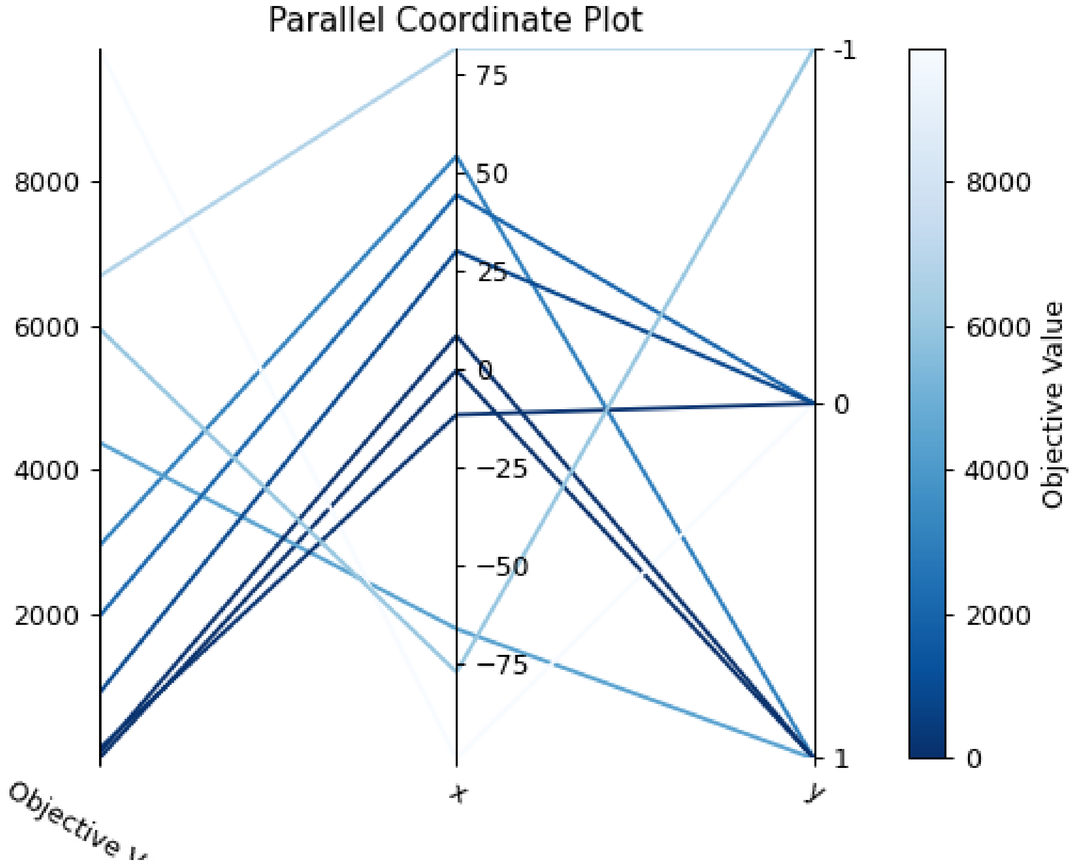
- **`target_name`** (*`str`*) – Target's name to display on the axis label and the legend.

Returns A `matplotlib.axes.Axes` object.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.

Return type matplotlib.axes._axes.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.


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

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

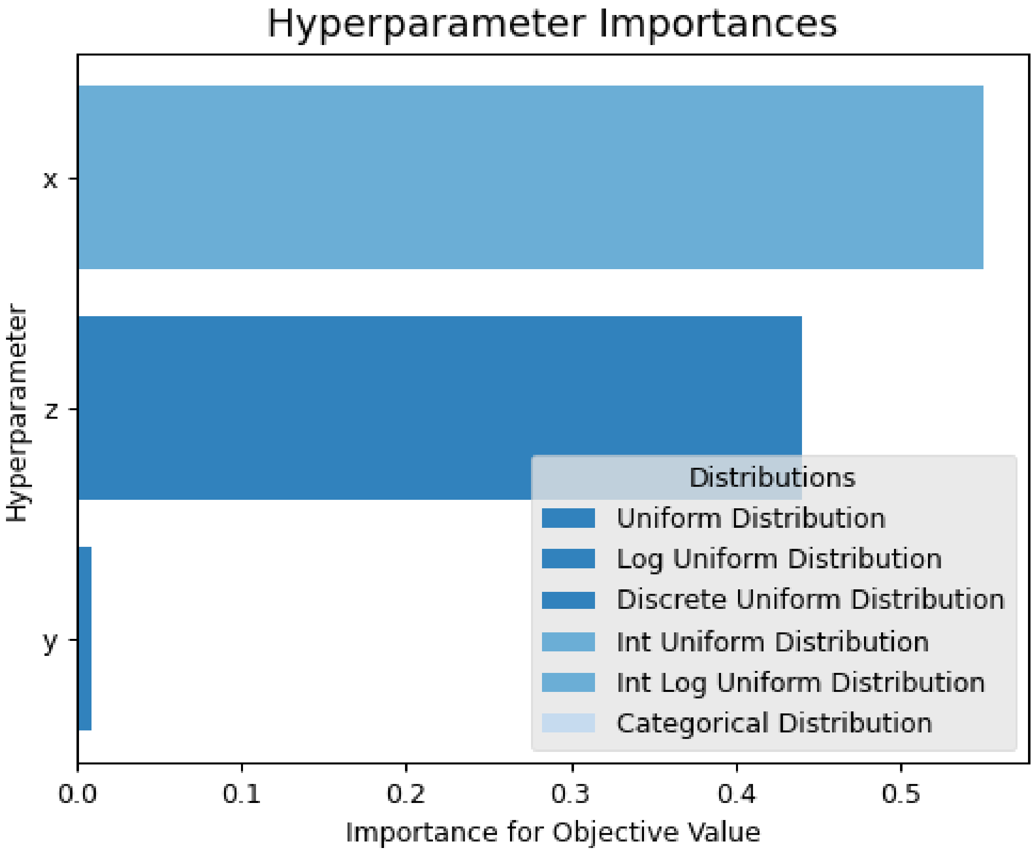
- **`study`** (`optuna.study.Study`) – An optimized study.
- **`evaluator`** (*`Optional[optuna.importance._base.BaseImportanceEvaluator]`*) – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *`FanovaImportanceEvaluator`*.
- **`params`** (*`Optional[List[str]]`*) – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.


Hyperparameter Importances
Hyperparameter
x
z
y
Importance for Objective Value
0.0
0.1
0.2
0.3
0.4
0.5
Distributions
Uniform Distribution
Log Uniform Distribution
Discrete Uniform Distribution
Int Uniform Distribution
Int Log Uniform Distribution
Categorical Distribution

Return type matplotlib.axes._axes.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_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

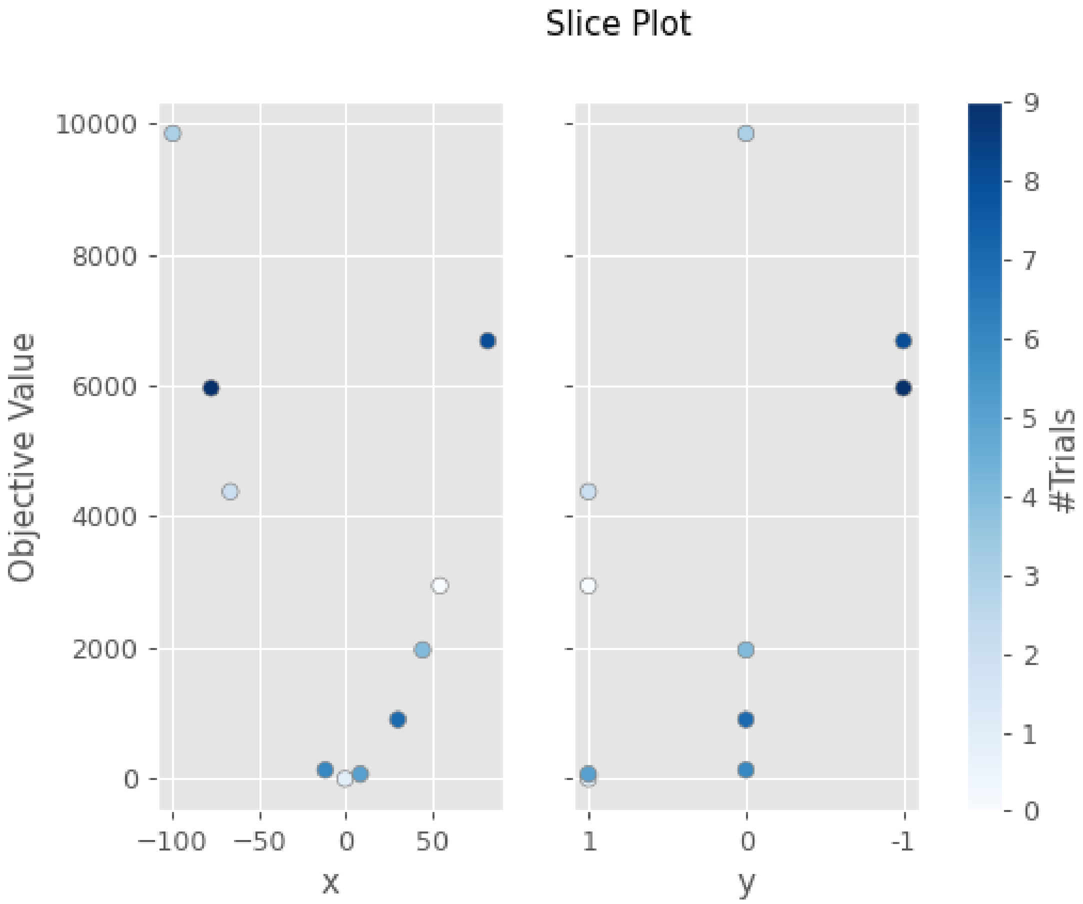
- **`study`** (`optuna.study.Study`) – A *`Study`* object whose trials are plotted for their target values.
- **`params`** (*`Optional[List[str]]`*) – Parameter list to visualize. The default is all parameters.
- **`target`** (*`Optional[Callable[[optuna.trial._frozen.FrozenTrial], float]]`*) – A function 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.

Raises **`ValueError`** – If `target` is `None` and `study` is being used for multi-objective optimization.


Slice Plot
Objective Value
10000
8000
6000
4000
2000
0
−100
−50
0
50
x
1
0
-1
y
#Trials
0
1
2
3
4
5
6
7
8
9

Return type matplotlib.axes._axes.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.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.

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:

```
import optuna


class Objective(object):
    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`:

```
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


# 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 depends 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}")
```

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 `RandomSampler` or `TPESampler` as follows:

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

However, there are two caveats.

First, when optimizing a study in distributed or parallel mode, there is inherent non-determinism. Thus it is very difficult to reproduce the same results in such condition. We recommend executing optimization of a study sequentially if you would like to reproduce the result.

Second, if your objective function behaves in a non-deterministic way (i.e., it does not return the same value even if the same parameters were suggested), you cannot reproduce an optimization. To deal with this problem, please set an option (e.g., random seed) to make the behavior deterministic if your optimization target (e.g., an ML library) provides it.

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	…	params	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.COMPLETE	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, 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
$ optuna study optimize foo.py objective --study-name foo --storage sqlite:///example.
↪db

# On another terminal.
#
# Specify to use the second GPU, and run another optimization.
$ export CUDA_VISIBLE_DEVICES=1
$ optuna study optimize bar.py objective --study-name bar --storage sqlite:///example.
↪db
```

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.

```
def objective(trial):
    x = trial.suggest_float("x", -1.0, 1.0)
    y = trial.suggest_int("y", -5, 5)
    return x + y
```

(continues on next page)

(continued from previous page)

```
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.

CHAPTER
SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX