Optuna Documentation

Release 2.0.0

Optuna Contributors.

May 26, 2022

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*
- *Parallel distributed optimization*
- *Pruning of unpromising trials*

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*.

```python
import ...

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

    # Invoke suggest methods of a Trial object to generate hyperparameters.
    regressor_name = trial.suggest_categorical('classifier', ['SVR', 'RandomForest'])
    if regressor_name == 'SVR':
        svr_c = trial.suggest_loguniform('svr_c', 1e-10, 1e10)
        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 3.5 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

6.2.1 First Optimization

Quadratic Function Example

Usually, Optuna is used to optimize hyper-parameters, but as an example, let us directly optimize a quadratic function in an IPython shell.

```
import optuna
```

The objective function is what will be optimized.

```
def objective(trial):
    x = trial.suggest_uniform('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_uniform()`) are called inside the objective function to obtain parameters for a trial. `suggest_uniform()` 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)
```

Out:

```
[I 2020-04-08 10:42:09,028] Trial 0 finished with value: 25.77382032395108 with␣
↪parameters: {'x': 7.076792326257898}. Best is trial 0 with value: 25.77382032395108.
[I 2020-04-08 10:42:09,064] Trial 1 finished with value: 1.5189812248635903 with␣
↪parameters: {'x': 0.7675304365366298}. Best is trial 1 with value: 1.5189812248635903.
[I 2020-04-08 10:42:09,106] Trial 2 finished with value: 34.4074691838153 with␣
↪parameters: {'x': -3.865788027521562}. Best is trial 1 with value: 1.5189812248635903.
[I 2020-04-08 10:42:09,145] Trial 3 finished with value: 3.3601305753722657 with␣
↪parameters: {'x': 3.8330658949891205}. Best is trial 1 with value: 1.5189812248635903.
[I 2020-04-08 10:42:09,185] Trial 4 finished with value: 61.16797535698886 with␣
↪parameters: {'x': -5.820995803412048}. Best is trial 1 with value: 1.5189812248635903.
[I 2020-04-08 10:42:09,228] Trial 5 finished with value: 90.08665552769618 with␣
↪parameters: {'x': -7.491399028999686}. Best is trial 1 with value: 1.5189812248635903.
[I 2020-04-08 10:42:09,274] Trial 6 finished with value: 25.254236332163032 with␣
↪parameters: {'x': 7.025359323686519}. Best is trial 1 with value: 1.5189812248635903.
...
[I 2020-04-08 10:42:14,237] Trial 99 finished with value: 0.5227007740782738 with␣
↪parameters: {'x': 2.7229804797352926}. Best is trial 67 with value: 2.916284393762304e-
↪06.
```

You can get the best parameter as follows.

```
study.best_params
```

Out:

```
{'x': 2.001707713205946}
```

We can see that Optuna found the best `x` value `2.001707713205946`, which is close to the optimal value of `2`.

Note: When used to search for hyper-parameters 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 best parameter:

```
study.best_params
```

Out:

```
{'x': 2.001707713205946}
```

To get the best value:

```
study.best_value
```

Out:

```
2.916284393762304e-06
```

To get the best trial:

```
study.best_trial
```

Out:

```
FrozenTrial(number=67, value=2.916284393762304e-06, datetime_start=datetime.
↪datetime(2020, 4, 8, 10, 42, 12, 595884), datetime_complete=datetime.datetime(2020, 4,␣
↪8, 10, 42, 12, 639969), params={'x': 2.001707713205946}, distributions={'x':␣
↪UniformDistribution(high=10, low=-10)}, user_attrs={}, system_attrs={}, intermediate_
↪values={}, trial_id=67, state=TrialState.COMPLETE)
```

To get all trials:

```
study.trials
```

Out:

```
[FrozenTrial(number=0, value=25.77382032395108, datetime_start=datetime.datetime(2020, 4,
↪ 8, 10, 42, 8, 987277), datetime_complete=datetime.datetime(2020, 4, 8, 10, 42, 9,␣
↪27959), params={'x': 7.076792326257898}, distributions={'x':␣
↪UniformDistribution(high=10, low=-10)}, user_attrs={}, system_attrs={}, intermediate_
↪values={}, trial_id=0, state=TrialState.COMPLETE),
 ...
 user_attrs={}, system_attrs={}, intermediate_values={}, trial_id=99, state=TrialState.
↪COMPLETE)]
```

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
```

6.2.2 Advanced Configurations

Defining Parameter Spaces

Optuna supports five kinds of parameters.

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

    # Int parameter
    num_layers = trial.suggest_int('num_layers', 1, 3)

    # Uniform parameter
    dropout_rate = trial.suggest_uniform('dropout_rate', 0.0, 1.0)

    # Loguniform parameter
    learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 1e-2)

    # Discrete-uniform parameter
    drop_path_rate = trial.suggest_discrete_uniform('drop_path_rate', 0.0, 1.0, 0.1)

    ...
```

Branches and Loops

You can use branches or loops depending on the parameter values.

```
def objective(trial):
    classifier_name = trial.suggest_categorical('classifier', ['SVC', 'RandomForest'])
    if classifier_name == 'SVC':
        svc_c = trial.suggest_loguniform('svc_c', 1e-10, 1e10)
        classifier_obj = sklearn.svm.SVC(C=svc_c)
```

(continues on next page)

(continued from previous page)

```
    else:
        rf_max_depth = int(trial.suggest_loguniform('rf_max_depth', 2, 32))
        classifier_obj = sklearn.ensemble.RandomForestClassifier(max_depth=rf_max_depth)

    ...
```

```
def create_model(trial):
    n_layers = trial.suggest_int('n_layers', 1, 3)

    layers = []
    for i in range(n_layers):
        n_units = int(trial.suggest_loguniform('n_units_l{}'.format(i), 4, 128))
        layers.append(L.Linear(None, n_units))
        layers.append(F.relu)
    layers.append(L.Linear(None, 10))

    return chainer.Sequential(*layers)
```

Please also refer to examples.

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.

Arguments for *Study.optimize*

The method *optimize()* (and `optuna study optimize` CLI command as well) has several useful options such as `timeout`. For details, please refer to the API reference for *optimize()*.

FYI: If you give neither `n_trials` nor `timeout` options, the optimization continues until it receives a termination signal such as Ctrl+C or SIGTERM. This is useful for use cases such as when it is hard to estimate the computational costs required to optimize your objective function.

6.2.3 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 *Distributed Optimization*.

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 optuna
study_name = 'example-study'  # Unique identifier of the study.
study = optuna.create_study(study_name=study_name, storage='sqlite:///example.db')
```

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

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

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

Resume Study

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

```
study = optuna.create_study(study_name='example-study', storage='sqlite:///example.db',␣
↪load_if_exists=True)
study.optimize(objective, n_trials=3)
```

Experimental History

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

```
import optuna
study = optuna.create_study(study_name='example-study', storage='sqlite:///example.db',␣
↪load_if_exists=True)
df = study.trials_dataframe(attrs=('number', 'value', 'params', 'state'))
```

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

```
print(df)
```

Out:

```
   number       value  params_x     state
0       0   25.301959 -3.030105  COMPLETE
1       1    1.406223  0.814157  COMPLETE
2       2   44.010366 -4.634031  COMPLETE
3       3   55.872181  9.474770  COMPLETE
4       4  113.039223 -8.631991  COMPLETE
5       5   57.319570  9.570969  COMPLETE
```

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

```
study.best_params  # Get best parameters for the objective function.
study.best_value  # Get best objective value.
study.best_trial  # Get best trial's information.
study.trials  # Get all trials' information.
```

6.2.4 Distributed Optimization

There is no complicated setup but just sharing the same study name among nodes/processes.

First, create a shared study using `optuna create-study` command (or using *`optuna.create_study()`* in a Python script).

```
$ optuna create-study --study-name "distributed-example" --storage "sqlite:///example.db"
[I 2018-10-31 18:21:57,885] 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_uniform('x', -10, 10)
    return (x - 2) ** 2

if __name__ == '__main__':
    study = optuna.load_study(study_name='distributed-example', storage='sqlite:///
↪example.db')
    study.optimize(objective, n_trials=100)
```

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 2018-10-31 18:46:44,308] Finished a trial resulted in value: 1.1097007755908204.␣
↪Current best value is 0.00020881104123229936 with parameters: {'x': 2.014450295541348}.
[I 2018-10-31 18:46:44,361] Finished a trial resulted in value: 0.5186699439824186.␣
↪Current best value is 0.00020881104123229936 with parameters: {'x': 2.014450295541348}.
...
```

Process 2 (the same command as process 1):

```
$ python foo.py
[I 2018-10-31 18:47:02,912] Finished a trial resulted in value: 29.821448668796563.␣
↪Current best value is 0.00020881104123229936 with parameters: {'x': 2.014450295541348}.
[I 2018-10-31 18:47:02,968] Finished a trial resulted in value: 0.7962498978463782.␣
↪Current best value is 0.00020881104123229936 with parameters: {'x': 2.014450295541348}.
...
```

Note: We do not recommend SQLite for large scale distributed optimizations because it may cause 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

6.2.5 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:

```python
import optuna


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

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

```python
def objective(trial):
    x = trial.suggest_uniform('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_uniform('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
```

(continues on next page)

(continued from previous page)

```
[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.

6.2.6 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 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'}
```

`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'}
```

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_loguniform('svc_c', 1e-10, 1e10)
    clf = sklearn.svm.SVC(C=svc_c)
    accuracy = sklearn.model_selection.cross_val_score(clf, x, y).mean()
```

(continues on next page)

(continued from previous page)

```
    trial.set_user_attr('accuracy', accuracy)

    return 1.0 - accuracy  # return error for minimization
```

We can access annotated attributes as:

```
study.trials[0].user_attrs  # {'accuracy': 0.83}
```

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

6.2.7 Pruning Unpromising Trials

This feature automatically stops unpromising trials at the early stages of the training (a.k.a., automated early-stopping). Optuna provides interfaces to concisely implement the pruning mechanism in iterative training algorithms.

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.

```
"""filename: prune.py"""

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

import optuna

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_loguniform('alpha', 1e-5, 1e-1)
    clf = sklearn.linear_model.SGDClassifier(alpha=alpha)

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

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

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

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

(continues on next page)

(continued from previous page)

```

# Set up the median stopping rule as the pruning condition.
study = optuna.create_study(pruner=optuna.pruners.MedianPruner())
study.optimize(objective, n_trials=20)
```

Executing the script above:

```
$ python prune.py
[I 2020-06-12 16:54:23,876] Trial 0 finished with value: 0.3157894736842105 and␣
↪parameters: {'alpha': 0.00181467547181131}. Best is trial 0 with value: 0.
↪3157894736842105.
[I 2020-06-12 16:54:23,981] Trial 1 finished with value: 0.07894736842105265 and␣
↪parameters: {'alpha': 0.015378744419287613}. Best is trial 1 with value: 0.
↪07894736842105265.
[I 2020-06-12 16:54:24,083] Trial 2 finished with value: 0.21052631578947367 and␣
↪parameters: {'alpha': 0.04089428832878595}. Best is trial 1 with value: 0.
↪07894736842105265.
[I 2020-06-12 16:54:24,185] Trial 3 finished with value: 0.052631578947368474 and␣
↪parameters: {'alpha': 0.004018735937374473}. Best is trial 3 with value: 0.
↪052631578947368474.
[I 2020-06-12 16:54:24,303] Trial 4 finished with value: 0.07894736842105265 and␣
↪parameters: {'alpha': 2.805688697062864e-05}. Best is trial 3 with value: 0.
↪052631578947368474.
[I 2020-06-12 16:54:24,315] Trial 5 pruned.
[I 2020-06-12 16:54:24,355] Trial 6 pruned.
[I 2020-06-12 16:54:24,511] Trial 7 finished with value: 0.052631578947368474 and␣
↪parameters: {'alpha': 2.243775785299103e-05}. Best is trial 3 with value: 0.
↪052631578947368474.
[I 2020-06-12 16:54:24,625] Trial 8 finished with value: 0.1842105263157895 and␣
↪parameters: {'alpha': 0.007021209286214553}. Best is trial 3 with value: 0.
↪052631578947368474.
[I 2020-06-12 16:54:24,629] Trial 9 pruned.
...
```

`Trial 5 pruned.`, etc. in the log messages means several trials were stopped before they finished all of the iterations.

Integration Modules for Pruning

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

- XGBoost: *optuna.integration.XGBoostPruningCallback*
- LightGBM: *optuna.integration.LightGBMPruningCallback*
- Chainer: *optuna.integration.ChainerPruningExtension*
- Keras: *optuna.integration.KerasPruningCallback*
- TensorFlow *optuna.integration.TensorFlowPruningHook*
- tf.keras *optuna.integration.TFKerasPruningCallback*
- MXNet *optuna.integration.MXNetPruningCallback*
- PyTorch Ignite *optuna.integration.PyTorchIgnitePruningHandler*

- PyTorch Lightning *optuna.integration.PyTorchLightningPruningCallback*
- FastAI *optuna.integration.FastAIPruningCallback*

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])
```

6.2.8 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_uniform()*) 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 {}

        #
        # An implementation of SA algorithm.
        #

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

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

        # 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_uniform() 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
```

(continues on next page)

(continued from previous page)

```
    #
    # The rest is boilerplate code and unrelated to SA algorithm.
    #
    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_uniform('x', -10, 10)
    y = trial.suggest_uniform('y', -5, 5)
    return x**2 + y

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

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.

6.3 API Reference

6.3.1 optuna

`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: Union[None, str, storages.BaseStorage] = None*, *sampler: samplers.BaseSampler = None*, *pruner: pruners.BasePruner = None*, *study_name: Optional[str] = None*, *direction: str = 'minimize'*, *load_if_exists: bool = False*) → *Study*

Create a new *Study*.

Parameters

- **storage** – 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** – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *TPESampler* is used as the default. See also `samplers`.
- **pruner** – A pruner object that decides early stopping of unpromising trials. See also `pruners`.
- **study_name** – Study's name. If this argument is set to None, a unique name is generated automatically.
- **direction** – Direction of optimization. Set `minimize` for minimization and `maximize` for maximization.
- **load_if_exists** – 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 *Study* object.

See also:

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

optuna.load_study

`optuna.load_study`(*study_name: str*, *storage: Union[str, storages.BaseStorage]*, *sampler: samplers.BaseSampler = None*, *pruner: pruners.BasePruner = None*) → *Study*

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

Parameters

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

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

See also:

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

optuna.delete_study

`optuna.`**`delete_study`**(*study_name: str*, *storage: Union[str, storages.BaseStorage]*) → None

Delete a *`Study`* object.

Parameters

- **`study_name`** – Study's name.
- **`storage`** – Database URL such as `sqlite:///example.db`. Please see also the documentation of *`create_study()`* for further details.

See also:

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

optuna.get_all_study_summaries

`optuna.`**`get_all_study_summaries`**(*storage: Union[str, storages.BaseStorage]*) → List[*StudySummary*]

Get all history of studies stored in a specified storage.

Parameters **`storage`** – 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.

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

```
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_uniform('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)
```

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

6.3.2 optuna.cli

```
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).

```
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>

A human-readable name of a study to distinguish it from others.

--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>

A human-readable name of a study to distinguish it from others.

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

`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>

A human-readable name of a study to distinguish it from others.

--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

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: float*, *high: float*)

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 excluded from the range.

`__init__`(*low: float*, *high: float*) → None

Methods

`__init__`(low, high)	
`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`() → bool

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

When this method returns `True`, `samplers` always sample the same value from the distribution.

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

`to_external_repr`(*param_value_in_internal_repr: float*) → Any

Convert internal representation of a parameter value into external representation.

Parameters **`param_value_in_internal_repr`** – Optuna's internal representation of a parameter value.

Returns Optuna's external representation of a parameter value.

`to_internal_repr`(*param_value_in_external_repr: Any*) → float

Convert external representation of a parameter value into internal representation.

Parameters **`param_value_in_external_repr`** – Optuna's external representation of a parameter value.

Returns Optuna's internal representation of a parameter value.

optuna.distributions.LogUniformDistribution

`class optuna.distributions.`**`LogUniformDistribution`**(*low: float*, *high: float*)

A uniform distribution in the log domain.

This object is instantiated by *`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 excluded from the range.

`__init__`(*low: float*, *high: float*) → None

Methods

`__init__`(low, high)	
`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() → bool

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

When this method returns `True`, `samplers` always sample the same value from the distribution.

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

to_external_repr(*param_value_in_internal_repr: float*) → Any

Convert internal representation of a parameter value into external representation.

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

Returns Optuna's external representation of a parameter value.

to_internal_repr(*param_value_in_external_repr: Any*) → float

Convert external representation of a parameter value into internal representation.

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

Returns Optuna's internal representation of a parameter value.

optuna.distributions.DiscreteUniformDistribution

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

A discretized uniform distribution in the linear domain.

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

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

__init__(*low: float, high: float, q: float*) → None

Methods

__init__(low, high, q)	
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() → bool

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

When this method returns `True`, `samplers` always sample the same value from the distribution.

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

to_external_repr(*param_value_in_internal_repr: float*) → Any

Convert internal representation of a parameter value into external representation.

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

Returns Optuna's external representation of a parameter value.

to_internal_repr(*param_value_in_external_repr: Any*) → float

Convert external representation of a parameter value into internal representation.

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

Returns Optuna's internal representation of a parameter value.

optuna.distributions.IntUniformDistribution

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

A uniform distribution on integers.

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

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

__init__(*low: int, high: int, step: int = 1*) → None

Methods

`__init__`(low, high[, step])	
`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`() → bool

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

When this method returns `True`, `samplers` always sample the same value from the distribution.

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

`to_external_repr`(*param_value_in_internal_repr: float*) → int

Convert internal representation of a parameter value into external representation.

Parameters **`param_value_in_internal_repr`** – Optuna's internal representation of a parameter value.

Returns Optuna's external representation of a parameter value.

`to_internal_repr`(*param_value_in_external_repr: int*) → float

Convert external representation of a parameter value into internal representation.

Parameters **`param_value_in_external_repr`** – Optuna's external representation of a parameter value.

Returns Optuna's internal representation of a parameter value.

optuna.distributions.IntLogUniformDistribution

class `optuna.distributions.`**`IntLogUniformDistribution`**(*low: int*, *high: int*, *step: int = 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.

`__init__`(*low: int*, *high: int*, *step: int = 1*) → None

Methods

`__init__`(low, high[, step])	
`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`() → bool

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

When this method returns `True`, `samplers` always sample the same value from the distribution.

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

`to_external_repr`(*param_value_in_internal_repr: float*) → int

Convert internal representation of a parameter value into external representation.

Parameters **`param_value_in_internal_repr`** – Optuna's internal representation of a parameter value.

Returns Optuna's external representation of a parameter value.

`to_internal_repr`(*param_value_in_external_repr: int*) → float

Convert external representation of a parameter value into internal representation.

Parameters **`param_value_in_external_repr`** – Optuna's external representation of a parameter value.

Returns Optuna's internal representation of a parameter value.

optuna.distributions.CategoricalDistribution

class `optuna.distributions.`**`CategoricalDistribution`**(*choices: Sequence[Union[None, bool, int, float, str]]*)

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.

`__init__`(*choices: Sequence[Union[None, bool, int, float, str]]*) → None

Methods

`__init__`(choices)	
`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`() → bool

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

When this method returns `True`, `samplers` always sample the same value from the distribution.

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

`to_external_repr`(*param_value_in_internal_repr: float*) → Union[None, bool, int, float, str]

Convert internal representation of a parameter value into external representation.

Parameters **`param_value_in_internal_repr`** – Optuna's internal representation of a parameter value.

Returns Optuna's external representation of a parameter value.

`to_internal_repr`(*param_value_in_external_repr: Union[None, bool, int, float, str]*) → float

Convert external representation of a parameter value into internal representation.

Parameters **`param_value_in_external_repr`** – Optuna's external representation of a parameter value.

Returns Optuna's internal representation of a parameter value.

optuna.distributions.distribution_to_json

`optuna.distributions.`**`distribution_to_json`**(*dist: optuna.distributions.BaseDistribution*) → str

Serialize a distribution to JSON format.

Parameters **`dist`** – A distribution to be serialized.

Returns A JSON string of a given distribution.

optuna.distributions.json_to_distribution

optuna.distributions.**json_to_distribution**(*json_str: str*) → optuna.distributions.BaseDistribution

Deserialize a distribution in JSON format.

Parameters **json_str** – A JSON-serialized distribution.

Returns A deserialized distribution.

optuna.distributions.check_distribution_compatibility

optuna.distributions.**check_distribution_compatibility**(*dist_old: optuna.distributions.BaseDistribution*, *dist_new: optuna.distributions.BaseDistribution*) → None

A function to check compatibility of two distributions.

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

Parameters

- **dist_old** – A distribution previously recorded in storage.
- **dist_new** – A distribution newly added to storage.

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

6.3.4 optuna.exceptions

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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_uniform('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)
```

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

optuna.exceptions.CLIUsageError

exception `optuna.exceptions.`**`CLIUsageError`**

Exception for CLI.

CLI raises this exception when it receives invalid configuration.

`with_traceback()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

optuna.exceptions.StorageInternalError

exception `optuna.exceptions.`**`StorageInternalError`**

Exception for storage operation.

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

`with_traceback()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

`with_traceback()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

6.3.5 optuna.importance

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: optuna.study.Study*, *, *evaluator: Optional[optuna.importance._base.BaseImportanceEvaluator] = None*, *params: Optional[List[str]] = None*) → Dict[str, float]

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`** – An optimized study.
- **`evaluator`** – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *`FanovaImportanceEvaluator`*.
- **`params`** – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.

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

optuna.importance.FanovaImportanceEvaluator

`class optuna.importance.`**`FanovaImportanceEvaluator`**`(`**, n_trees: int = 64, max_depth: int = 64, seed: Optional[int] = 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`.

__init__(***, *n_trees: int = 64*, *max_depth: int = 64*, *seed: Optional[int] = None*) → None

Methods

`__init__`(*[, n_trees, max_depth, seed])	
`evaluate`(study[, params])	Evaluate parameter importances based on completed trials in the given study.

evaluate(*study: optuna.study.Study*, *params: Optional[List[str]] = None*) → Dict[str, float]

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** – An optimized study.
- **params** – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.

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

optuna.importance.MeanDecreaseImpurityImportanceEvaluator

class optuna.importance.**MeanDecreaseImpurityImportanceEvaluator**(***, *n_trees: int = 64*, *max_depth: int = 64*, *seed: Optional[int] = 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.

__init__(**, n_trees: int = 64, max_depth: int = 64, seed: Optional[int] = None*) → None

Methods

__init__(*[, n_trees, max_depth, seed])	
evaluate(study[, params])	Evaluate parameter importances based on completed trials in the given study.

evaluate(*study: optuna.study.Study, params: Optional[List[str]] = None*) → Dict[str, float]

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** – An optimized study.
- **params** – A list of names of parameters to assess. If `None`, all parameters that are present in all of the completed trials are assessed.

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

6.3.6 optuna.integration

optuna.integration.ChainerPruningExtension	Chainer extension to prune unpromising trials.
optuna.integration.ChainerMNStudy	A wrapper of *Study* to incorporate Optuna with ChainerMN.
optuna.integration.CatalystPruningCallback	Catalyst callback to prune unpromising trials.
optuna.integration.PyCmaSampler	A Sampler using cma library as the backend.
optuna.integration.CmaEsSampler	Wrapper class of PyCmaSampler for backward compatibility.
optuna.integration.FastAIPruningCallback	FastAI callback to prune unpromising trials for fastai.
optuna.integration.KerasPruningCallback	Keras callback to prune unpromising trials.
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.MLflowCallback	Callback to track Optuna trials with MLflow.
optuna.integration.MXNetPruningCallback	MXNet callback to prune unpromising trials.
optuna.integration.PyTorchIgnitePruningHandler	PyTorch Ignite handler to prune unpromising trials.
optuna.integration.PyTorchLightningPruningCallback	PyTorch Lightning callback to prune unpromising trials.
optuna.integration.SkoptSampler	Sampler using Scikit-Optimize as the backend.
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.XGBoostPruningCallback	Callback for XGBoost to prune unpromising trials.
optuna.integration.OptunaSearchCV	Hyperparameter search with cross-validation.
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.ChainerPruningExtension

`class optuna.integration.`**`ChainerPruningExtension`**(*trial: optuna.trial.Trial*, *observation_key: str*, *pruner_trigger: TriggerType*)

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')`.

`__init__`(*trial: optuna.trial.Trial*, *observation_key: str*, *pruner_trigger: TriggerType*) → None

Methods

`__init__`(trial, observation_key, pruner_trigger)	

optuna.integration.ChainerMNStudy

`class optuna.integration.`**`ChainerMNStudy`**(*study: Study*, *comm: CommunicatorBase*)

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.

`__init__`(*study: Study*, *comm: CommunicatorBase*) → None

Methods

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

`optimize`(*func: Callable[[ChainerMNTrial, CommunicatorBase], float]*, *n_trials: Optional[int] = None*, *timeout: Optional[float] = None*, *catch: Tuple[Type[Exception], ...] = ()*) → None

Optimize an objective function.

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

optuna.integration.CatalystPruningCallback

`class optuna.integration.`**`CatalystPruningCallback`**(*trial: optuna.trial._trial.Trial*, *metric: str = 'loss'*)

Catalyst callback to prune unpromising trials.

See the example if you want to add a pruning callback which observes the accuracy of Catalyst's `SupervisedRunner`.

Parameters

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

- **metric** (*str*) – Name of a metric, which is passed to *catalyst.core.State.valid_metrics* dictionary to fetch the value of metric computed on validation set. Pruning decision is made based on this value.

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.

__init__(*trial: optuna.trial._trial.Trial*, *metric: str = 'loss'*) → None

Methods

`__init__`(trial[, metric])	
on_epoch_end(state)	

optuna.integration.PyCmaSampler

class optuna.integration.**PyCmaSampler**(*x0: Optional[Dict[str, Any]] = None*, *sigma0: Optional[float] = None*, *cma_stds: Optional[Dict[str, float]] = None*, *seed: Optional[int] = None*, *cma_opts: Optional[Dict[str, Any]] = None*, *n_startup_trials: int = 1*, *independent_sampler: Optional[BaseSampler] = None*, *warn_independent_sampling: bool = 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_uniform('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.

__init__(*x0: Optional[Dict[str, Any]] = None*, *sigma0: Optional[float] = None*, *cma_stds: Optional[Dict[str, float]] = None*, *seed: Optional[int] = None*, *cma_opts: Optional[Dict[str, Any]] = None*, *n_startup_trials: int = 1*, *independent_sampler: Optional[BaseSampler] = None*, *warn_independent_sampling: bool = True*) → None

Methods

__init__([x0, sigma0, cma_stds, seed, ...])	
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: Study*, *trial: FrozenTrial*) → Dict[str, BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

sample_independent(*study: Study*, *trial: FrozenTrial*, *param_name: str*, *param_distribution: BaseDistribution*) → float

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: Study*, *trial: FrozenTrial*, *search_space: Dict[str, BaseDistribution]*) → Dict[str, float]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.integration.CmaEsSampler

class optuna.integration.**CmaEsSampler**(*x0: Optional[Dict[str, Any]] = None*, *sigma0: Optional[float] = None*, *cma_stds: Optional[Dict[str, float]] = None*, *seed: Optional[int] = None*, *cma_opts: Optional[Dict[str, Any]] = None*, *n_startup_trials: int = 1*, *independent_sampler: Optional[BaseSampler] = None*, *warn_independent_sampling: bool = 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*.

__init__(*x0: Optional[Dict[str, Any]] = None*, *sigma0: Optional[float] = None*, *cma_stds: Optional[Dict[str, float]] = None*, *seed: Optional[int] = None*, *cma_opts: Optional[Dict[str, Any]] = None*, *n_startup_trials: int = 1*, *independent_sampler: Optional[BaseSampler] = None*, *warn_independent_sampling: bool = True*) → None

Methods

__init__([x0, sigma0, cma_stds, seed, ...])	
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: Study*, *trial: FrozenTrial*) → Dict[str, BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

sample_independent(*study: Study, trial: FrozenTrial, param_name: str, param_distribution: BaseDistribution*) → float

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: Study, trial: FrozenTrial, search_space: Dict[str, BaseDistribution]*) → Dict[str, float]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.integration.FastAIPruningCallback

class optuna.integration.**FastAIPruningCallback**(*learn: Learner*, *trial:* optuna.trial.Trial, *monitor: str*)

FastAI callback to prune unpromising trials for fastai.

Note: This callback is for fastai<2.0, not the coming version developed in fastai/fastai_dev.

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.Callback reference for further details.

__init__(*learn: Learner*, *trial:* optuna.trial.Trial, *monitor: str*) → None

Methods

__init__(learn, trial, monitor)
on_epoch_end(epoch, **kwargs)

optuna.integration.KerasPruningCallback

class optuna.integration.**KerasPruningCallback**(*trial: optuna.trial._trial.Trial*, *monitor: str*, *interval: int = 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.

__init__(*trial: optuna.trial._trial.Trial*, *monitor: str*, *interval: int = 1*) → None

Methods

`__init__`(trial, monitor[, interval])	
on_epoch_end(epoch[, logs])	

optuna.integration.LightGBMPruningCallback

class optuna.integration.**LightGBMPruningCallback**(*trial: optuna.trial._trial.Trial*, *metric: str*, *valid_name: str = '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.

__init__(*trial: optuna.trial._trial.Trial*, *metric: str*, *valid_name: str = 'valid_0'*) → None

Methods

`__init__`(trial, metric[, valid_name])	

optuna.integration.lightgbm.train

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

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.

optuna.integration.lightgbm.LightGBMTuner

class `optuna.integration.lightgbm.LightGBMTuner`(*params: Dict[str, Any], train_set: lgb.Dataset, num_boost_round: int = 1000, valid_sets: Optional[VALID_SET_TYPE] = None, valid_names: Optional[Any] = None, fobj: Optional[Callable[[...], Any]] = None, feval: Optional[Callable[[...], Any]] = None, feature_name: str = 'auto', categorical_feature: str = 'auto', early_stopping_rounds: Optional[int] = None, evals_result: Optional[Dict[Any, Any]] = None, verbose_eval: Optional[Union[bool, int]] = True, learning_rates: Optional[List[float]] = None, keep_training_booster: Optional[bool] = False, callbacks: Optional[List[Callable[[...], Any]]] = None, time_budget: Optional[int] = None, sample_size: Optional[int] = None, study: Optional[optuna.study.Study] = None, optuna_callbacks: Optional[List[Callable[[optuna.study.Study, optuna.trial._frozen.FrozenTrial], None]]] = None, model_dir: Optional[str] = None, verbosity: Optional[int] = None, show_progress_bar: bool = 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.

`__init__`(*params: Dict[str, Any], train_set: lgb.Dataset, num_boost_round: int = 1000, valid_sets: Optional[VALID_SET_TYPE] = None, valid_names: Optional[Any] = None, fobj: Optional[Callable[[...], Any]] = None, feval: Optional[Callable[[...], Any]] = None, feature_name: str = 'auto', categorical_feature: str = 'auto', early_stopping_rounds: Optional[int] = None, evals_result: Optional[Dict[Any, Any]] = None, verbose_eval: Optional[Union[bool, int]] = True, learning_rates: Optional[List[float]] = None, keep_training_booster: Optional[bool] = False, callbacks: Optional[List[Callable[[...], Any]]] = None, time_budget: Optional[int] = None, sample_size: Optional[int] = None, study: Optional[*optuna.study.Study*] = None, optuna_callbacks: Optional[List[Callable[[*optuna.study.Study*, *optuna.trial._frozen.FrozenTrial*], None]]] = None, model_dir: Optional[str] = None, verbosity: Optional[int] = None, show_progress_bar: bool = True*) → None

Methods

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

Attributes

`best_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: lgb.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: Dict[str, Any]

Return parameters of the best booster.

property best_score: float

Return the score of the best booster.

get_best_booster() → lgb.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` arguments of *`__init__()`* when you resume tuning or you run tuning in parallel.

`run`() → None

Perform the hyperparameter-tuning with given parameters.

`sample_train_set`() → None

Make subset of *self.train_set* Dataset object.

optuna.integration.lightgbm.LightGBMTunerCV

`class optuna.integration.lightgbm.`**`LightGBMTunerCV`**(*params: Dict[str, Any], train_set: lgb.Dataset, num_boost_round: int = 1000, folds: Optional[Union[Generator[Tuple[int, int], None, None], Iterator[Tuple[int, int]], BaseCrossValidator]] = None, nfold: int = 5, stratified: bool = True, shuffle: bool = True, fobj: Optional[Callable[[...], Any]] = None, feval: Optional[Callable[[...], Any]] = None, feature_name: str = 'auto', categorical_feature: str = 'auto', early_stopping_rounds: Optional[int] = None, fpreproc: Optional[Callable[[...], Any]] = None, verbose_eval: Optional[Union[bool, int]] = True, show_stdv: bool = True, seed: int = 0, callbacks: Optional[List[Callable[[...], Any]]] = None, time_budget: Optional[int] = None, sample_size: Optional[int] = None, study: Optional[*optuna.study.Study*] = None, optuna_callbacks: Optional[List[Callable[[*optuna.study.Study*, *optuna.trial._frozen.FrozenTrial*], None]]] = None, verbosity: Optional[int] = None, show_progress_bar: bool = True*)

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() .

- **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.

__init__(*params: Dict[str, Any], train_set: lgb.Dataset, num_boost_round: int = 1000, folds: Optional[Union[Generator[Tuple[int, int], None, None], Iterator[Tuple[int, int]], BaseCrossValidator]] = None, nfold: int = 5, stratified: bool = True, shuffle: bool = True, fobj: Optional[Callable[[...], Any]] = None, feval: Optional[Callable[[...], Any]] = None, feature_name: str = 'auto', categorical_feature: str = 'auto', early_stopping_rounds: Optional[int] = None, fpreproc: Optional[Callable[[...], Any]] = None, verbose_eval: Optional[Union[bool, int]] = True, show_stdv: bool = True, seed: int = 0, callbacks: Optional[List[Callable[[...], Any]]] = None, time_budget: Optional[int] = None, sample_size: Optional[int] = None, study: Optional[*optuna.study.Study*] = None, optuna_callbacks: Optional[List[Callable[[*optuna.study.Study, optuna.trial._frozen.FrozenTrial*], None]]] = None, verbosity: Optional[int] = None, show_progress_bar: bool = True*) → None

Methods

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

Attributes

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

property best_params: `Dict[str, Any]`

Return parameters of the best booster.

property best_score: `float`

Return the score of the best booster.

run() → None

Perform the hyperparameter-tuning with given parameters.

sample_train_set() → None

Make subset of *self.train_set* Dataset object.

optuna.integration.MLflowCallback

class optuna.integration.**MLflowCallback**(*tracking_uri: Optional[str] = None*, *metric_name: str = 'value'*)

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.

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

def objective(trial):
    x = trial.suggest_uniform('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.

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.

`__init__`(*tracking_uri: Optional[str] = None*, *metric_name: str = 'value'*) → None

Methods

`__init__`([tracking_uri, metric_name])	

optuna.integration.MXNetPruningCallback

`class optuna.integration.`**`MXNetPruningCallback`**(*trial: optuna.trial._trial.Trial*, *eval_metric: str*)

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.

`__init__`(*trial: optuna.trial._trial.Trial*, *eval_metric: str*) → None

Methods

`__init__`(trial, eval_metric)	

optuna.integration.PyTorchIgnitePruningHandler

`class optuna.integration.`**`PyTorchIgnitePruningHandler`**(*trial: Trial*, *metric: str*, *trainer: Engine*)

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.

`__init__`(*trial: Trial*, *metric: str*, *trainer: Engine*) → None

Methods

__init__(trial, metric, trainer)	

optuna.integration.PyTorchLightningPruningCallback

class optuna.integration.**PyTorchLightningPruningCallback**(*trial:* optuna.trial._trial.Trial, *monitor: str*)

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_end` and the names thus depend on how this dictionary is formatted.

__init__(*trial:* optuna.trial._trial.Trial, *monitor: str*) → None

Methods

__init__(trial, monitor)	
on_epoch_end(trainer, pl_module)	
on_validation_end(trainer, pl_module)	

optuna.integration.SkoptSampler

class optuna.integration.**SkoptSampler**(*independent_sampler: Optional[*optuna.samplers._base.BaseSampler*] = None, warn_independent_sampling: bool = True, skopt_kwargs: Optional[Dict[str, Any]] = None, n_startup_trials: int = 1, *, consider_pruned_trials: bool = 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_uniform('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.

`__init__`(*independent_sampler: Optional[optuna.samplers._base.BaseSampler] = None, warn_independent_sampling: bool = True, skopt_kwargs: Optional[Dict[str, Any]] = None, n_startup_trials: int = 1, *, consider_pruned_trials: bool = False*) → None

Methods

`__init__`([independent_sampler, ...])	
`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: optuna.study.Study, trial: optuna.trial._frozen.FrozenTrial*) → Dict[str, optuna.distributions.BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

`reseed_rng`() → None

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.

`sample_independent`(*study: optuna.study.Study, trial: optuna.trial._frozen.FrozenTrial, param_name: str, param_distribution: optuna.distributions.BaseDistribution*) → Any

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`** – Target study object.
- **`trial`** – Target trial object. Take a copy before modifying this object.
- **`param_name`** – Name of the sampled parameter.
- **`param_distribution`** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

`sample_relative`(*study: optuna.study.Study*, *trial: optuna.trial._frozen.FrozenTrial*, *search_space: Dict[str, optuna.distributions.BaseDistribution]*) → Dict[str, Any]

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`** – Target study object.
- **`trial`** – Target trial object. Take a copy before modifying this object.
- **`search_space`** – The search space returned by *`infer_relative_search_space()`*.

Returns A dictionary containing the parameter names and the values.

optuna.integration.TensorBoardCallback

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

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.

`__init__`(*dirname: str*, *metric_name: str*) → None

Methods

`__init__`(dirname, metric_name)	

optuna.integration.TensorFlowPruningHook

class `optuna.integration.`**`TensorFlowPruningHook`**(*trial: optuna.trial.Trial*, *estimator: tf.estimator.Estimator*, *metric: str*, *run_every_steps: int*)

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.

`__init__`(*trial: optuna.trial.Trial*, *estimator: tf.estimator.Estimator*, *metric: str*, *run_every_steps: int*) → None

Methods

`__init__`(trial, estimator, metric, ...)	
`after_run`(run_context, run_values)	
`before_run`(run_context)	
`begin`()	

optuna.integration.TFKerasPruningCallback

class `optuna.integration.`**`TFKerasPruningCallback`**(*trial: optuna.trial._trial.Trial*, *monitor: str*)

tf.keras callback to prune unpromising trials.

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

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`.

`__init__`(*trial: optuna.trial._trial.Trial*, *monitor: str*) → None

Methods

`__init__`(trial, monitor)	
on_epoch_end(epoch[, logs])	

optuna.integration.XGBoostPruningCallback

`class optuna.integration.`**`XGBoostPruningCallback`**(*trial:* optuna.trial._trial.Trial, *observation_key:* *str*)

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.

`__init__`(*trial:* optuna.trial._trial.Trial, *observation_key:* *str*) → None

Methods

`__init__`(trial, observation_key)	

optuna.integration.OptunaSearchCV

`class optuna.integration.`**`OptunaSearchCV`**(*estimator: BaseEstimator*, *param_distributions: Mapping[str, distributions.BaseDistribution]*, *cv: Optional[Union[BaseCrossValidator, int]] = 5*, *enable_pruning: bool = False*, *error_score: Union[Number, str] = nan*, *max_iter: int = 1000*, *n_jobs: int = 1*, *n_trials: int = 10*, *random_state: Optional[Union[int, np.random.RandomState]] = None*, *refit: bool = True*, *return_train_score: bool = False*, *scoring: Optional[Union[Callable[..., float], str]] = None*, *study: Optional[study_module.Study] = None*, *subsample: Union[float, int] = 1.0*, *timeout: Optional[float] = None*, *verbose: int = 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 parallel jobs. `-1` means using all processors.
- **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, 1e+10)
}
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.

__init__(*estimator: BaseEstimator*, *param_distributions: Mapping[str, distributions.BaseDistribution]*, *cv: Optional[Union[BaseCrossValidator, int]] = 5*, *enable_pruning: bool = False*, *error_score: Union[Number, str] = nan*, *max_iter: int = 1000*, *n_jobs: int = 1*, *n_trials: int = 10*, *random_state: Optional[Union[int, np.random.RandomState]] = None*, *refit: bool = True*, *return_train_score: bool = False*, *scoring: Optional[Union[Callable[..., float], str]] = None*, *study: Optional[study_module.Study] = None*, *subsample: Union[float, int] = 1.0*, *timeout: Optional[float] = None*, *verbose: int = 0*) → None

Methods

`__init__`(estimator, param_distributions[, ...])	
`fit`(X[, y, groups])	Run fit with all sets of parameters.
`score`(X[, y])	Return the score on the given data.

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: TwoDimArrayLikeType*, *y: Optional[Union[OneDimArrayLikeType, TwoDimArrayLikeType]] = None*, *groups: Optional[OneDimArrayLikeType] = None*, ***fit_params: Any*) → *OptunaSearchCV*

Run fit with all sets of parameters.

Parameters

- **X** – Training data.
- **y** – Target variable.
- **groups** – Group labels for the samples used while splitting the dataset into train/validation set.
- ****fit_params** – Parameters passed to `fit` on the estimator.

Returns Return self.

Return type self

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: TwoDimArrayLikeType*, *y: Optional[Union[OneDimArrayLikeType, TwoDimArrayLikeType]] = None*) → float

Return the score on the given data.

Parameters

- **X** – Data.
- **y** – 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`.

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*.

optuna.integration.AllenNLPExecutor

`class optuna.integration.`**`AllenNLPExecutor`**(*trial: optuna.trial._trial.Trial*, *config_file: str*, *serialization_dir: str*, *metrics: str = 'best_validation_accuracy'*, ***, *include_package: Optional[Union[str, List[str]]] = None*)

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 and config file.

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`.
- **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.

`__init__`(*trial: optuna.trial._trial.Trial*, *config_file: str*, *serialization_dir: str*, *metrics: str = 'best_validation_accuracy'*, ***, *include_package: Optional[Union[str, List[str]]] = None*)

Methods

`__init__`(trial, config_file, serialization_dir)	
`run`()	Train a model using AllenNLP.

run() → float

Train a model using AllenNLP.

optuna.integration.allennlp.dump_best_config

optuna.integration.allennlp.**dump_best_config**(*input_config_file: str*, *output_config_file: str*, *study: optuna.study.Study*) → None

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

Parameters

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

optuna.integration.AllenNLPPruningCallback

class optuna.integration.**AllenNLPPruningCallback**(*trial: optuna.trial._trial.Trial*, *monitor: str*)

AllenNLP callback to prune unpromising trials.

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

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.

__init__(*trial: optuna.trial._trial.Trial*, *monitor: str*)

Methods

`__init__`(trial, monitor)

6.3.7 optuna.logging

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**() → int

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

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

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: int*) → None

Set the level for the Optuna's root logger.

Parameters **verbosity** – Logging level, e.g., `optuna.logging.DEBUG` and `optuna.logging.INFO`.

optuna.logging.disable_default_handler

optuna.logging.**disable_default_handler**() → None

Disable the default handler of the Optuna's root logger.

Example

Stop and then resume logging to `sys.stderr`.

```python
import optuna

study = optuna.create_study()

# There are no logs in sys.stderr.
optuna.logging.disable_default_handler()
```

(continues on next page)

(continued from previous page)

```
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: ...
# ...
```

optuna.logging.enable_default_handler

optuna.logging.**enable_default_handler**() → None

Enable the default handler of the Optuna's root logger.

Please refer to the example shown in *disable_default_handler()*.

optuna.logging.disable_propagation

optuna.logging.**disable_propagation**() → None

Disable propagation of the library log outputs.

Note that log propagation is disabled by default.

optuna.logging.enable_propagation

optuna.logging.**enable_propagation**() → None

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)
```

(continues on next page)

(continued from previous page)

```
with open('foo.log') as f:
    assert f.readline() == "Start optimization.\n"
    assert f.readline().startswith("Trial 0 finished with value:")
```

6.3.8 optuna.multi_objective

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.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.

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.

__init__(**args*, ***kwargs*) → None

Methods

__init__(*args, **kwargs)	
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
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:* optuna.multi_objective.study.MultiObjectiveStudy, *trial:* optuna.multi_objective.trial.FrozenMultiObjectiveTrial) → Dict[str, optuna.distributions.BaseDistribution]

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** – Target study object.
- **trial** – Target trial object.

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

See also:

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

abstract sample_independent(*study: optuna.multi_objective.study.MultiObjectiveStudy, trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial, param_name: str, param_distribution: optuna.distributions.BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

abstract sample_relative(*study: optuna.multi_objective.study.MultiObjectiveStudy, trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial, search_space: Dict[str, optuna.distributions.BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.multi_objective.samplers.NSGAIIMultiObjectiveSampler

class optuna.multi_objective.samplers.**NSGAIIMultiObjectiveSampler**(*population_size: int = 50, mutation_prob: Optional[float] = None, crossover_prob: float = 0.9, swapping_prob: float = 0.5, seed: Optional[int] = 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.

Note: Added in v1.5.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v1.5.0.

__init__(*population_size: int = 50, mutation_prob: Optional[float] = None, crossover_prob: float = 0.9, swapping_prob: float = 0.5, seed: Optional[int] = None*) → None

Methods

__init__([population_size, mutation_prob, ...])	
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
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: optuna.multi_objective.study.MultiObjectiveStudy, trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial*) → Dict[str, optuna.distributions.BaseDistribution]

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** – Target study object.
- **trial** – Target trial object.

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

See also:

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

sample_independent(*study: optuna.multi_objective.study.MultiObjectiveStudy*, *trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial*, *param_name: str*, *param_distribution: optuna.distributions.BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: optuna.multi_objective.study.MultiObjectiveStudy*, *trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial*, *search_space: Dict[str, optuna.distributions.BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.multi_objective.samplers.RandomMultiObjectiveSampler

class optuna.multi_objective.samplers.**RandomMultiObjectiveSampler**(*seed: Optional[int] = 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_uniform('x', -5, 5)
    y = trial.suggest_uniform('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.

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.

__init__(*seed: Optional[int] = None*) → None

Methods

__init__([seed])	
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
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:* optuna.multi_objective.study.MultiObjectiveStudy, *trial:* optuna.multi_objective.trial.FrozenMultiObjectiveTrial) → Dict[str, optuna.distributions.BaseDistribution]

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` – Target study object.
- `trial` – Target trial object.

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

See also:

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

`sample_independent`(*study: optuna.multi_objective.study.MultiObjectiveStudy*, *trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial*, *param_name: str*, *param_distribution: optuna.distributions.BaseDistribution*) → Any

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` – Target study object.
- `trial` – Target trial object.
- `param_name` – Name of the sampled parameter.
- `param_distribution` – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

`sample_relative`(*study: optuna.multi_objective.study.MultiObjectiveStudy*, *trial: optuna.multi_objective.trial.FrozenMultiObjectiveTrial*, *search_space: Dict[str, optuna.distributions.BaseDistribution]*) → Dict[str, Any]

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` – Target study object.
- `trial` – Target trial object.
- `search_space` – The search space returned by *`infer_relative_search_space()`*.

Returns A dictionary containing the parameter names and the values.

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:* optuna.study.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.

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.

__init__(*study:* optuna.study.Study)

Methods

__init__(study)	
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])	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: List[*optuna._study_direction.StudyDirection*]

Return the optimization direction list.

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

enqueue_trial(*params: Dict[str, Any]*) → None

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** – Parameter values to pass your objective function.

get_pareto_front_trials() → List[*optuna.multi_objective.trial.FrozenMultiObjectiveTrial*]

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.

get_trials(*deepcopy: bool = True*) → List[*optuna.multi_objective.trial.FrozenMultiObjectiveTrial*]

Return all trials in the study.

The returned trials are ordered by trial number.

For library users, it's recommended to use more handy `trials` property to get the trials instead.

Parameters **deepcopy** – 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.

Returns A list of `FrozenMultiObjectiveTrial` objects.

property n_objectives: int

Return the number of objectives.

Returns Number of objectives.

optimize(*objective: Callable[[optuna.multi_objective.trial.MultiObjectiveTrial], Sequence[float]], timeout: Optional[int] = None, n_trials: Optional[int] = None, n_jobs: int = 1, catch: Tuple[Type[Exception], ...] = (), callbacks: Optional[List[Callable[[optuna.multi_objective.study.MultiObjectiveStudy, optuna.multi_objective.trial.FrozenMultiObjectiveTrial], None]]] = None, gc_after_trial: bool = True, show_progress_bar: bool = False*) → None

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.

property sampler: optuna.multi_objective.samplers._base.BaseMultiObjectiveSampler

Return the sampler.

Returns A `BaseMultiObjectiveSampler` object.

set_system_attr(*key: str, value: Any*) → None

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** – A key string of the attribute.

- **value** – A value of the attribute. The value should be JSON serializable.

`set_user_attr`(*key: str*, *value: Any*) → None

Set a user attribute to the study.

Parameters

- **key** – A key string of the attribute.
- **value** – A value of the attribute. The value should be JSON serializable.

property system_attrs: Dict[str, Any]

Return system attributes.

Returns A dictionary containing all system attributes.

property trials: List[*optuna.multi_objective.trial.FrozenMultiObjectiveTrial*]

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)`.

Returns A list of *FrozenMultiObjectiveTrial* objects.

property user_attrs: Dict[str, Any]

Return user attributes.

Returns A dictionary containing all user attributes.

optuna.multi_objective.study.create_study

`optuna.multi_objective.study.create_study`(*directions: List[str]*, *study_name: Optional[str] = None*, *storage: Union[None, str, optuna.storages._base.BaseStorage] = None*, *sampler: Optional[optuna.multi_objective.samplers._base.BaseMultiObjectiveSampler] = None*, *load_if_exists: bool = False*) → *optuna.multi_objective.study.MultiObjectiveStudy*

Create a new *MultiObjectiveStudy*.

Parameters

- **directions** – Optimization direction for each objective value. Set `minimize` for minimization and `maximize` for maximization.
- **study_name** – Study's name. If this argument is set to None, a unique name is generated automatically.
- **storage** – 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** – 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** – 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.

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.

optuna.multi_objective.study.load_study

`optuna.multi_objective.study.`**`load_study`**(*study_name: str*, *storage: Union[str, optuna.storages._base.BaseStorage]*, *sampler: Optional[optuna.multi_objective.samplers._base.BaseMultiObjectiveSampler] = None*) → *optuna.multi_objective.study.MultiObjectiveStudy*

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

Parameters

- **study_name** – Study's name. Each study has a unique name as an identifier.
- **storage** – Database URL such as `sqlite:///example.db`. Please see also the documentation of *`create_study()`* for further details.
- **sampler** – 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.

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.

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:* optuna.trial._trial.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.

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.

`__init__`(*trial:* optuna.trial._trial.Trial)

Methods

`__init__`(trial)	
`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.
`user_attrs`	Return user attributes.

`property datetime_start: Optional[datetime.datetime]`

Return start datetime.

Returns Datetime where the *`Trial`* started.

property distributions: Dict[str, optuna.distributions.BaseDistribution]

Return distributions of parameters to be optimized.

Returns A dictionary containing all distributions.

property number: int

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

Returns A trial number.

property params: Dict[str, Any]

Return parameters to be optimized.

Returns A dictionary containing all parameters.

report(*values: Sequence[float]*, *step: int*) → None

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

set_system_attr(*key: str*, *value: Any*) → None

Set system attributes to the trial.

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

set_user_attr(*key: str*, *value: Any*) → None

Set user attributes to the trial.

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

suggest_categorical(*name: str*, *choices: Sequence[Union[None, bool, int, float, str]]*) → Union[None, bool, int, float, str]

Suggest a value for the categorical parameter.

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

suggest_discrete_uniform(*name: str*, *low: float*, *high: float*, *q: float*) → float

Suggest a value for the discrete parameter.

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

suggest_float(*name: str*, *low: float*, *high: float*, ***, *step: Optional[float] = None*, *log: bool = False*) → float

Suggest a value for the floating point parameter.

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

suggest_int(*name: str, low: int, high: int, step: int = 1, log: bool = False*) → int

Suggest a value for the integer parameter.

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

suggest_loguniform(*name: str, low: float, high: float*) → float

Suggest a value for the continuous parameter.

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

suggest_uniform(*name: str, low: float, high: float*) → float

Suggest a value for the continuous parameter.

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

property system_attrs: Dict[str, Any]

Return system attributes.

Returns A dictionary containing all system attributes.

property user_attrs: Dict[str, Any]

Return user attributes.

Returns A dictionary containing all user attributes.

optuna.multi_objective.trial.FrozenMultiObjectiveTrial

class optuna.multi_objective.trial.**FrozenMultiObjectiveTrial**(*n_objectives: int, trial: optuna.trial._frozen.FrozenTrial*)

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()*.

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.

__init__(*n_objectives: int*, *trial:* optuna.trial._frozen.FrozenTrial)

Methods

__init__(n_objectives, trial)

Attributes

datetime_complete
datetime_start
distributions
last_step
number
params
state
system_attrs
user_attrs

optuna.multi_objective.visualization

Note: `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: optuna.multi_objective.study.MultiObjectiveStudy, names: Optional[List[str]] = None*) → go.Figure

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)

optuna.multi_objective.visualization.plot_pareto_front(study)
```

Parameters

- **study** – A *MultiObjectiveStudy* object whose trials are plotted for their objective values.
- **names** – Objective name list used as the axis titles. If None is specified, "Objective {objective_index}" is used instead.

Returns A plotly.graph_objs.Figure object.

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

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.

6.3.9 optuna.pruners

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.

__init__()

Methods

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

abstract prune(*study:* Study, *trial:* FrozenTrial) → bool

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

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

Parameters

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

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

optuna.pruners.MedianPruner

class optuna.pruners.**MedianPruner**(*n_startup_trials:* int = *5*, *n_warmup_steps:* int = *0*, *interval_steps:* int = *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)
```

(continues on next page)

(continued from previous page)

```python
def objective(trial):
    alpha = trial.suggest_uniform('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.

`__init__`(*n_startup_trials: int = 5, n_warmup_steps: int = 0, interval_steps: int = 1*) → None

Methods

`__init__`([n_startup_trials, n_warmup_steps, ...])	
`prune`(study, trial)	Judge whether the trial should be pruned based on the reported values.

`prune`(*study: Study, trial: FrozenTrial*) → bool

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

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

Parameters

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

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

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

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_uniform('alpha', 0.0, 1.0)
    clf = SGDClassifier(alpha=alpha)
    n_train_iter = 100

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

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

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

    return clf.score(X_valid, y_valid)

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

__init__()

Methods

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

prune(*study:* Study, *trial:* FrozenTrial) → bool

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

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

Parameters

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

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

optuna.pruners.PercentilePruner

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

Pruner to keep the specified percentile of the trials.

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

Example

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

import optuna

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

def objective(trial):
    alpha = trial.suggest_uniform('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)
```

(continues on next page)

(continued from previous page)

```
        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.

`__init__`(*percentile: float*, *n_startup_trials: int = 5*, *n_warmup_steps: int = 0*, *interval_steps: int = 1*) → None

Methods

`__init__`(percentile[, n_startup_trials, ...])	
`prune`(study, trial)	Judge whether the trial should be pruned based on the reported values.

`prune`(*study: Study*, *trial: FrozenTrial*) → bool

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

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

Parameters

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

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

optuna.pruners.SuccessiveHalvingPruner

class optuna.pruners.**SuccessiveHalvingPruner**(*min_resource: Union[str, int] = 'auto', reduction_factor: int = 4, min_early_stopping_rate: int = 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.py, `EPOCH` number in chainer_integration.py).

See also:

Please refer to `report()`.

Example

We minimize an objective function with `SuccessiveHalvingPruner`.

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

import optuna

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

def objective(trial):
    alpha = trial.suggest_uniform('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).

`__init__`(*min_resource: Union[str, int] = 'auto', reduction_factor: int = 4, min_early_stopping_rate: int = 0*) → None

Methods

`__init__`([min_resource, reduction_factor, ...])	
`prune`(study, trial)	Judge whether the trial should be pruned based on the reported values.

`prune`(*study:* Study, *trial:* FrozenTrial) → bool

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

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

Parameters

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

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

optuna.pruners.HyperbandPruner

class optuna.pruners.**HyperbandPruner**(*min_resource: int = 1, max_resource: Union[str, int] = 'auto', reduction_factor: int = 3*)

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.

```
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_test, y_train, y_test = train_test_split(X, y)
classes = np.unique(y)
```

(continues on next page)

(continued from previous page)

```
n_train_iter = 100

def objective(trial):
    alpha = trial.suggest_uniform('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 represents and should match the maximum iteration steps (e.g., the number of epochs for neural networks). When this argument is "auto", the maximum resource is estimated according to the completed trials. The default value of this argument is "auto".

  Note: With "auto", the maximum resource will be the largest step reported by *`report()`* in the first, or one of the first if trained in parallel, completed trial. No trials will be pruned until the maximum resource is determined.

  Note: If the step of the last intermediate value may change with each trial, please manually specify the maximum possible step to `max_resource`.

- **`reduction_factor`** – A parameter for specifying reduction factor of promotable trials noted as η in the paper. See the details for *`SuccessiveHalvingPruner`*.

`__init__`(*min_resource: int = 1, max_resource: Union[str, int] = 'auto', reduction_factor: int = 3*) → None

Methods

`__init__`([min_resource, max_resource, ...])	
`prune`(study, trial)	Judge whether the trial should be pruned based on the reported values.

`prune`(*study: optuna.study.Study, trial: optuna.trial._frozen.FrozenTrial*) → bool

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

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

Parameters

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

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

optuna.pruners.ThresholdPruner

`class optuna.pruners.`**`ThresholdPruner`**(*lower: Optional[float] = None, upper: Optional[float] = None, n_warmup_steps: int = 0, interval_steps: int = 1*)

Pruner to detect outlying metrics of the trials.

Prune if a metric exceeds upper threshold, falls behind lower threshold or reaches `nan`.

Example

```python
from optuna import create_study
from optuna.pruners import ThresholdPruner
from optuna import TrialPruned

def objective_for_upper(trial):
    for step, y in enumerate(ys_for_upper):
        trial.report(y, step)

        if trial.should_prune():
            raise TrialPruned()
    return ys_for_upper[-1]


def objective_for_lower(trial):
    for step, y in enumerate(ys_for_lower):
        trial.report(y, step)

        if trial.should_prune():
```

(continues on next page)

(continued from previous page)

```
            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]
n_trial_step = 5

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 until the trial exceeds the given number of step.

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.

`__init__`(*lower: Optional[float] = None*, *upper: Optional[float] = None*, *n_warmup_steps: int = 0*, *interval_steps: int = 1*) → None

Methods

`__init__`([lower, upper, n_warmup_steps, ...])	
`prune`(study, trial)	Judge whether the trial should be pruned based on the reported values.

`prune`(*study: optuna.study.Study*, *trial: optuna.trial._frozen.FrozenTrial*) → bool

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

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

Parameters

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

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

6.3.10 optuna.samplers

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 CMA-ES 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.

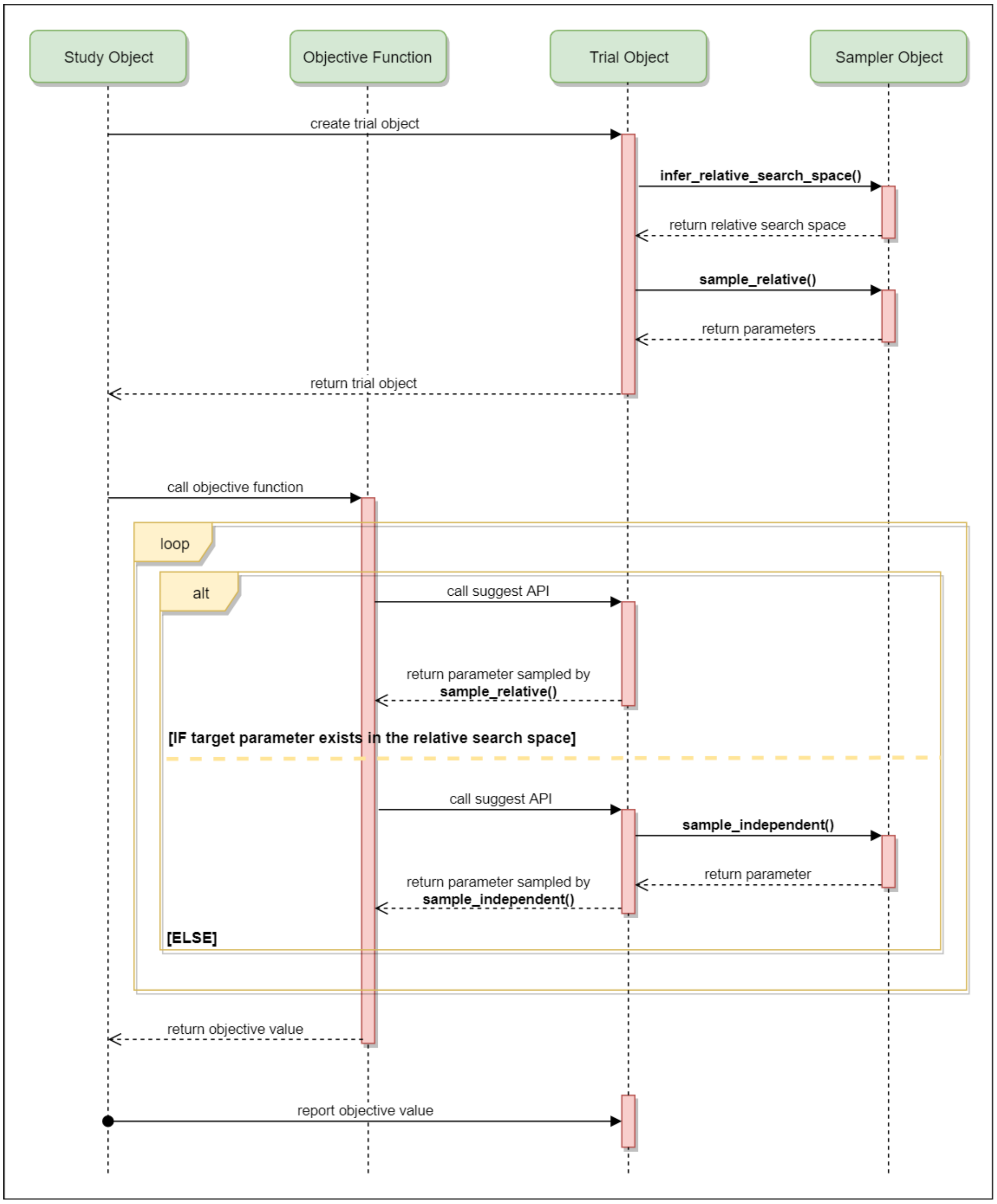
Optuna combines two types of sampling strategies, which are called *relative sampling* and *independent sampling*.

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

`__init__()`

Methods

`__init__`()	
`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: Study*, *trial: FrozenTrial*) → Dict[str, BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

abstract sample_independent(*study: Study*, *trial: FrozenTrial*, *param_name: str*, *param_distribution: BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

abstract sample_relative(*study: Study*, *trial: FrozenTrial*, *search_space: Dict[str, BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.samplers.GridSampler

class optuna.samplers.**GridSampler**(*search_space: Mapping[str, Sequence[GridValueType]]*)

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_uniform('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, n_trials=3*3)
```

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_discrete_uniform('x', -5, 5, 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)
```

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

`__init__`(*search_space: Mapping[str, Sequence[GridValueType]]*) → None

Methods

`__init__`(search_space)	
`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: Study*, *trial: FrozenTrial*) → Dict[str, BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

Parameters

- **`study`** – Target study object.
- **`trial`** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

sample_independent(*study: Study*, *trial: FrozenTrial*, *param_name: str*, *param_distribution: BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: Study*, *trial: FrozenTrial*, *search_space: Dict[str, BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.samplers.RandomSampler

class optuna.samplers.**RandomSampler**(*seed: Optional[int] = 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_uniform('x', -5, 5)
    return x**2

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

Args: seed: Seed for random number generator.

__init__(*seed: Optional[int] = None*) → None

Methods

__init__([seed])	
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: Study*, *trial: FrozenTrial*) → Dict[str, BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

sample_independent(*study: Study*, *trial: FrozenTrial*, *param_name: str*, *param_distribution: distributions.BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: Study*, *trial: FrozenTrial*, *search_space: Dict[str, BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.samplers.TPESampler

class `optuna.samplers.TPESampler`(*consider_prior: bool = True, prior_weight: float = 1.0, consider_magic_clip: bool = True, consider_endpoints: bool = False, n_startup_trials: int = 10, n_ei_candidates: int = 24, gamma: Callable[[int], int] = <function default_gamma>, weights: Callable[[int], np.ndarray] = <function default_weights>, seed: Optional[int] = None*)

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

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

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

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

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

Example

```
import optuna
from optuna.samplers import TPESampler

def objective(trial):
    x = trial.suggest_uniform('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.

__init__(*consider_prior: bool = True, prior_weight: float = 1.0, consider_magic_clip: bool = True, consider_endpoints: bool = False, n_startup_trials: int = 10, n_ei_candidates: int = 24, gamma: Callable[[int], int] = <function default_gamma>, weights: Callable[[int], np.ndarray] = <function default_weights>, seed: Optional[int] = None*) → None

Methods

__init__([consider_prior, prior_weight, ...])	
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.

static hyperopt_parameters() → Dict[str, Any]

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_uniform('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.

infer_relative_search_space(*study: Study*, *trial: FrozenTrial*) → Dict[str, BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using `sample_independent()` method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

sample_independent(*study: Study*, *trial: FrozenTrial*, *param_name: str*, *param_distribution: BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: Study*, *trial: FrozenTrial*, *search_space: Dict[str, BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.samplers.CmaEsSampler

class optuna.samplers.**CmaEsSampler**(*x0: Optional[Dict[str, Any]] = None, sigma0: Optional[float] = None, n_startup_trials: int = 1, independent_sampler: Optional[*optuna.samplers._base.BaseSampler*] = None, warn_independent_sampling: bool = True, seed: Optional[int] = None, *, consider_pruned_trials: bool = False*)

A Sampler using CMA-ES algorithm.

Example

Optimize a simple quadratic function by using *CmaEsSampler*.

```
import optuna

def objective(trial):
    x = trial.suggest_uniform('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.

See also:

You can also use *optuna.integration.CmaEsSampler* 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.

- **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.
- **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.

__init__(*x0: Optional[Dict[str, Any]] = None*, *sigma0: Optional[float] = None*, *n_startup_trials: int = 1*, *independent_sampler: Optional[*optuna.samplers._base.BaseSampler*] = None*, *warn_independent_sampling: bool = True*, *seed: Optional[int] = None*, ***, *consider_pruned_trials: bool = False*) → None

Methods

__init__([x0, sigma0, n_startup_trials, ...])	
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: optuna.study.Study*, *trial: optuna.trial._frozen.FrozenTrial*) → Dict[str, optuna.distributions.BaseDistribution]

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 pass to it. The parameters not contained in the search space will be sampled by using *sample_independent()* method.

Parameters

- **study** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.

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

See also:

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

reseed_rng() → None

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.

sample_independent(*study: optuna.study.Study*, *trial: optuna.trial._frozen.FrozenTrial*, *param_name: str*, *param_distribution: optuna.distributions.BaseDistribution*) → Any

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **param_name** – Name of the sampled parameter.
- **param_distribution** – Distribution object that specifies a prior and/or scale of the sampling algorithm.

Returns A parameter value.

sample_relative(*study: optuna.study.Study*, *trial: optuna.trial._frozen.FrozenTrial*, *search_space: Dict[str, optuna.distributions.BaseDistribution]*) → Dict[str, Any]

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** – Target study object.
- **trial** – Target trial object. Take a copy before modifying this object.
- **search_space** – The search space returned by *infer_relative_search_space()*.

Returns A dictionary containing the parameter names and the values.

optuna.samplers.IntersectionSearchSpace

class optuna.samplers.**IntersectionSearchSpace**

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.

__init__() → None

Methods

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

calculate(*study: optuna.study.BaseStudy*, *ordered_dict: bool = False*) → Dict[str, optuna.distributions.BaseDistribution]

Returns the intersection search space of the BaseStudy.

Parameters

- **study** – A study with completed trials.
- **ordered_dict** – 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.

optuna.samplers.intersection_search_space

`optuna.samplers.`**`intersection_search_space`**(*study: optuna.study.BaseStudy*, *ordered_dict: bool = False*) → Dict[str, optuna.distributions.BaseDistribution]

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`** – A study with completed trials.
- **`ordered_dict`** – 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.

6.3.11 optuna.storages

`optuna.storages.RDBStorage`	Storage class for RDB backend.
`optuna.storages.RedisStorage`	Storage class for Redis backend.

optuna.storages.RDBStorage

class `optuna.storages.`**`RDBStorage`**(*url: str*, *engine_kwargs: Optional[Dict[str, Any]] = None*, *skip_compatibility_check: bool = False*)

Storage class for RDB backend.

Note that library users can instantiate this class, but the attributes provided by this class are not supposed to be directly accessed by them.

Example

Create an *`RDBStorage`* instance with customized `pool_size` and `timeout` settings.

```python
import optuna

def objective(trial):
    x = trial.suggest_uniform('x', -100, 100)
    return x ** 2

storage = optuna.storages.RDBStorage(
    url='sqlite:///:memory:',
```

(continues on next page)

(continued from previous page)

```
    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.

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.

`__init__`(*url: str*, *engine_kwargs: Optional[Dict[str, Any]] = None*, *skip_compatibility_check: bool = False*) → None

Methods

`__init__`(url[, engine_kwargs, ...])	
`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.
`get_all_study_summaries`()	Read a list of `StudySummary` objects.
`get_all_trials`(study_id[, deepcopy])	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_head_version`()	Return the latest schema version.
`get_n_trials`(study_id[, state])	Count the number of trials in a study.
`get_study_direction`(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.

continues on next page

Table 1 – continued from previous page

`get_study_user_attrs`(study_id)	Read the user-defined attributes of a study.
`get_trial`(trial_id)	Read 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.
`read_trials_from_remote_storage`(study_id)	Make an internal cache of trials up-to-date.
`remove_session`()	Removes the current session.
`set_study_direction`(study_id, direction)	Register an optimization problem direction 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_value`(trial_id, value)	Set a return value of an objective function.
`upgrade`()	Upgrade the storage schema.

check_trial_is_updatable(*trial_id: int*, *trial_state:* optuna.trial._state.TrialState) → None

Check whether a trial state is updatable.

Parameters

- **trial_id** – ID of the trial. Only used for an error message.
- **trial_state** – Trial state to check.

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

create_new_study(*study_name: Optional[str] = None*) → int

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** – 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.

create_new_trial(*study_id: int*, *template_trial: Optional[*optuna.trial._frozen.FrozenTrial*] = None*) → int

Create and add a new trial to a study.

The returned trial ID is unique among all current and deleted trials.

Parameters

- **study_id** – ID of the study.
- **template_trial** – 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.

`delete_study`(*study_id: int*) → None

Delete a study.

Parameters **`study_id`** – ID of the study.

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

`get_all_study_summaries`() → List[*optuna._study_summary.StudySummary*]

Read a list of *`StudySummary`* objects.

Returns A list of *`StudySummary`* objects.

`get_all_trials`(*study_id: int*, *deepcopy: bool = True*) → List[*optuna.trial._frozen.FrozenTrial*]

Read all trials in a study.

Parameters

- **`study_id`** – ID of the study.
- **`deepcopy`** – Whether to copy the list of trials before returning. Set to `True` if you intend to update the list or elements of the list.

Returns List of trials in the study.

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

`get_all_versions`() → List[str]

Return the schema version list.

`get_best_trial`(*study_id: int*) → *optuna.trial._frozen.FrozenTrial*

Return the trial with the best value in a study.

Parameters **`study_id`** – 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.
- `ValueError` – If no trials have been completed.

`get_current_version`() → str

Return the schema version currently used by this storage.

`get_head_version`() → str

Return the latest schema version.

`get_n_trials`(*study_id: int*, *state: Optional[*optuna.trial._state.TrialState*] = None*) → int

Count the number of trials in a study.

Parameters

- **`study_id`** – ID of the study.
- **`state`** – *`TrialState`* to filter trials.

Returns Number of trials in the study.

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

get_study_direction(*study_id: int*) → *optuna._study_direction.StudyDirection*

Read whether a study maximizes or minimizes an objective.

Parameters **study_id** – ID of a study.

Returns Optimization direction of the study.

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

get_study_id_from_name(*study_name: str*) → int

Read the ID of a study.

Parameters **study_name** – Name of the study.

Returns ID of the study.

Raises KeyError – If no study with the matching study_name exists.

get_study_id_from_trial_id(*trial_id: int*) → int

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

Parameters **trial_id** – ID of the trial.

Returns ID of the study.

Raises KeyError – If no trial with the matching trial_id exists.

get_study_name_from_id(*study_id: int*) → str

Read the study name of a study.

Parameters **study_id** – ID of the study.

Returns Name of the study.

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

get_study_system_attrs(*study_id: int*) → Dict[str, Any]

Read the optuna-internal attributes of a study.

Parameters **study_id** – 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.

get_study_user_attrs(*study_id: int*) → Dict[str, Any]

Read the user-defined attributes of a study.

Parameters **study_id** – ID of the study.

Returns Dictionary with the user attributes of the study.

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

get_trial(*trial_id: int*) → *optuna.trial._frozen.FrozenTrial*

Read a trial.

Parameters **trial_id** – ID of the trial.

Returns Trial with a matching trial ID.

Raises KeyError – If no trial with the matching trial_id exists.

get_trial_number_from_id(*trial_id: int*) → int

Read the trial number of a trial.

Note: The trial number is only unique within a study, and is sequential.

Parameters **trial_id** – ID of the trial.

Returns Number of the trial.

Raises KeyError – If no trial with the matching `trial_id` exists.

get_trial_param(*trial_id: int*, *param_name: str*) → float

Read the parameter of a trial.

Parameters

- **trial_id** – ID of the trial.
- **param_name** – 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.

get_trial_params(*trial_id: int*) → Dict[str, Any]

Read the parameter dictionary of a trial.

Parameters **trial_id** – 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.

get_trial_system_attrs(*trial_id: int*) → Dict[str, Any]

Read the optuna-internal attributes of a trial.

Parameters **trial_id** – 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.

get_trial_user_attrs(*trial_id: int*) → Dict[str, Any]

Read the user-defined attributes of a trial.

Parameters **trial_id** – 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.

read_trials_from_remote_storage(*study_id: int*) → None

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

Parameters **study_id** – ID of the study.

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

remove_session() → None

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.

set_study_direction(*study_id: int*, *direction: optuna._study_direction.StudyDirection*) → None

Register an optimization problem direction to a study.

Parameters

- **study_id** – ID of the study.
- **direction** – Either `MAXIMIZE` or `MINIMIZE`.

Raises

- `KeyError` – If no study with the matching `study_id` exists.
- `ValueError` – If the direction is already set and the passed `direction` is the opposite direction or `NOT_SET`.

set_study_system_attr(*study_id: int*, *key: str*, *value: Any*) → None

Register an optuna-internal attribute to a study.

This method overwrites any existing attribute.

Parameters

- **study_id** – ID of the study.
- **key** – Attribute key.
- **value** – Attribute value. It should be JSON serializable.

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

set_study_user_attr(*study_id: int*, *key: str*, *value: Any*) → None

Register a user-defined attribute to a study.

This method overwrites any existing attribute.

Parameters

- **study_id** – ID of the study.
- **key** – Attribute key.
- **value** – Attribute value. It should be JSON serializable.

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

set_trial_intermediate_value(*trial_id: int*, *step: int*, *intermediate_value: float*) → None

Report an intermediate value of an objective function.

This method overwrites any existing intermediate value associated with the given step.

Parameters

- **trial_id** – ID of the trial.
- **step** – Step of the trial (e.g., the epoch when training a neural network).

- **intermediate_value** – Intermediate value corresponding to the step.

Raises

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

set_trial_param(*trial_id: int*, *param_name: str*, *param_value_internal: float*, *distribution: optuna.distributions.BaseDistribution*) → None

Set a parameter to a trial.

Parameters

- **trial_id** – ID of the trial.
- **param_name** – Name of the parameter.
- **param_value_internal** – Internal representation of the parameter value.
- **distribution** – Sampled distribution of the parameter.

Raises

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

set_trial_state(*trial_id: int*, *state: optuna.trial._state.TrialState*) → bool

Update the state of a trial.

Parameters

- **trial_id** – ID of the trial.
- **state** – 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.

set_trial_system_attr(*trial_id: int*, *key: str*, *value: Any*) → None

Set an optuna-internal attribute to a trial.

This method overwrites any existing attribute.

Parameters

- **trial_id** – ID of the trial.
- **key** – Attribute key.
- **value** – 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.

set_trial_user_attr(*trial_id: int*, *key: str*, *value: Any*) → None

Set a user-defined attribute to a trial.

This method overwrites any existing attribute.

Parameters

- **trial_id** – ID of the trial.
- **key** – Attribute key.
- **value** – 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.

set_trial_value(*trial_id: int*, *value: float*) → None

Set a return value of an objective function.

This method overwrites any existing trial value.

Parameters

- **trial_id** – ID of the trial.
- **value** – Value of the objective function.

Raises

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

upgrade() → None

Upgrade the storage schema.

optuna.storages.RedisStorage

class optuna.storages.**RedisStorage**(*url: str*)

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',
>>> )
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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.

`__init__`(*url: str*) → None

Methods

`__init__`(url)	
`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.
`get_all_study_summaries`()	Read a list of `StudySummary` objects.
`get_all_trials`(study_id[, deepcopy])	Read all trials in a study.
`get_best_trial`(study_id)	Return the trial with the best value in a study.
`get_n_trials`(study_id[, state])	Count the number of trials in a study.
`get_study_direction`(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_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.
`read_trials_from_remote_storage`(study_id)	Make an internal cache of trials up-to-date.
`remove_session`()	Clean up all connections to a database.
`set_study_direction`(study_id, direction)	Register an optimization problem direction 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.

continues on next page

Table 2 – continued from previous page

`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_value`(trial_id, value)	Set a return value of an objective function.

check_trial_is_updatable(*trial_id: int, trial_state: optuna.trial._state.TrialState*) → None

Check whether a trial state is updatable.

Parameters

- **trial_id** – ID of the trial. Only used for an error message.
- **trial_state** – Trial state to check.

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

create_new_study(*study_name: Optional[str] = None*) → int

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** – 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.

create_new_trial(*study_id: int, template_trial: Optional[FrozenTrial] = None*) → int

Create and add a new trial to a study.

The returned trial ID is unique among all current and deleted trials.

Parameters

- **study_id** – ID of the study.
- **template_trial** – 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.

delete_study(*study_id: int*) → None

Delete a study.

Parameters **study_id** – ID of the study.

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

get_all_study_summaries() → List[*StudySummary*]

Read a list of *StudySummary* objects.

Returns A list of *StudySummary* objects.

get_all_trials(*study_id: int, deepcopy: bool = True*) → List[*FrozenTrial*]

Read all trials in a study.

Parameters

- **study_id** – ID of the study.
- **deepcopy** – Whether to copy the list of trials before returning. Set to `True` if you intend to update the list or elements of the list.

Returns List of trials in the study.

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

get_best_trial(*study_id: int*) → *optuna.trial._frozen.FrozenTrial*

Return the trial with the best value in a study.

Parameters **study_id** – 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.
- `ValueError` – If no trials have been completed.

get_n_trials(*study_id: int*, *state: Optional[*TrialState*] = None*) → int

Count the number of trials in a study.

Parameters

- **study_id** – ID of the study.
- **state** – *TrialState* to filter trials.

Returns Number of trials in the study.

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

get_study_direction(*study_id: int*) → *optuna._study_direction.StudyDirection*

Read whether a study maximizes or minimizes an objective.

Parameters **study_id** – ID of a study.

Returns Optimization direction of the study.

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

get_study_id_from_name(*study_name: str*) → int

Read the ID of a study.

Parameters **study_name** – Name of the study.

Returns ID of the study.

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

get_study_id_from_trial_id(*trial_id: int*) → int

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

Parameters **trial_id** – ID of the trial.

Returns ID of the study.

Raises `KeyError` – If no trial with the matching `trial_id` exists.

get_study_name_from_id(*study_id: int*) → str

Read the study name of a study.

Parameters **study_id** – ID of the study.

Returns Name of the study.

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

`get_study_system_attrs`(*study_id: int*) → Dict[str, Any]

Read the optuna-internal attributes of a study.

Parameters **`study_id`** – 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.

`get_study_user_attrs`(*study_id: int*) → Dict[str, Any]

Read the user-defined attributes of a study.

Parameters **`study_id`** – ID of the study.

Returns Dictionary with the user attributes of the study.

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

`get_trial`(*trial_id: int*) → *optuna.trial._frozen.FrozenTrial*

Read a trial.

Parameters **`trial_id`** – ID of the trial.

Returns Trial with a matching trial ID.

Raises `KeyError` – If no trial with the matching `trial_id` exists.

`get_trial_number_from_id`(*trial_id: int*) → int

Read the trial number of a trial.

Note: The trial number is only unique within a study, and is sequential.

Parameters **`trial_id`** – ID of the trial.

Returns Number of the trial.

Raises `KeyError` – If no trial with the matching `trial_id` exists.

`get_trial_param`(*trial_id: int*, *param_name: str*) → float

Read the parameter of a trial.

Parameters

- **`trial_id`** – ID of the trial.
- **`param_name`** – 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.

`get_trial_params`(*trial_id: int*) → Dict[str, Any]

Read the parameter dictionary of a trial.

Parameters **`trial_id`** – 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.

`get_trial_system_attrs`(*trial_id: int*) → Dict[str, Any]

Read the optuna-internal attributes of a trial.

Parameters **`trial_id`** – 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.

`get_trial_user_attrs`(*trial_id: int*) → Dict[str, Any]

Read the user-defined attributes of a trial.

Parameters **`trial_id`** – 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.

`read_trials_from_remote_storage`(*study_id: int*) → None

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

Parameters **`study_id`** – ID of the study.

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

`remove_session`() → None

Clean up all connections to a database.

`set_study_direction`(*study_id: int*, *direction: optuna._study_direction.StudyDirection*) → None

Register an optimization problem direction to a study.

Parameters

- **`study_id`** – ID of the study.
- **`direction`** – Either *`MAXIMIZE`* or *`MINIMIZE`*.

Raises

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

`set_study_system_attr`(*study_id: int*, *key: str*, *value: Any*) → None

Register an optuna-internal attribute to a study.

This method overwrites any existing attribute.

Parameters

- **`study_id`** – ID of the study.
- **`key`** – Attribute key.
- **`value`** – Attribute value. It should be JSON serializable.

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

`set_study_user_attr`(*study_id: int*, *key: str*, *value: Any*) → None

Register a user-defined attribute to a study.

This method overwrites any existing attribute.

Parameters

- **`study_id`** – ID of the study.

- **key** – Attribute key.
- **value** – Attribute value. It should be JSON serializable.

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

set_trial_intermediate_value(*trial_id: int*, *step: int*, *intermediate_value: float*) → None

Report an intermediate value of an objective function.

This method overwrites any existing intermediate value associated with the given step.

Parameters

- **trial_id** – ID of the trial.
- **step** – Step of the trial (e.g., the epoch when training a neural network).
- **intermediate_value** – Intermediate value corresponding to the step.

Raises

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

set_trial_param(*trial_id: int*, *param_name: str*, *param_value_internal: float*, *distribution: optuna.distributions.BaseDistribution*) → None

Set a parameter to a trial.

Parameters

- **trial_id** – ID of the trial.
- **param_name** – Name of the parameter.
- **param_value_internal** – Internal representation of the parameter value.
- **distribution** – Sampled distribution of the parameter.

Raises

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

set_trial_state(*trial_id: int*, *state:* optuna.trial._state.TrialState) → bool

Update the state of a trial.

Parameters

- **trial_id** – ID of the trial.
- **state** – 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.

set_trial_system_attr(*trial_id: int*, *key: str*, *value: Any*) → None

Set an optuna-internal attribute to a trial.

This method overwrites any existing attribute.

Parameters

- **`trial_id`** – ID of the trial.
- **`key`** – Attribute key.
- **`value`** – 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.

`set_trial_user_attr`(*trial_id: int*, *key: str*, *value: Any*) → None

Set a user-defined attribute to a trial.

This method overwrites any existing attribute.

Parameters

- **`trial_id`** – ID of the trial.
- **`key`** – Attribute key.
- **`value`** – 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.

`set_trial_value`(*trial_id: int*, *value: float*) → None

Set a return value of an objective function.

This method overwrites any existing trial value.

Parameters

- **`trial_id`** – ID of the trial.
- **`value`** – Value of the objective function.

Raises

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

6.3.12 optuna.structs

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: int, state:* TrialState, *value: Optional[float], datetime_start: Optional[datetime], datetime_complete: Optional[datetime], params: Dict[str, Any], distributions: Dict[str, BaseDistribution], user_attrs: Dict[str, Any], system_attrs: Dict[str, Any], intermediate_values: Dict[int, float], trial_id: int*)

Status and results of a *Trial*.

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*.

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()*.

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.

class `optuna.structs.StudySummary`(*study_name: str, direction: _study_direction.StudyDirection, best_trial: Optional[FrozenTrial], user_attrs: Dict[str, Any], system_attrs: Dict[str, Any], n_trials: int, datetime_start: Optional[datetime], study_id: int*)

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`.

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.

> **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

`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: str, storage: Union[str, storages.BaseStorage], sampler: samplers.BaseSampler = None, pruner: pruners.BasePruner = 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.

__init__(*study_name: str, storage: Union[str, storages.BaseStorage], sampler: samplers.BaseSampler = None, pruner: pruners.BasePruner = None*) → None

Methods

`__init__`(study_name, storage[, sampler, pruner])	
`add_trial`(trial)	Add trial to study.
`enqueue_trial`(params)	Enqueue a trial with given parameter values.
`get_trials`([deepcopy])	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.
`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_value`	Return the best objective value in the study.
`direction`	Return the direction of the study.
`system_attrs`	Return system attributes.
`trials`	Return all trials in the study.
`user_attrs`	Return user attributes.

add_trial(*trial: optuna.trial._frozen.FrozenTrial*) → None

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_uniform('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`** – Trial to add.

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

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.

`property best_params`

Return parameters of the best trial in the study.

Returns A dictionary containing parameters of the best trial.

property best_trial

Return the best trial in the study.

Returns A `FrozenTrial` object of the best trial.

property best_value

Return the best objective value in the study.

Returns A float representing the best objective value.

property direction

Return the direction of the study.

Returns A *StudyDirection* object.

enqueue_trial(*params: Dict[str, Any]*) → None

Enqueue a trial with given parameter values.

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

Example

```
import optuna

def objective(trial):
    x = trial.suggest_uniform('x', 0, 10)
    return x ** 2

study = optuna.create_study()
study.enqueue_trial({'x': 5})
study.enqueue_trial({'x': 0})
study.optimize(objective, n_trials=2)

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

Parameters **params** – Parameter values to pass your objective function.

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: bool = True*) → List[*FrozenTrial*]

Return all trials in the study.

The returned trials are ordered by trial number.

For library users, it's recommended to use more handy *trials* property to get the trials instead.

Parameters **deepcopy** – 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.

Returns A list of `FrozenTrial` objects.

optimize(*func: ObjectiveFuncType*, *n_trials: Optional[int] = None*, *timeout: Optional[float] = None*, *n_jobs: int = 1*, *catch: Tuple[Type[Exception], ...] = ()*, *callbacks: Optional[List[Callable[[Study, FrozenTrial], None]]] = None*, *gc_after_trial: bool = False*, *show_progress_bar: bool = False*) → None

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'.

Parameters

- **func** – A callable that implements objective function.
- **n_trials** – 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** – 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** – The number of parallel jobs. If this argument is set to `-1`, the number is set to CPU count.
- **catch** – A study continues to run even when a trial raises one of the exceptions specified in this argument. Default is an empty tuple, i.e. the study will stop for any exception except for `TrialPruned`.
- **callbacks** – List 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** – 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** – 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` $\neq 1$.

set_system_attr(*key: str*, *value: Any*) → None

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** – A key string of the attribute.
- **value** – A value of the attribute. The value should be JSON serializable.

set_user_attr(*key: str*, *value: Any*) → None

Set a user attribute to the study.

Parameters

- **key** – A key string of the attribute.
- **value** – A value of the attribute. The value should be JSON serializable.

stop() → None

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.

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

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)`.

Returns A list of `FrozenTrial` objects.

trials_dataframe(*attrs: Tuple[str, ...] = ('number', 'value', 'datetime_start', 'datetime_complete', 'duration', 'params', 'user_attrs', 'system_attrs', 'state')*, *multi_index: bool = False*) → pd.DataFrame

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_uniform('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** – Specifies field names of `FrozenTrial` to include them to a DataFrame of trials.
- **multi_index** – 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*.

property user_attrs

Return user attributes.

Returns A dictionary containing all user attributes.

optuna.study.create_study

optuna.study.**create_study**(*storage: Union[None, str, storages.BaseStorage] = None*, *sampler: samplers.BaseSampler = None*, *pruner: pruners.BasePruner = None*, *study_name: Optional[str] = None*, *direction: str = 'minimize'*, *load_if_exists: bool = False*) → *Study*

Create a new *Study*.

Parameters

- **storage** – 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** – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *TPESampler* is used as the default. See also `samplers`.
- **pruner** – A pruner object that decides early stopping of unpromising trials. See also `pruners`.
- **study_name** – Study's name. If this argument is set to None, a unique name is generated automatically.
- **direction** – Direction of optimization. Set `minimize` for minimization and `maximize` for maximization.
- **load_if_exists** – 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 *Study* object.

See also:

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

optuna.study.load_study

`optuna.study.load_study`(*study_name: str, storage: Union[str, storages.BaseStorage], sampler: samplers.BaseSampler = None, pruner: pruners.BasePruner = None*) → *Study*

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

Parameters

- **study_name** – Study's name. Each study has a unique name as an identifier.
- **storage** – Database URL such as `sqlite:///example.db`. Please see also the documentation of `create_study()` for further details.
- **sampler** – A sampler object that implements background algorithm for value suggestion. If `None` is specified, `TPESampler` is used as the default. See also `samplers`.
- **pruner** – A pruner object that decides early stopping of unpromising trials. If `None` is specified, `MedianPruner` is used as the default. See also `pruners`.

See also:

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

optuna.study.delete_study

`optuna.study.delete_study`(*study_name: str, storage: Union[str, storages.BaseStorage]*) → None

Delete a `Study` object.

Parameters

- **study_name** – Study's name.
- **storage** – Database URL such as `sqlite:///example.db`. Please see also the documentation of `create_study()` for further details.

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: Union[str, storages.BaseStorage]*) → List[*StudySummary*]

Get all history of studies stored in a specified storage.

Parameters **storage** – 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.

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.

`__init__`()

Attributes

`NOT_SET`
`MINIMIZE`
`MAXIMIZE`

optuna.study.StudySummary

class `optuna.study.`**`StudySummary`**(*study_name: str, direction: optuna._study_direction.StudyDirection, best_trial: Optional[optuna.trial._frozen.FrozenTrial], user_attrs: Dict[str, Any], system_attrs: Dict[str, Any], n_trials: int, datetime_start: Optional[datetime.datetime], study_id: int*)

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`*.

`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.

__init__(*study_name: str*, *direction:* optuna._study_direction.StudyDirection, *best_trial: Optional[*optuna.trial._frozen.FrozenTrial*]*, *user_attrs: Dict[str, Any]*, *system_attrs: Dict[str, Any]*, *n_trials: int*, *datetime_start: Optional[datetime.datetime]*, *study_id: int*)

Methods

__init__(study_name, direction, best_trial, ...)	

6.3.14 optuna.trial

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:* Study, *trial_id: int*)

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.

__init__(*study:* Study, *trial_id: int*) → None

Methods

`__init__`(study, trial_id)	
`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: float*, *step: int*) → None

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** – A value returned from the objective function.
- **step** – 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.

set_system_attr(*key: str*, *value: Any*) → None

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** – A key string of the attribute.
- **value** – A value of the attribute. The value should be JSON serializable.

set_user_attr(*key: str*, *value: Any*) → None

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** – A key string of the attribute.
- **value** – A value of the attribute. The value should be JSON serializable.

should_prune() → bool

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.

suggest_categorical(*name: str*, *choices: Sequence[CategoricalChoiceType]*) → CategoricalChoiceType

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** – A parameter name.
- **choices** – Parameter value candidates.

See also:

CategoricalDistribution.

Returns A suggested value.

suggest_discrete_uniform(*name: str*, *low: float*, *high: float*, *q: float*) → float

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 $\mathsf{low}, \mathsf{low} + q, \mathsf{low} + 2q, \ldots, \mathsf{low} + kq \le \mathsf{high}$, 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** – A parameter name.
- **low** – Lower endpoint of the range of suggested values. `low` is included in the range.
- **high** – Upper endpoint of the range of suggested values. `high` is included in the range.
- **q** – A step of discretization.

Returns A suggested float value.

suggest_float(*name: str, low: float, high: float, *, step: Optional[float] = None, log: bool = False*) → float

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
```

(continues on next page)

(continued from previous page)

```
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** – A parameter name.
- **low** – Lower endpoint of the range of suggested values. `low` is included in the range.
- **high** – Upper endpoint of the range of suggested values. `high` is excluded from the range.
- **step** – 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** – 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.

suggest_int(*name: str*, *low: int*, *high: int*, *step: int = 1*, *log: bool = False*) → int

Suggest a value for the integer parameter.

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

Example

Suggest the number of trees in RandomForestClassifier.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)

def objective(trial):
    n_estimators = trial.suggest_int('n_estimators', 50, 400)
    clf = RandomForestClassifier(n_estimators=n_estimators, random_state=0)
    clf.fit(X_train, y_train)
    return clf.score(X_valid, y_valid)

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

Parameters

- **name** – A parameter name.
- **low** – Lower endpoint of the range of suggested values. `low` is included in the range.
- **high** – Upper endpoint of the range of suggested values. `high` is included in the range.
- **step** – 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** – 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.

suggest_loguniform(*name: str*, *low: float*, *high: float*) → float

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)
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** – A parameter name.
- **low** – Lower endpoint of the range of suggested values. `low` is included in the range.
- **high** – Upper endpoint of the range of suggested values. `high` is excluded from the range.

Returns A suggested float value.

suggest_uniform(*name: str*, *low: float*, *high: float*) → float

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)
    clf.fit(X_train, y_train)

    return clf.score(X_valid, y_valid)

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

Parameters

- **name** – A parameter name.
- **low** – Lower endpoint of the range of suggested values. `low` is included in the range.
- **high** – Upper endpoint of the range of suggested values. `high` is excluded from the range.

Returns A suggested float value.

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: Dict[str, Any], number: int = 0*)

A trial class which suggests a fixed value for each parameter.

This object has the same methods as *Trial*, and it suggests pre-defined parameter values. The parameter values can be determined at the construction of the *FixedTrial* object. In contrast to *Trial*, *FixedTrial* does not depend on *Study*, and it is useful for deploying optimization results.

Example

Evaluate an objective function with parameter values given by a user.

```python
import optuna

def objective(trial):
    x = trial.suggest_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`.

__init__(*params: Dict[str, Any], number: int = 0*) → None

Methods

__init__(params[, number])
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: int*, *state:* optuna.trial._state.TrialState, *value: Optional[float]*, *datetime_start: Optional[datetime.datetime]*, *datetime_complete: Optional[datetime.datetime]*, *params: Dict[str, Any]*, *distributions: Dict[str, optuna.distributions.BaseDistribution]*, *user_attrs: Dict[str, Any]*, *system_attrs: Dict[str, Any]*, *intermediate_values: Dict[int, float]*, *trial_id: int*)

Status and results of a *`Trial`*.

`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`*.

`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()`*.

`__init__`(*number: int*, *state:* optuna.trial._state.TrialState, *value: Optional[float]*, *datetime_start: Optional[datetime.datetime]*, *datetime_complete: Optional[datetime.datetime]*, *params: Dict[str, Any]*, *distributions: Dict[str, optuna.distributions.BaseDistribution]*, *user_attrs: Dict[str, Any]*, *system_attrs: Dict[str, Any]*, *intermediate_values: Dict[int, float]*, *trial_id: int*) → None

Methods

__init__(number, state, value, ...)	

Attributes

distributions	Dictionary that contains the distributions of *params*.
duration	Return the elapsed time taken to complete the trial.
last_step	

property distributions

Dictionary that contains the distributions of *params*.

property duration

Return the elapsed time taken to complete the trial.

Returns The duration.

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.

__init__()

Methods

is_finished()

Attributes

RUNNING
COMPLETE
PRUNED
FAIL
WAITING

optuna.trial.create_trial

optuna.trial.**create_trial**(*, *state: Optional*[optuna.trial._state.TrialState] *= None*, *value: Optional[float] = None*, *params: Optional[Dict[str, Any]] = None*, *distributions: Optional[Dict[str, optuna.distributions.BaseDistribution]] = None*, *user_attrs: Optional[Dict[str, Any]] = None*, *system_attrs: Optional[Dict[str, Any]] = None*, *intermediate_values: Optional[Dict[int, float]] = None*) → *optuna.trial._frozen.FrozenTrial*

Create a new `FrozenTrial`.

Example

```python
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()`.

Parameters

- **state** – Trial state.
- **value** – Trial objective value. Must be specified if `state` is *TrialState.COMPLETE*.
- **params** – Dictionary with suggested parameters of the trial.
- **distributions** – Dictionary with parameter distributions of the trial.
- **user_attrs** – Dictionary with user attributes.
- **system_attrs** – Dictionary with system attributes. Should not have to be used for most users.
- **intermediate_values** – Dictionary with intermediate objective values of the trial.

Returns Created trial.

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

Note: `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.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-dimentional parameter relationships in a study.
optuna.visualization.plot_param_importances	Plot hyperparameter importances.
optuna.visualization.plot_slice	Plot the parameter relationship as slice plot in a study.
optuna.visualization.is_available	Returns whether visualization is available or not.

optuna.visualization.plot_contour

`optuna.visualization.`**`plot_contour`**(*study:* optuna.study.Study, *params: Optional[List[str]] = None*) → go.Figure

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_uniform('x', -100, 100)
    y = trial.suggest_categorical('y', [-1, 0, 1])
    return x ** 2 + y

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

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

Parameters

- **study** – A *Study* object whose trials are plotted for their objective values.
- **params** – Parameter list to visualize. The default is all parameters.

Returns A plotly.graph_objs.Figure object.

optuna.visualization.plot_edf

optuna.visualization.**plot_edf**(*study: Union[*optuna.study.Study, *Sequence[*optuna.study.Study*]]*) → go.Figure

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)))
```

(continues on next page)

(continued from previous page)

```
    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()

# 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.plot_edf([study0, study1, study2])
```

Parameters **study** – A target *Study* object. You can pass multiple studies if you want to compare those EDFs.

Returns A `plotly.graph_objs.Figure` object.

optuna.visualization.plot_intermediate_values

`optuna.visualization.`**`plot_intermediate_values`**(*study: optuna.study.Study*) → go.Figure

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_loguniform("lr", 1e-5, 1e-1)
```

(continues on next page)

(continued from previous page)

```
    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

study = optuna.create_study()
study.optimize(objective, n_trials=16)

optuna.visualization.plot_intermediate_values(study)
```

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

Returns A `plotly.graph_objs.Figure` object.

optuna.visualization.plot_optimization_history

`optuna.visualization.`**`plot_optimization_history`**(*study:* optuna.study.Study) → go.Figure

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_uniform('x', -100, 100)
    y = trial.suggest_categorical('y', [-1, 0, 1])
    return x ** 2 + y

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

optuna.visualization.plot_optimization_history(study)
```

Parameters **study** – A *Study* object whose trials are plotted for their objective values.

Returns A `plotly.graph_objs.Figure` object.

optuna.visualization.plot_parallel_coordinate

`optuna.visualization.plot_parallel_coordinate(study: optuna.study.Study, params: Optional[List[str]] = None) → go.Figure`

Plot the high-dimentional 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-dimentional parameter relationships.

```
import optuna

def objective(trial):
    x = trial.suggest_uniform('x', -100, 100)
    y = trial.suggest_categorical('y', [-1, 0, 1])
    return x ** 2 + y

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

optuna.visualization.plot_parallel_coordinate(study, params=['x', 'y'])
```

Parameters

- **study** – A *Study* object whose trials are plotted for their objective values.
- **params** – Parameter list to visualize. The default is all parameters.

Returns A `plotly.graph_objs.Figure` object.

optuna.visualization.plot_param_importances

`optuna.visualization.plot_param_importances(study: optuna.study.Study, evaluator: optuna.importance._base.BaseImportanceEvaluator = None, params: Optional[List[str]] = None) → go.Figure`

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

study = optuna.create_study(sampler=optuna.samplers.RandomSampler())
```

(continues on next page)

(continued from previous page)

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

optuna.visualization.plot_param_importances(study)
```

See also:

This function visualizes the results of *optuna.importance.get_param_importances()*.

Parameters

- **study** – An optimized study.
- **evaluator** – An importance evaluator object that specifies which algorithm to base the importance assessment on. Defaults to *FanovaImportanceEvaluator*.
- **params** – A list of names of parameters to assess. If None, all parameters that are present in all of the completed trials are assessed.

Returns A plotly.graph_objs.Figure object.

optuna.visualization.plot_slice

optuna.visualization.**plot_slice**(*study: optuna.study.Study*, *params: Optional[List[str]] = None*) → go.Figure

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_uniform('x', -100, 100)
    y = trial.suggest_categorical('y', [-1, 0, 1])
    return x ** 2 + y

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

optuna.visualization.plot_slice(study, params=['x', 'y'])
```

Parameters

- **study** – A *Study* object whose trials are plotted for their objective values.
- **params** – Parameter list to visualize. The default is all parameters.

Returns A plotly.graph_objs.Figure object.

optuna.visualization.is_available

`optuna.visualization.is_available()` → bool

Returns whether visualization 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 is available, `False` otherwise.

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):
```

(continues on next page)

(continued from previous page)

```
        # 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_uniform('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_uniform('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.

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 2018-12-05 11:41:42,324] Finished a trial resulted in 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_loguniform('svc_c', 1e-10, 1e10)
    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)
```

(continues on next page)

(continued from previous page)

```
    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_uniform()* 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_uniform('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_uniform('x', -1.0, 1.0)
    y = trial.suggest_int('y', -5, 5)
    return x + y

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

# `gc_after_trial=True` is more or less identical to the following.
study.optimize(objective, n_trials=10, callbacks=[lambda study, trial: gc.collect()])
```

There is a performance trade-off for running the garbage collector, which could be non-negligible depending on how fast your objective function otherwise is. Therefore, `gc_after_trial` is `False` by default. Note that the above examples are similar to running the garbage collector inside the objective function, except for the fact that `gc.collect()` is called even when errors, including *TrialPruned* are raised.

Note: *ChainerMNStudy* does currently not provide `gc_after_trial` nor callbacks for *optimize()*. When using this class, you will have to call the garbage collector inside the objective function.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX