Optuna Documentation

Release 0.7.0

Preferred Networks, Inc.

Feb 21, 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 a trial 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 a trial 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 a trial 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 a trial 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 a trial 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 a trial 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 a trial 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 a trial 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(trial_id=26, state=<TrialState.COMPLETE: 1>, params={'x': 1.
↪9926578647650126}, user_attrs={}, system_attrs={}, value=5.390694980884334e-05,␣
↪intermediate_values={}, params_in_internal_repr={'x': 1.9926578647650126}, datetime_
↪start=datetime.datetime(2018, 5, 9, 10, 23, 0, 87060), datetime_complete=datetime.
↪datetime(2018, 5, 9, 10, 23, 0, 91010))

In [8]: study.trials  # all trials
Out[8]:
[FrozenTrial(trial_id=0, state=<TrialState.COMPLETE: 1>, params={'x': -4.
↪219801301030433}, user_attrs={}, system_attrs={}, value=38.685928224299865,␣
↪intermediate_values={}, params_in_internal_repr={'x': -4.219801301030433}, datetime_
↪start=datetime.datetime(2018, 5, 9, 10, 22, 59, 983824), datetime_complete=datetime.
↪datetime(2018, 5, 9, 10, 22, 59, 984053)),
 ...
 user_attrs={}, system_attrs={}, value=8.2881000286123179, intermediate_values={},␣
↪params_in_internal_repr={'x': 4.8789060472013182}, datetime_start=datetime.
↪datetime(2018, 5, 9, 10, 23, 0, 886434), datetime_complete=datetime.datetime(2018,␣
↪5, 9, 10, 23, 0, 891347))]
```

(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.Study(study_name='example-study', storage='sqlite:///example.db')
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.Study(study_name='example-study', storage='sqlite:///example.db')
df = study.trials_dataframe()
```

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

```
trial_id                state       value             datetime_start
↪datetime_complete    params

↪                   x
       1  TrialState.COMPLETE   46.904095 2018-10-31 16:06:28.264950 2018-10-31
↪16:06:28.296937  8.848656
       2  TrialState.COMPLETE   25.416075 2018-10-31 16:06:28.310073 2018-10-31
↪16:06:28.333799 -3.041436
       3  TrialState.COMPLETE   50.302101 2018-10-31 16:06:28.344672 2018-10-31
↪16:06:28.364514  9.092397
       4  TrialState.COMPLETE   53.415845 2018-10-31 16:06:28.380938 2018-10-31
↪16:06:28.400815 -5.308614
       5  TrialState.COMPLETE   29.780800 2018-10-31 16:06:28.415496 2018-10-31
↪16:06:28.449833  7.457179
       6  TrialState.COMPLETE    6.950141 2018-10-31 16:06:28.466843 2018-10-31
↪16:06:28.484284  4.636312
```

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.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).
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(step):
            raise optuna.structs.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 a trial 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 a trial 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 a trial 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 a trial 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 a trial 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 trial status as TrialState.PRUNED.
[I 2018-11-21 17:27:58,666] Setting trial status as TrialState.PRUNED.
[I 2018-11-21 17:27:58,675] Setting trial status as TrialState.PRUNED.
[I 2018-11-21 17:27:59,183] Finished a trial 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 trial status as TrialState.PRUNED.
...
```

We can see `Setting trial status 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`

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])
```

CHAPTER 3

API Reference

3.1 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.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=(<class 'Exception'>,)*)
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.**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.**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.

3.2 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()*.

3.3 Pruners

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

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.

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 `min_resource * (reduction_factor ** 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 `1/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., `min_resource * (reduction_factor ** (min_early_stopping_rate + 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 `eta`). At the completion point of each rung, about `1/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`).

3.4 Samplers

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

Sampler using random 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*)

3.5 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.

MNIMIZE
Study minimizes the objective function.

MAXIMIZE
Study maximizes the objective function.

class optuna.structs.**FrozenTrial**
Status and results of a *Trial*.

trial_id
Identifier of the *Trial*.

state
TrialState of the *Trial*.

value
Objective value of the *Trial*.

datetime_start
Datetime where the *Trial* started.

datetime_complete
Datetime where the *Trial* finished.

params
Dictionary that contains suggested parameters.

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

system_attrs
Dictionary that contains the attributes of the *Trial* internally set by Optuna.

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

params_in_internal_repr
Optuna's internal representation of *params*.

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.

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

class optuna.structs.**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(step):
>>>             raise TrailPruned()
```

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

CLI raises this exception when it receives invalid configuration.

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

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

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

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

3.6 Study

class optuna.study.**Study**(*study_name*, *storage*, *sampler=None*, *pruner=None*, *direction='minimize'*)
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.

Parameters

- **study_name** – Study's name. Each study has a unique name as an identifier.

- **storage** – Database URL such as `sqlite:///example.db`. Optuna internally uses SQLAlchemy to handle databases. Please refer to SQLAlchemy's document 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*.
- **direction** – Direction of optimization. Set `minimize` for minimization and `maximize` for maximization. Note that `maximize` is currently unsupported.

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=(<class 'Exception'>,)*)
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 exceptions specified in this argument. Default is (Exception,), where all non-exit exceptions are handled by this logic.

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*.

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.

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.
- **sampler** – A sampler object that implements background algorithm for value suggestion. 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. Note that `maximize` is currently unsupported.
- **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.**get_all_study_summaries**(*storage*)
Get all history of studies stored in a specified storage.

Parameters **storage** – Database URL.

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

3.7 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 this object is seamlessly instantiated and passed to the objective function behind *optuna.study.Study.optimize()* method (as well as optimize function); hence, in typical use cases, library users do not care about instantiation of this object.

Parameters

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

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.

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(step):
>>>             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*)
Judge whether the trial should be pruned.

This method calls prune method of the pruner, which judges whether the trial should be pruned at the given step. Please refer to the example code of *optuna.trial.Trial.report()*.

Parameters **step** – Step of the trial (e.g., epoch of neural network training).

Returns A boolean value. If `True`, the trial should be pruned. Otherwise, the trial will be continued.

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 `[low, high]`, and the step of discretization is `q`.

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 `[low, 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 `[low, high)` in the log domain.

Example

Suggest penalty parameter `C` of SVC.

```
>>> def objective(trial):
>>>     ...
>>>     c = trial.suggest_logunifrom('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 `[low, high)` in the linear domain.

Example

Suggest a dropout rate for neural network training.

```
>>> def objective(trial):
>>>     ...
>>>     dropout_rate = trial.suggest_unifrom('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.

3.8 Visualization

optuna.visualization.**plot_intermediate_values**(*study*)
Inside Jupyter notebook, plot intermediate values of all trials in a study.

Example

The following code snippet shows how to plot intermediate values inside Jupyter Notebook.

```
import optuna

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

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

optuna.visualization.plot_intermediate_values(study)
```

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

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 `trial_id` 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.trial_id), '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.trial_id), '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 trial status 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 trial status 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()
```

trial_id	state	value	...	params	system_attrs
0	TrialState.FAIL		...	0	Setting trial status 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

S

T

U

V

X