Optuna Documentation

Release 0.18.1

Optuna Contributors.

Nov 14, 2019

Contents:

CHAPTER 1

Installation

Optuna supports Python 2.7 and 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/pfnet/optuna.git
```

CHAPTER 2

Tutorial

2.1 First Optimization

2.1.1 Quadratic Function Example

Let us try very simple optimization in IPython shell.

```
In [1]: import optuna
```

Here, we use a very simple quadratic function as an example of objective function.

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

Our goal is to find out `x` that minimizes the output of `objective` function, which we refer to as "optimization." During the optimization, Optuna repeatedly invokes 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 (e.g., *`suggest_uniform()`*) are called inside the objective function to obtain parameters for a trial.

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

```
In [3]: study = optuna.create_study()
In [4]: study.optimize(objective, n_trials=100)
[I 2018-05-09 10:03:22,469] Finished trial#0 resulted in value: 52.9345515866657.␣
↪Current best value is 52.9345515866657 with parameters: {'x': -5.275613485244093}.
[I 2018-05-09 10:03:22,474] Finished trial#1 resulted in value: 32.82718929591965.␣
↪Current best value is 32.82718929591965 with parameters: {'x': -3.7295016620924066}.
[I 2018-05-09 10:03:22,475] Finished trial#2 resulted in value: 46.89428737068025.␣
↪Current best value is 32.82718929591965 with parameters: {'x': -3.7295016620924066}.
```

(continues on next page)

(continued from previous page)

```
[I 2018-05-09 10:03:22,476] Finished trial#3 resulted in value: 100.99613064563654.␣
↪Current best value is 32.82718929591965 with parameters: {'x': -3.7295016620924066}.
[I 2018-05-09 10:03:22,477] Finished trial#4 resulted in value: 110.56391159932272.␣
↪Current best value is 32.82718929591965 with parameters: {'x': -3.7295016620924066}.
[I 2018-05-09 10:03:22,478] Finished trial#5 resulted in value: 42.486606942847395.␣
↪Current best value is 32.82718929591965 with parameters: {'x': -3.7295016620924066}.
[I 2018-05-09 10:03:22,479] Finished trial#6 resulted in value: 1.130813338091735.␣
↪Current best value is 1.130813338091735 with parameters: {'x': 3.063397074517198}.
...
[I 2018-05-09 10:03:23,431] Finished trial#99 resulted in value: 8.760381111220335.␣
↪Current best value is 0.0026232243068543526 with parameters: {'x': 1.
↪9487825780924659}.
In [5]: study.best_params
Out[5]: {'x': 1.9487825780924659}
```

We can see that Optuna found the best `x` value `1.9487825780924659`, which is close to the optimal value of `2`.

Note: In practice, it is expected that training of machine learning algorithms is invoked in objective functions, and metrics such as loss or error are reported.

2.1.2 Study Object

Let us clarify the terminology in Optuna as follows.

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

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

```
In [5]: study.best_params
Out[5]: {'x': 1.9926578647650126}

In [6]: study.best_value
Out[6]: 5.390694980884334e-05

In [7]: study.best_trial
Out[7]: FrozenTrial(number=26, state=<TrialState.COMPLETE: 1>, params={'x': 1.
↪9926578647650126}, user_attrs={}, system_attrs={'_number': 26}, value=5.
↪390694980884334e-05, intermediate_values={}, datetime_start=datetime.datetime(2018,␣
↪5, 9, 10, 23, 0, 87060), datetime_complete=datetime.datetime(2018, 5, 9, 10, 23, 0,␣
↪91010), trial_id=26)

In [8]: study.trials  # all trials
Out[8]:
[FrozenTrial(number=0, state=<TrialState.COMPLETE: 1>, params={'x': -4.
↪219801301030433}, user_attrs={}, system_attrs={'_number': 0}, value=38.
↪685928224299865, intermediate_values={}, datetime_start=datetime.datetime(2018, 5,␣
↪9, 10, 22, 59, 983824), datetime_complete=datetime.datetime(2018, 5, 9, 10, 22, 59,␣
↪984053), trial_id=0),
 ...
 user_attrs={}, system_attrs={'_number': 99}, value=8.2881000286123179, intermediate_
↪values={}, datetime_start=datetime.datetime(2018, 5, 9, 10, 23, 0, 886434),␣
↪datetime_complete=datetime.datetime(2018, 5, 9, 10, 23, 0, 891347), trial_id=99)]
```

(continues on next page)

(continued from previous page)

```
In [9]: len(study.trials)
Out[9]: 100
```

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

```
In [10]: study.optimize(objective, n_trials=100)
...

In [11]: len(study.trials)
Out[11]: 200
```

2.2 Advanced Configurations

2.2.1 Defining Parameter Spaces

Currently, we support 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)

    ...
```

2.2.2 Branches and Loops

You can use branches or loops depending on 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)
    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. We recommend not to add unimportant parameters.

2.2.3 Arguments for *Study.optimize*

Method *optimize()* (and `optuna study optimize` CLI command as well) has several useful options such as `timeout`. Please refer to its docstring.

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 feature is useful for certain use cases, e.g., when it is hard to estimate computational costs required to optimize your objective function.

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.

2.3.1 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)
```

2.3.2 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)
```

2.3.3 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()
```

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

```
number                state       value             datetime_start          datetime_
↪complete    params system_attrs

↪                 x      _number
     0  TrialState.COMPLETE   25.301959 2019-03-14 10:57:27.716141 2019-03-14␣
↪10:57:27.746354 -3.030105            0
     1  TrialState.COMPLETE    1.406223 2019-03-14 10:57:27.774461 2019-03-14␣
↪10:57:27.835520  0.814157            1
     2  TrialState.COMPLETE   44.010366 2019-03-14 10:57:27.871365 2019-03-14␣
↪10:57:27.926247 -4.634031            2
     3  TrialState.COMPLETE   55.872181 2019-03-14 10:59:00.845565 2019-03-14␣
↪10:59:00.899305  9.474770            3
     4  TrialState.COMPLETE  113.039223 2019-03-14 10:59:00.921534 2019-03-14␣
↪10:59:00.947233 -8.631991            4
     5  TrialState.COMPLETE   57.319570 2019-03-14 10:59:00.985909 2019-03-14␣
↪10:59:01.028819  9.570969            5
```

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.
```

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

2.5 Command-Line Interface

Command	Description
create-study	Create a new study.
dashboard	Launch web dashboard (beta).
storage upgrade	Upgrade the schema of a storage.
studies	Show a list of studies.
study optimize	Start optimization of a study.
study set-user-attr	Set a user attribute to a study.

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

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

```
import optuna


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

```
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 $STUDY_NAME
[I 2018-05-09 10:40:25,196] Finished a trial resulted in value: 54.353767789264026.␣
↪Current best value is 54.353767789264026 with parameters: {'x': -5.372500782588228}.
[I 2018-05-09 10:40:25,197] Finished a trial resulted in value: 15.784266965526376.␣
↪Current best value is 15.784266965526376 with parameters: {'x': 5.972941852774387}.
...
[I 2018-05-09 10:40:26,204] Finished a trial resulted in value: 14.704254135013741.␣
↪Current best value is 2.280758099793617e-06 with parameters: {'x': 1.
↪9984897821018828}.
```

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

2.6 User Attributes

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

2.6.1 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.

2.6.2 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()

    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*.

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.

2.7.1 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, test_x, train_y, test_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(test_x, test_y)
        trial.report(intermediate_value, step)

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

    return 1.0 - clf.score(test_x, test_y)

# 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 2018-11-21 17:27:57,836] Finished trial#0 resulted in value: 0.052631578947368474.␣
↪Current best value is 0.052631578947368474 with parameters: {'alpha': 0.
↪011428158279113485}.
[I 2018-11-21 17:27:57,963] Finished trial#1 resulted in value: 0.02631578947368418.␣
↪Current best value is 0.02631578947368418 with parameters: {'alpha': 0.
↪01862693201743629}.
[I 2018-11-21 17:27:58,164] Finished trial#2 resulted in value: 0.21052631578947367.␣
↪Current best value is 0.02631578947368418 with parameters: {'alpha': 0.
↪01862693201743629}.
```

(continues on next page)

(continued from previous page)

```
[I 2018-11-21 17:27:58,333] Finished trial#3 resulted in value: 0.02631578947368418.␣
↪Current best value is 0.02631578947368418 with parameters: {'alpha': 0.
↪01862693201743629}.
[I 2018-11-21 17:27:58,617] Finished trial#4 resulted in value: 0.23684210526315785.␣
↪Current best value is 0.02631578947368418 with parameters: {'alpha': 0.
↪01862693201743629}.
[I 2018-11-21 17:27:58,642] Setting status of trial#5 as TrialState.PRUNED.
[I 2018-11-21 17:27:58,666] Setting status of trial#6 as TrialState.PRUNED.
[I 2018-11-21 17:27:58,675] Setting status of trial#7 as TrialState.PRUNED.
[I 2018-11-21 17:27:59,183] Finished trial#8 resulted in value: 0.39473684210526316.␣
↪Current best value is 0.02631578947368418 with parameters: {'alpha': 0.
↪01862693201743629}.
[I 2018-11-21 17:27:59,202] Setting status of trial#9 as TrialState.PRUNED.
...
```

We can see `Setting status of trial#{} as TrialState.PRUNED` in the log messages. This means several trials are stopped before they finish all iterations.

2.7.2 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=[(dtest, 'validation')], callbacks=[pruning_
↪callback])
```

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.

2.8.1 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.

2.8.2 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:
```

(continues on next page)

(continued from previous page)

```
            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

    #
    # 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.

CHAPTER 3

API Reference

3.1 Distributions

class optuna.distributions.**UniformDistribution**(*low*, *high*)
A uniform distribution in the linear domain.

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

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

high
Upper endpoint of the range of the distribution. `high` is excluded from the range.

single()
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`.

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

single()
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`.

class optuna.distributions.**DiscreteUniformDistribution**(*low*, *high*, *q*)
A discretized uniform distribution in the linear domain.

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

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

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

q
A discretization step.

single()
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`.

class optuna.distributions.**IntUniformDistribution**(*low*, *high*)
A uniform distribution on integers.

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.

single()
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`.

class optuna.distributions.**CategoricalDistribution**(*choices*)
A categorical distribution.

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

choices
Candidates of parameter values.

single()
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`.

optuna.distributions.**distribution_to_json**(*dist*)
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**(*json_str*)
Deserialize a distribution in JSON format.

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

Returns A deserialized distribution.

optuna.distributions.**check_distribution_compatibility**(*dist_old*, *dist_new*)
A function to check compatibility of two distributions.

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

Parameters

- **dist_old** – 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.

3.2 Exceptions

class optuna.exceptions.**OptunaError**
Base class for Optuna specific errors.

class 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.

Example

```
>>> def objective(trial):
>>>     ...
>>>     for step in range(n_train_iter):
>>>         ...
>>>         if trial.should_prune():
>>>             raise TrailPruned()
```

class optuna.exceptions.**CLIUsageError**
Exception for CLI.

CLI raises this exception when it receives invalid configuration.

class optuna.exceptions.**StorageInternalError**
Exception for storage operation.

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

class optuna.exceptions.**DuplicatedStudyError**
Exception for a duplicated study name.

This error is raised when a specified study name already exists in the storage.

3.3 Integration

class optuna.integration.**ChainerPruningExtension**(*trial*, *observation_key*, *pruner_trigger*)
Chainer extension to prune unpromising trials.

Example

Add a pruning extension which observes validation losses to Chainer Trainer.

```
trainer.extend(
    ChainerPruningExtension(trial, 'validation/main/loss', (1, 'epoch')))
```

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')`.

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

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

See also:

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

Example

Optimize an objective function that trains neural network written with ChainerMN.

```
comm = chainermn.create_communicator('naive')
study = optuna.load_study(study_name, storage_url)
chainermn_study = optuna.integration.ChainerMNStudy(study, comm)
chainermn_study.optimize(objective, n_trials=25)
```

Parameters

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

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

Optimize an objective function.

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

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

A Sampler using cma library as the backend.

Example

Optimize a simple quadratic function by using *CmaEsSampler*.

```
def objective(trial):
    x = trial.suggest_uniform('x', -1, 1)
    y = trial.suggest_int('y', -1, 1)
    return x**2 + y

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

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.

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.CMAEvotionStrategy 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.CMAEvotionStrategy 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.CMAEvotionStrategy for further details of `cma_stds`.
- **seed** – A random seed for CMA-ES.
- **cma_opts** – Options passed to the constructor of cma.CMAEvotionStrategy class.

  Note that `BoundaryHandler`, `bounds`, `CMA_stds` and `seed` arguments in `cma_opts` will be ignored because it is added by *CmaEsSampler* 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 *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.

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

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.

Example

Add a pruning callback which monitors validation loss directly to `Learner`.

```
# If registering this callback in construction
from functools import partial

learn = Learner(
    data, model,
    callback_fns=[partial(FastAIPruningCallback, trial=trial, monitor='valid_loss
↪')])
```

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.

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

PyTorch Ignite handler to prune unpromising trials.

Example

Add a pruning handler which observes validation accuracy.

```
evaluator = create_supervised_evaluator(model,
                                        metrics={'accuracy': Accuracy()},
                                        device=device)
handler = PyTorchIgnitePruningHandler(trial, 'accuracy', trainer)
evaluator.add_event_handler(Events.COMPLETED, handler)

@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(engine):
    evaluator.run(val_loader)
```

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.

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

Example

Add a pruning callback which observes validation losses.

```
model.fit(X, y, callbacks=KerasPruningCallback(trial, 'val_loss'))
```

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_acc`. Please refer to keras.Callback reference for further details.

class optuna.integration.**LightGBMPruningCallback**(*trial*, *metric*, *valid_name='valid_0'*)
Callback for LightGBM to prune unpromising trials.

Example

Add a pruning callback which observes validation scores to training of a LightGBM model.

```
param = {'objective': 'binary', 'metric': 'binary_error'}
pruning_callback = LightGBMPruningCallback(trial, 'binary_error')
gbm = lgb.train(param, dtrain, valid_sets=[dtest], callbacks=[pruning_callback])
```

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.

class optuna.integration.**MXNetPruningCallback**(*trial*, *eval_metric*)
MXNet callback to prune unpromising trials.

Example

Add a pruning callback which observes validation accuracy.

```
model.fit(train_data=X, eval_data=Y,
          eval_end_callback=MXNetPruningCallback(trial, eval_metric='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.

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

PyTorch Lightning callback to prune unpromising trials.

Example

Add a pruning callback which observes validation accuracy.

```
trainer.pytorch_lightning.Trainer(
    early_stop_callback=PyTorchLightningPruningCallback(trial, monitor='avg_val_
↪acc'))
```

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.

class optuna.integration.**SkoptSampler**(*independent_sampler=None*, *warn_independent_sampling=True*, *skopt_kwargs=None*)

Sampler using Scikit-Optimize as the backend.

Example

Optimize a simple quadratic function by using *SkoptSampler*.

```
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=100)
```

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.

class optuna.integration.**TensorFlowPruningHook**(*trial*, *estimator*, *metric*, *run_every_steps*, *is_higher_better=None*)

TensorFlow SessionRunHook to prune unpromising trials.

Example

Add a pruning SessionRunHook for a TensorFlow's Estimator.

```
pruning_hook = TensorFlowPruningHook(
    trial=trial,
    estimator=clf,
    metric="accuracy",
    is_higher_better=True,
    run_every_steps=10,
)
hooks = [pruning_hook]
tf.estimator.train_and_evaluate(
    clf,
    tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=500, hooks=hooks),
    eval_spec
)
```

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.
- **is_higher_better** – Please do not use this argument because this class refers to *StudyDirection* to check whether the current study is `minimize` or `maximize`.

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

tf.keras callback to prune unpromising trials.

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

Example

Add a pruning callback which observes validation losses.

```
model.fit(x, y, callbacks=[TFKerasPruningCallback(trial, 'val_loss')])
```

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`.

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

Callback for XGBoost to prune unpromising trials.

Example

Add a pruning callback which observes validation errors to training of an XGBoost model.

```
pruning_callback = XGBoostPruningCallback(trial, 'validation-error')
bst = xgb.train(param, dtrain, evals=[(dtest, 'validation')],
                callbacks=[pruning_callback])
```

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`. Please refer to `eval_metric` in XGBoost reference for further details.

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

Hyperparameter search with cross-validation.

Warning: This feature is experimental. The interface may be changed in the future.

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, test) 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 test 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) # doctest: +ELLIPSIS
OptunaSearchCV(...)
>>> y_pred = optuna_search.predict(X)
```

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.

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

fit (*X*, *y=None*, *groups=None*, ***fit_params*)
Run fit with all sets of parameters.

Parameters

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

Returns Return self.

Return type self

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`.

n_trials_
Actual number of trials.

predict
Call `predict` on the best estimator.

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

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`.

predict_proba
Call `predict_proba` on the best estimator.

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

score(*X*, *y=None*)
Return the score on the given data.

Parameters

- **X** – Data.
- **y** – Target variable.

Returns Scaler score.

Return type score

score_samples
Call `score_samples` on the best estimator.

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

set_user_attr
Call `set_user_attr` on the *Study*.

transform
Call `transform` on the best estimator.

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

trials_
All trials in the *Study*.

trials_dataframe
Call `trials_dataframe` on the *Study*.

user_attrs_
User attributes in the *Study*.

3.4 Logging

optuna.logging.**get_verbosity**()
Return the current level for the Optuna's root logger.

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

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**(*verbosity*)
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**()
Disable the default handler of the Optuna's root logger.

Example

Stop and then resume logging to standard output.

```
>> study = optuna.create_study()
>> optuna.logging.disable_default_handler()
>> study.optimize(objective, n_trials=10)
>> len(study.trials)
10
>> optuna.logging.enable_default_handler()
>> study.optimize(objective, n_trials=10)
[I 2018-11-07 16:11:28,285] Finished a trial resulted in value: 3787.44371584515.␣
↪...
```

optuna.logging.**enable_default_handler**()
Enable the default handler of the Optuna's root logger.

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

optuna.logging.**disable_propagation**()
Disable propagation of the library log outputs.

Note that log propagation is disabled by default.

optuna.logging.**enable_propagation**()
Enable propagation of the library log outputs.

Please disable the Optuna's default handler to prevent double logging if the root logger has been configured.

Example

Propagate all log output to the root logger in order to save them to the file.

```
>> logging.getLogger().setLevel(logging.INFO)  # Setup the root logger.
>> logging.getLogger().addHandler(logging.FileHandler('foo.log'))

>> optuna.logging.enable_propagation()  # Propagate logs to the root logger.
>> optuna.logging.disable_default_handler()  # Stop showing logs in stderr.

>> study = optuna.create_study()
>> logging.getLogger().info("Start optimization.")
>> study.optimize(objective, n_trials=10)
>> open('foo.log').readlines()
["Start optimization.", "Finished trial#0 resulted in value: ...
```

3.5 Pruners

class optuna.pruners.**MedianPruner**(*n_startup_trials=5*, *n_warmup_steps=0*, *interval_steps=1*)
Pruner using the median stopping rule.

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

Example

We minimize an objective function with the median stopping rule.

```
>>> from optuna import create_study
>>> from optuna.pruners import MedianPruner
>>>
>>> def objective(trial):
>>>     ...
>>>
>>> study = create_study(pruner=MedianPruner())
>>> study.optimize(objective)
```

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 reaches 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.

class optuna.pruners.**NopPruner**
Pruner which never prunes trials.

Example

```
>>> from optuna import create_study
>>> from optuna.pruners import NopPruner
>>>
>>> def objective(trial):
>>>     ...
>>>
>>> study = create_study(pruner=NopPruner())
>>> study.optimize(objective)
```

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

Pruner to keep the specified percentile of the trials.

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

Example

```
>>> from optuna import create_study
>>> from optuna.pruners import PercentilePruner
>>>
>>> def objective(trial):
>>>     ...
>>>
>>> study = create_study(pruner=PercentilePruner(25.0))
>>> study.optimize(objective)
```

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 reaches 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.

class optuna.pruners.**SuccessiveHalvingPruner**(*min_resource=1*, *reduction_factor=4*, *min_early_stopping_rate=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).

Example

We minimize an objective function with `SuccessiveHalvingPruner`.

```
>>> from optuna import create_study
>>> from optuna.pruners import SuccessiveHalvingPruner
>>>
>>> def objective(trial):
>>>     ...
>>>
>>> study = create_study(pruner=SuccessiveHalvingPruner())
>>> study.optimize(objective)
```

Parameters

- **min_resource** – A parameter for specifying the minimum resource allocated to a trial (in the paper this parameter is referred to as r).

  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.

- **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).

3.6 Samplers

class optuna.samplers.**BaseSampler**

Base class for samplers.

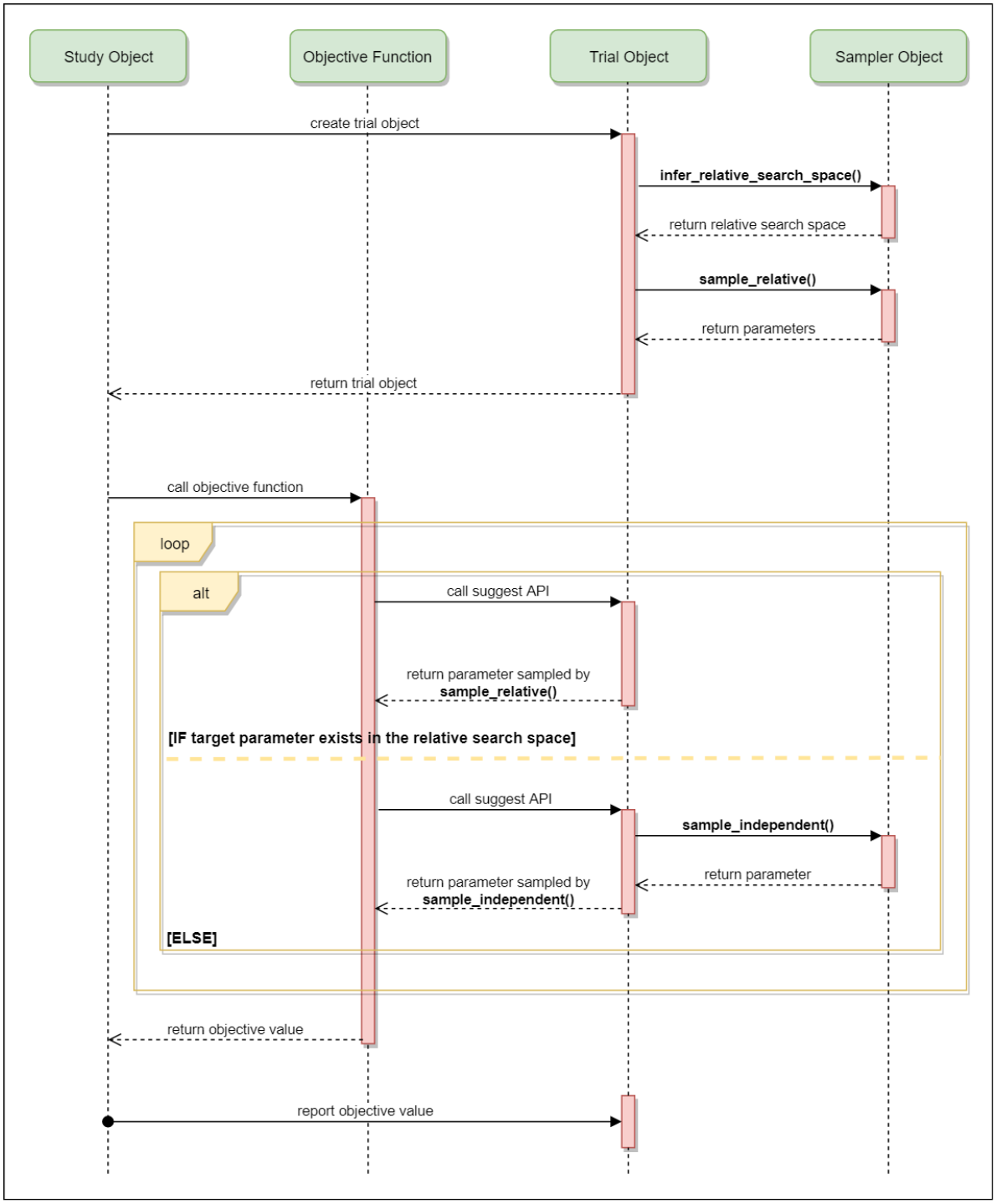
Optuna combines two types of sampling strategies, which are called *relative sampling* and *independent sampling*.

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.



infer_relative_search_space(*study*, *trial*)

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

This method is called right before *sample_relative()* method, and the search space returned by

this method is 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.

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*, *trial*, *param_name*, *param_distribution*)
Sample a parameter for a given distribution.

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

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*, *trial*, *search_space*)
Sample parameters in a given search space.

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

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.

class optuna.samplers.**RandomSampler**(*seed=None*)
Sampler using random sampling.

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

Example

```
>>> study = optuna.create_study(sampler=RandomSampler())
>>> study.optimize(objective, direction='minimize')
```

Args: seed: Seed for random number generator.

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

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=100)
```

static hyperopt_parameters()

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

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

Example

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

```
import optuna
from optuna.samplers import TPESampler

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

sampler = TPESampler(**TPESampler.hyperopt_parameters())
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=100)
```

Returns A dictionary containing the default parameters of hyperopt.

optuna.samplers.**intersection_search_space**(*study*)
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).

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

optuna.samplers.**product_search_space**(*study*)
Return the product search space of the BaseStudy.

Deprecated since version 0.14.0: Please use *intersection_search_space()* instead.

3.7 Storages

class optuna.storages.**RDBStorage**(*url*, *engine_kwargs=None*, *enable_cache=True*, *skip_compatibility_check=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

We create an *RDBStorage* instance with customized pool_size and max_overflow settings.

```
>>> import optuna
>>>
>>> def objective(trial):
>>>     ...
>>>
>>> storage = optuna.storages.RDBStorage(
>>>     url='postgresql://foo@localhost/db',
>>>     engine_kwargs={
>>>         'pool_size': 20,
>>>         'max_overflow': 0
>>>     }
>>> )
>>>
>>> study = optuna.create_study(storage=storage)
>>> study.optimize(objective)
```

Parameters

- **url** – URL of the storage.
- **engine_kwargs** – A dictionary of keyword arguments that is passed to sqlalchemy.engine.create_engine function.
- **enable_cache** – Flag to control whether to enable storage layer caching. If this flag is set to True (the default), the finished trials are cached on memory and never re-fetched from the storage. Otherwise, the trials are fetched from the storage whenever they are needed.

3.8 Structs

class optuna.structs.**TrialState**
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.

class optuna.structs.**StudyDirection**
Direction of a *Study*.

NOT_SET
Direction has not been set.

MINIMIZE
Study minimizes the objective function.

MAXIMIZE
Study maximizes the objective function.

class optuna.structs.**FrozenTrial**(*number*, *state*, *value*, *datetime_start*, *datetime_complete*, *params*, *distributions*, *user_attrs*, *system_attrs*, *intermediate_values*, *trial_id*)
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.

distributions
Dictionary that contains the distributions of *params*.

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

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

distributions
Return the distributions for this trial.

Returns The distributions.

class optuna.structs.**StudySummary**
Basic attributes and aggregated results of a *Study*.

See also *optuna.study.get_all_study_summaries()*.

study_id
Identifier of the *Study*.

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.

3.9 Study

class optuna.study.**Study**(*study_name*, *storage*, *sampler=None*, *pruner=None*)
A study corresponds to an optimization task, i.e., a set of trials.

This object provides interfaces to run a new *Trial*, access trials' history, set/get user-defined attributes of the study itself.

Note that the direct use of this constructor is not recommended. To create and load a study, please refer to the documentation of *create_study()* and *load_study()* respectively.

best_params
Return parameters of the best trial in the study.

Returns A dictionary containing parameters of the best trial.

best_trial
Return the best trial in the study.

Returns A *FrozenTrial* object of the best trial.

best_value
Return the best objective value in the study.

Returns A float representing the best objective value.

direction
Return the direction of the study.

Returns A *StudyDirection* object.

optimize(*func*, *n_trials=None*, *timeout=None*, *n_jobs=1*, *catch=()*, *callbacks=None*, *gc_after_trial=True*)
Optimize an objective function.

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 counts.
- **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.
- **gc_after_trial** – Flag to execute garbage collection at the end of each trial. By default, garbage collection is enabled, just in case. You can turn it off with this argument if memory is safely managed in your objective function.

set_user_attr(*key*, *value*)
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.

trials
Return all trials in the study.

The returned trials are ordered by trial number.

Returns A list of *FrozenTrial* objects.

trials_dataframe(*include_internal_fields=False*)
Export trials as a pandas DataFrame.

The DataFrame provides various features to analyze studies. It is also useful to draw a histogram of objective values and to export trials as a CSV file. Note that DataFrames returned by *trials_dataframe()* employ MultiIndex, and columns have a hierarchical structure. Please refer to the example below to access DataFrame elements. If there are no trials, an empty DataFrame is returned.

Example

Get an objective value and a value of parameter `x` in the first row.

```
>>> df = study.trials_dataframe()
>>> df
>>> df.value[0]
0.0
>>> df.params.x[0]
1.0
```

Parameters **include_internal_fields** – By default, internal fields of *FrozenTrial* are excluded from a DataFrame of trials. If this argument is True, they will be included in the DataFrame.

Returns A pandas DataFrame of trials in the *Study*.

user_attrs

Return user attributes.

Returns A dictionary containing all user attributes.

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

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.

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

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*.

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

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

3.10 Trial

class optuna.trial.**Trial**(*study*, *trial_id*)
A trial is a process of evaluating an objective function.

This object is passed to an objective function and provides interfaces to get parameter suggestion, manage the trial's state, and set/get user-defined attributes of the trial.

Note that the direct use of this constructor is not recommended. This object is seamlessly instantiated and passed to the objective function behind the *optuna.study.Study.optimize()* method; hence library users do not care about instantiation of this object.

Parameters

- **study** – A *Study* object.
- **trial_id** – A trial ID that is automatically generated.

datetime_start
Return start datetime.

Returns Datetime where the *Trial* started.

distributions
Return distributions of parameters to be optimized.

Returns A dictionary containing all distributions.

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

Returns A trial number.

params
Return parameters to be optimized.

Returns A dictionary containing all parameters.

report (*value*, *step=None*)
Report an objective function value.

If step is set to `None`, the value is stored as a final value of the trial. Otherwise, it is saved as an intermediate value.

Note that 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

```
>>> def objective(trial):
>>>     ...
>>>     clf = sklearn.linear_model.SGDClassifier()
>>>     for step in range(100):
>>>         clf.partial_fit(x_train , y_train , classes)
>>>         intermediate_value = clf.score(x_val , y_val)
>>>         trial.report(intermediate_value , step=step)
>>>         if trial.should_prune():
>>>             raise TrialPruned()
>>>     ...
```

Parameters

- **value** – A value returned from the objective function.
- **step** – Step of the trial (e.g., Epoch of neural network training).

set_user_attr (*key*, *value*)
Set user attributes to the trial.

The user attributes in the trial can be access via *optuna.trial.Trial.user_attrs()*.

Example

Save fixed hyperparameters of neural network training:

```
>>> def objective(trial):
>>>     ...
>>>     trial.set_user_attr('BATCHSIZE', 128)
>>>
>>> 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(*step=None*)
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()*.

Parameters **step** – Deprecated since 0.12.0: Step of the trial (e.g., epoch of neural network training). Deprecated in favor of always considering the most recent step.

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*, *choices*)
Suggest a value for the categorical parameter.

The value is sampled from `choices`.

Example

Suggest a kernel function of SVC.

```
>>> def objective(trial):
>>>     ...
>>>     kernel = trial.suggest_categorical('kernel', ['linear', 'poly', 'rbf
↪'])
>>>     clf = sklearn.svm.SVC(kernel=kernel)
>>>     ...
```

Parameters

- **name** – A parameter name.
- **choices** – Candidates of parameter values.

Returns A suggested value.

suggest_discrete_uniform(*name*, *low*, *high*, *q*)
Suggest a value for the discrete parameter.

The value is sampled from the range $[\mathsf{low}, \mathsf{high}]$, and the step of discretization is q. More specifically, this method returns one of the values in the sequence $\mathsf{low}, \mathsf{low} + q, \mathsf{low} + 2q, \ldots, \mathsf{low} + kq \leq \mathsf{high}$, where k denotes an integer. Note that $high$ may be changed due to round-off errors if q is not an integer. Please check warning messages to find the changed values.

Example

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

```
>>> def objective(trial):
>>>     ...
>>>     subsample = trial.suggest_discrete_uniform('subsample', 0.1, 1.0, 0.1)
>>>     clf = sklearn.ensemble.GradientBoostingClassifier(subsample=subsample)
>>>     ...
```

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_int(*name*, *low*, *high*)

Suggest a value for the integer parameter.

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

Example

Suggest the number of trees in RandomForestClassifier.

```
>>> def objective(trial):
>>>     ...
>>>     n_estimators = trial.suggest_int('n_estimators', 50, 400)
>>>     clf = sklearn.ensemble.RandomForestClassifier(n_estimators=n_
↪estimators)
>>>     ...
```

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.

Returns A suggested integer value.

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

Suggest a value for the continuous parameter.

The value is sampled from the range $[\mathsf{low}, \mathsf{high})$ in the log domain. When $\mathsf{low} = \mathsf{high}$, the value of low will be returned.

Example

Suggest penalty parameter `C` of SVC.

```
>>> def objective(trial):
>>>     ...
>>>     c = trial.suggest_loguniform('c', 1e-5, 1e2)
>>>     clf = sklearn.svm.SVC(C=c)
>>>     ...
```

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*, *low*, *high*)

Suggest a value for the continuous parameter.

The value is sampled from the range $[\mathsf{low}, \mathsf{high})$ in the linear domain. When $\mathsf{low} = \mathsf{high}$, the value of low will be returned.

Example

Suggest a dropout rate for neural network training.

```
>>> def objective(trial):
>>>     ...
>>>     dropout_rate = trial.suggest_uniform('dropout_rate', 0, 1.0)
>>>     ...
```

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.

user_attrs

Return user attributes.

Returns A dictionary containing all user attributes.

class optuna.trial.**FixedTrial**(*params*)

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:

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

Note: Please refer to *Trial* for details of methods and properties.

Parameters **params** – A dictionary containing all parameters.

3.11 Visualization

optuna.visualization.**plot_contour**(*study*, *params=None*)

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):
    ...

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

optuna.visualization.plot_contour(study, params=['param_a', 'param_b'])
```

Parameters

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

optuna.visualization.**plot_intermediate_values**(*study*)

Plot intermediate values of all trials in a study.

Example

The following code snippet shows how to plot intermediate values.

```
import optuna

def objective(trial):
    # Intermediate values are supposed to be reported inside the objective␣
↪function.
    ...

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

optuna.visualization.plot_intermediate_values(study)
```

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

optuna.visualization.**plot_optimization_history**(*study*)

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):
    ...

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

optuna.visualization.plot_optimization_history(study)
```

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

optuna.visualization.**plot_parallel_coordinate**(*study*, *params=None*)

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):
    ...

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

optuna.visualization.plot_parallel_coordinate(study, params=['param_a', 'param_b
↪'])
```

Parameters

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

optuna.visualization.**plot_slice**(*study*, *params=None*)

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):
    ...

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

optuna.visualization.plot_slice(study, params=['param_a', 'param_b'])
```

Parameters

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

optuna.visualization.**is_available**()

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.

CHAPTER 4

FAQ

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.

4.2 How to define objective functions that have own arguments?

There are two ways to realize it.

First, callable classes can be used for that purpose as follows:

```
import optuna

class Objective(object):
    def __init__(self, min_x, max_x):
        # Hold this implementation specific arguments as the fields of the class.
        self.min_x = min_x
        self.max_x = max_x

    def __call__(self, trial):
        # Calculate an objective value by using the extra arguments.
        x = trial.suggest_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.

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.

4.4 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.

4.5 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)
    return 1.0 - accuracy_score(y_test, clf.predict(X_test))


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_test, best_clf.predict(X_test)))
```

4.6 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.

4.7 How does Optuna handle NaNs and exceptions reported by the objective function?

Optuna treats such trials as failures (i.e., *FAIL*) and continues the study. The Optuna's system process will not be crashed by any objective values or exceptions raised in objective functions.

You can find the failed trials in log messages. Errors raised in objective functions are shown as follows:

```
[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.')
```

And trials which returned `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.
```

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	

4.8 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 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 bar --storage sqlite:///example.db
```

Please refer to CUDA C Programming Guide for further details.

4.9 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}))
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

Q

R

S

T

U

V

X