Optuna Dashboard

Optuna Dashboard Contributors.

Mar 29, 2024

CONTENTS:

Real-time dashboard for Optuna.

CHAPTER

ONE

GETTING STARTED

Fig. 1: Optuna Dashboard

1.1 Installation

1.1.1 Prerequisite

Optuna Dashboard supports Python 3.7 or newer.

1.1.2 Installing from PyPI

You can install optuna-dashboard via PyPI or Anaconda Cloud.

```
$ pip install optuna-dashboard
```

Also, you can install following optional dependencies to make optuna-dashboard faster.

```
$ pip install optuna-fast-fanova gunicorn
```

1.1.3 Installing from the source code

Since it requires to build TypeScript files, `pip install git+https://.../optuna-dashboard.git` does not actually work. Please clone the git repository and execute following commands to build sdist package:

```
$ git clone git@github.com:optuna/optuna-dashboard.git
$ cd optuna
```

```
# Node.js v16 is required to compile TypeScript files.
$ npm install
$ npm run build:prd
$ python -m build --sdist
```

Then you can install it like:

```
$ pip install dist/optuna-dashboard-x.y.z.tar.gz
```

See CONTRIBUTING.md for more details.

1.2 Command-line Interface

The most common usage of Optuna Dashboard is using the command-line interface. Assuming that Optuna's optimization history is persisted using `RDBStorage`, you can use the command line interface like `optuna-dashboard <STORAGE_URL>`.

```
import optuna

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

study = optuna.create_study(
    storage="sqlite:///db.sqlite3",  # Specify the storage URL here.
    study_name="quadratic-simple"
)
study.optimize(objective, n_trials=100)
print(f"Best value: {study.best_value} (params: {study.best_params})")
```

```
$ optuna-dashboard sqlite:///db.sqlite3
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

If you are using JournalStorage classes introduced in Optuna v3.1, you can use them like below:

```
# JournalFileStorage
$ optuna-dashboard ./path/to/journal.log

# JournalRedisStorage
$ optuna-dashboard redis://localhost:6379
```

1.3 Using an official Docker image

You can also use an official Docker image instead of setting up your Python environment. The Docker image only supports SQLite3, MySQL(PyMySQL), and PostgreSQL(Psycopg2).

SQLite3

```
$ docker run -it --rm -p 8080:8080 -v `pwd`:/app -w /app ghcr.io/optuna/optuna-dashboard␣
↪sqlite:///db.sqlite3
```

MySQL (PyMySQL)

```
$ docker run -it --rm -p 8080:8080 ghcr.io/optuna/optuna-dashboard mysql+pymysql://
↪username:password@hostname:3306/dbname
```

PostgreSQL (Psycopg2)

```
$ docker run -it --rm -p 8080:8080 ghcr.io/optuna/optuna-dashboard postgresql+psycopg2://
↪username:password@hostname:5432/dbname
```

1.4 Python Interface

Python interfaces are also provided for users who want to use other storage implementations (e.g. `InMemoryStorage`). You can use *run_server()* function like below:

```
import optuna
from optuna_dashboard import run_server

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

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

run_server(storage)
```

1.5 Using Gunicorn or uWSGI server

Optuna Dashboard uses wsgiref module, which is in the Python's standard libraries, by default. However, as described here, `wsgiref` is implemented for testing or debugging purpose. You can switch to other WSGI server implementations by using *wsgi()* function.

Listing 1: wsgi.py

```
from optuna.storages import RDBStorage
from optuna_dashboard import wsgi

storage = RDBStorage("sqlite:///db.sqlite3")
application = wsgi(storage)
```

Then please execute following commands to start.

```
$ pip install gunicorn
$ gunicorn --workers 4 wsgi:application
```

or

```
$ pip install uwsgi
$ uwsgi --http :8080 --workeers 4 --wsgi-file wsgi.py
```

1.6 Jupyter Lab Extension (Experimental)

You can install the Jupyter Lab extension via PyPI.

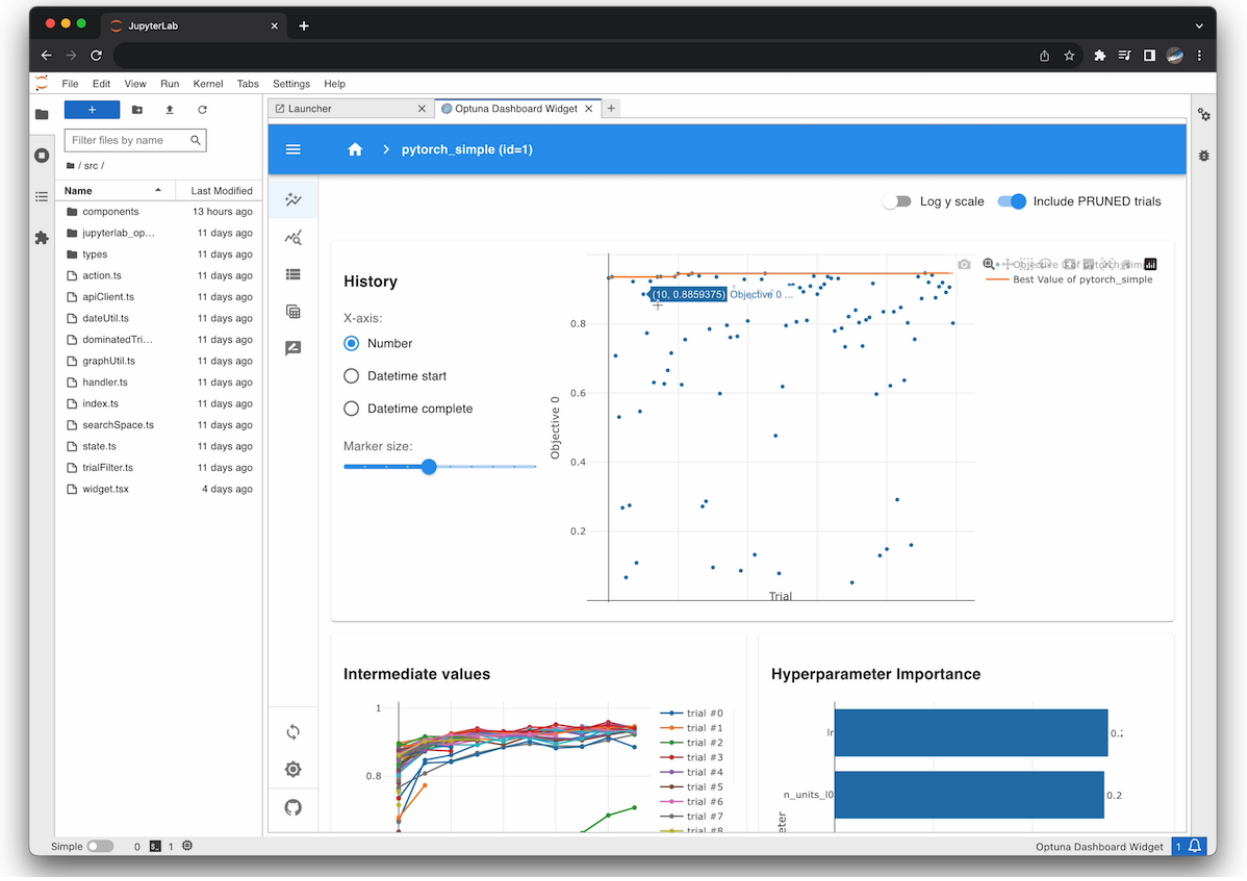


Fig. 2: Jupyter Lab Extension

To use, click the tile to launch the extension, and enter your Optuna's storage URL (e.g. `sqlite:///db.sqlite3`) in the dialog.

1.7 Browser-only version (Experimental)

Fig. 3: Browser-only version of Optuna Dashboard, powered by Wasm.

We've developed the version that operates solely within your web browser. There's no need to install Python or any other dependencies. Simply open the following URL in your browser, drag and drop your SQLite3 file onto the page, and you're ready to view your Optuna studies!

https://optuna.github.io/optuna-dashboard/

Warning: Currently, only a subset of features is available. However, you can still check the optimization history, hyperparameter importances, and etc. in graphs and tables.

1.8 VS Code and code-server Extension (Experimental)

You can install the VS Code extension via Visual Studio Marketplace, or install the code-server extension via Open VSX.

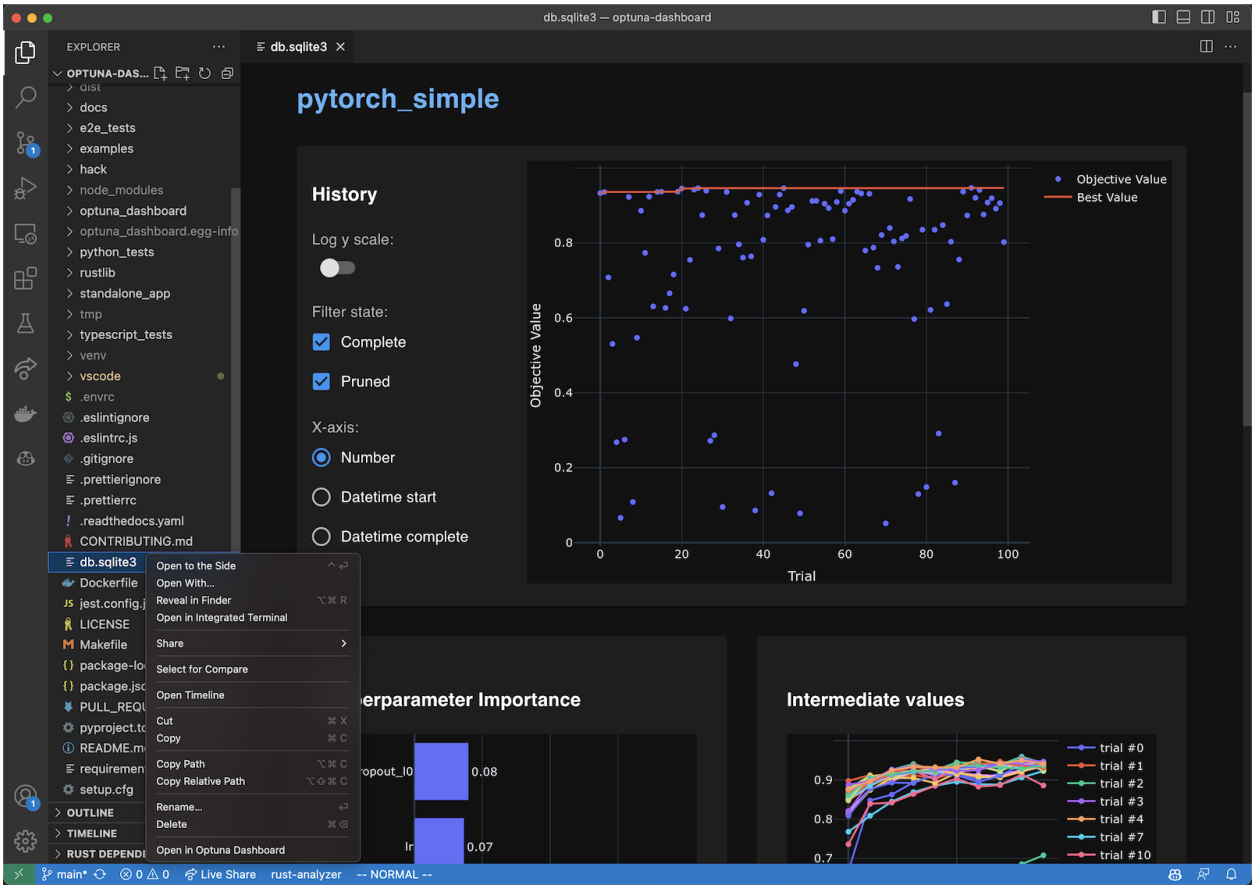


Fig. 4: VS Code Extension

To use, right-click the SQLite3 files (`*.db` or `*.sqlite3`) in the file explorer and select the "Open in Optuna Dashboard" from the dropdown menu. This extension leverages the browser-only version of Optuna Dashboard, so the same limitations apply.

1.9 Google Colaboratory

When you want to check the optimization history on Google Colaboratory, you can use `google.colab.output()` function as follows:

```
import optuna
import threading
from google.colab import output
from optuna_dashboard import run_server

def objective(trial):
    x = trial.suggest_float("x", -100, 100)
```

(continues on next page)

(continued from previous page)

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

# Run optimization
storage = optuna.storages.InMemoryStorage()
study = optuna.create_study(storage=storage)
study.optimize(objective, n_trials=100)

# Start Optuna Dashboard
port = 8081
thread = threading.Thread(target=run_server, args=(storage,), kwargs={"port": port})
thread.start()
output.serve_kernel_port_as_window(port, path='/dashboard/')
```

Then please open http://localhost:8081/dashboard to browse.

CHAPTER

TWO

API REFERENCE

2.1 General APIs

optuna_dashboard.run_server	Start running optuna-dashboard and blocks until the server terminates.
optuna_dashboard.wsgi	This function exposes WSGI interface for people who want to run on the production-class WSGI servers like Gunicorn or uWSGI.
optuna_dashboard.save_note	Save the note (Markdown format) to the Study or Trial.
optuna_dashboard.save_plotly_graph_object	Save the user-defined plotly's graph object to the study.
optuna_dashboard.artifact.get_artifact_path	Get the URL path for a given artifact ID.

2.1.1 optuna_dashboard.run_server

optuna_dashboard.**run_server**(*storage*, *host='localhost'*, *port=8080*, *artifact_store=None*, ***, *artifact_backend=None*)

Start running optuna-dashboard and blocks until the server terminates.

This function uses wsgiref module which is not intended for the production use. If you want to run optuna-dashboard more secure and/or more fast, please use WSGI server like Gunicorn or uWSGI via *wsgi()* function.

Parameters

- **storage** (*Union[str, BaseStorage]*) –
- **host** (*str*) –
- **port** (*int*) –
- **artifact_store** (*Optional[ArtifactStore | ArtifactBackend]*) –
- **artifact_backend** (*Optional[ArtifactBackend]*) –

Return type

None

2.1.2 optuna_dashboard.wsgi

optuna_dashboard.**wsgi**(*storage*, *artifact_store=None*, ***, *artifact_backend=None*)

This function exposes WSGI interface for people who want to run on the production-class WSGI servers like Gunicorn or uWSGI.

Parameters

- **storage** (*Union[str, BaseStorage]*) –
- **artifact_store** (*Optional[ArtifactBackend | ArtifactStore]*) –
- **artifact_backend** (*Optional[ArtifactBackend]*) –

Return type
WSGIApplication

2.1.3 optuna_dashboard.save_note

optuna_dashboard.**save_note**(*study_or_trial*, *body*)

Save the note (Markdown format) to the Study or Trial.

Example

```
import optuna
from optuna_dashboard import save_note


def objective(trial: optuna.Trial) -> float:
    x1 = trial.suggest_float("x1", 0, 10)

    save_note(trial, textwrap.dedent(f'''              ## Trial {trial.number}

    You can *freely* take a **note** that is associated with the Trial.
    '''))
    return (x1 - 2) ** 2


study = optuna.create_study()
save_note(study, textwrap.dedent(f'''          ## {study.study_name}

You can *freely* take a **note** that is associated with the study.
'''))
study.optimize(objective, n_trials=10)
```

Parameters

- **study_or_trial** (*Study | Trial*) –
- **body** (*str*) –

Return type
None

2.1.4 optuna_dashboard.save_plotly_graph_object

optuna_dashboard.**save_plotly_graph_object**(*study*, *figure*, *, *graph_object_id=None*)

Save the user-defined plotly's graph object to the study.

Example

```
import optuna
from optuna_dashboard import save_plotly_graph_object

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

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

figure = optuna.visualization.plot_optimization_history(study)
save_plotly_graph_object(study, figure)
```

Parameters

- **study** (*Study*) – Target study object.
- **plot_data** – The plotly's graph object to save.
- **graph_object_id** (*str | None*) – Unique identifier of the graph object. If specified, the graph object is overwritten. This must be a valid HTML id attribute value.
- **figure** (*go.Figure*) –

Returns

The graph object ID.

Return type

str

2.1.5 optuna_dashboard.artifact.get_artifact_path

optuna_dashboard.artifact.**get_artifact_path**(*study_or_trial*, *artifact_id*)

Get the URL path for a given artifact ID.

Parameters

- **study_or_trial** (*Trial | Study*) – A Trial object, or a Study object.
- **artifact_id** (*str*) – An artifact ID.

Returns

A URL path to the artifact.

Return type

str

2.2 Human-in-the-loop

2.2.1 Form Widgets

optuna_dashboard.register_objective_form_wi	Register a list of form widgets to an Optuna study.
optuna_dashboard.register_user_attr_form_wi	Register a list of form widgets to an Optuna study.
optuna_dashboard.dict_to_form_widget	Restore form widget objects from the dictionary.
optuna_dashboard.ChoiceWidget	A widget representing a choice with associated values.
optuna_dashboard.SliderWidget	A widget representing a slider for selecting a value within a range.
optuna_dashboard.TextInputWidget	A text input widget class that defines a text input field.
optuna_dashboard.ObjectiveUserAttrRef	A class representing a reference to a value of *trial.user_attrs*.

optuna_dashboard.register_objective_form_widgets

optuna_dashboard.**register_objective_form_widgets**(*study*, *widgets*)

Register a list of form widgets to an Optuna study.

Submitted values to the forms are told as each trial's objective values.

Parameters

- **study** (*Study*) – The Optuna study object to register the form widgets for.
- **widgets** (*list[ChoiceWidget | SliderWidget | TextInputWidget | ObjectiveUserAttrRef]*) – A list of ObjectiveFormWidget objects to be registered in the study.

Raises

- **ValueError** – If the length of study directions is not equal to the length of widgets.
- **Warning** – If any widget has *user_attr_key* specified, but it will not be used.

Return type

None

Examples

```
import optuna
from optuna_dashboard import ChoiceWidget, SliderWidget
from optuna_dashboard import register_objective_form_widgets


study = optuna.create_study()
register_objective_form_widgets(
    study,
    widgets=[
        ObjectiveChoiceWidget(
            choices=["Good ", "Bad "],
            values=[-1, 1],
            description="Please input your score!",
```

(continues on next page)

(continued from previous page)

```
        ),
        ObjectiveSliderWidget(
            min=1,
            max=10,
            step=1,
            description="Higher is better.",
        ),
    ],
)
```

optuna_dashboard.register_user_attr_form_widgets

optuna_dashboard.**register_user_attr_form_widgets**(*study*, *widgets*)

Register a list of form widgets to an Optuna study.

Submitted values to the forms are registered as each trial's user_attrs.

Parameters

- **study** (*Study*) – The Optuna study object to register the form widgets for.
- **widgets** (*list[ChoiceWidget | SliderWidget | TextInputWidget | ObjectiveUserAttrRef]*) – A list of ObjectiveFormWidget objects to be registered in the study.

Raises

- **ValueError** – If an ObjectiveUserAttrRef is specified or if *user_attr_key* is not specified.
- **ValueError** – If *user_attr_key* is not unique for each widget.

Return type

None

Examples

```
import optuna
from optuna_dashboard import ChoiceWidget, SliderWidget
from optuna_dashboard import register_user_attr_form_widgets


study = optuna.create_study()
register_user_attr_form_widgets(
    study,
    widgets=[
        ChoiceWidget(
            choices=["Good ", "Bad "],
            values=[-1, 1],
            description="Please input your score!",
            user_attr_key="hitl/choice",
        ),
        SliderWidget(
            min=1,
            max=10,
```

(continues on next page)

(continued from previous page)

```
            step=1,
            description="Higher is better.",
            user_attr_key="hitl/slider",
        ),
    ],
)
```

optuna_dashboard.dict_to_form_widget

optuna_dashboard.**dict_to_form_widget**(*d*)

Restore form widget objects from the dictionary.

Parameters
d (*dict[str, Any]*) – A dictionary object.

Returns
an instance of the restored form widget class.

Return type
object

optuna_dashboard.ChoiceWidget

class optuna_dashboard.**ChoiceWidget**(*choices*, *values*, *description=None*, *user_attr_key=None*)

A widget representing a choice with associated values.

Parameters

- **choices** (*list[str]*) – A list of strings representing the available choices.
- **values** (*list[float]*) – A list of float values associated with each choice.
- **description** (*Optional[str]*) – A description of the widget. Defaults to None.
- **user_attr_key** (*Optional[str]*) – The key used by *register_user_attr_form_widgets*. Form output is saved as *trial.user_attrs[user_attr_key]*. Defaults to None.

Example

```
from optuna_dashboard import ChoiceWidget


choice_widget = ChoiceWidget(
    choices=["A", "B", "C"], values=[1.0, 2.0, 3.0], description="Choose one"
)
```

Methods

to_dict()	Convert the ChoiceWidget object to a dictionary.

Attributes

description
user_attr_key
choices
values

to_dict()

Convert the ChoiceWidget object to a dictionary.

Returns
A dictionary representing the ChoiceWidget object.

Return type
ChoiceWidgetJSON

optuna_dashboard.SliderWidget

class optuna_dashboard.**SliderWidget**(*min*, *max*, *step=None*, *labels=None*, *description=None*, *user_attr_key=None*)

A widget representing a slider for selecting a value within a range.

Parameters

- **min** (*float*) – The minimum value of the slider.
- **max** (*float*) – The maximum value of the slider.
- **step** (*Optional[float]*) – The step size for the slider. Defaults to None.
- **labels** (*Optional[list[tuple[float, str]]]*) – A list of tuples containing value and label for the slider. Defaults to None.
- **description** (*Optional[str]*) – A description for the slider. Defaults to None.
- **user_attr_key** (*Optional[str]*) – The key used by *register_user_attr_form_widgets*. Form output is saved as *trial.user_attrs[user_attr_key]*. Defaults to None.

Example

```
from optuna_dashboard import SliderWidget


slide_widget = SliderWidget(min=0, max=10, step=1, description="Example slider")
```

Methods

to_dict()	Convert the SliderWidget instance to a dictionary.

Attributes

description
labels
step
user_attr_key
min
max

to_dict()

Convert the SliderWidget instance to a dictionary.

Returns
A dictionary representation of the SliderWidget instance.

Return type
SliderWidgetJSON

optuna_dashboard.TextInputWidget

class optuna_dashboard.**TextInputWidget**(*description=None*, *user_attr_key=None*, *optional=False*)

A text input widget class that defines a text input field.

Parameters

- **description** (*Optional[str]*) – A description of the text input field.
- **user_attr_key** (*Optional[str]*) – The key used by *register_user_attr_form_widgets*. Form output is saved as *trial.user_attrs[user_attr_key]*. Defaults to None.
- **optional** (*bool*) – If True, an empty string is acceptable.

Example

```
from optuna_dashboard import TextInputWidget


text_input = TextInputWidget(description="Text Input Example")
```

Methods

to_dict()	Converts the TextInputWidget instance to a dictionary representation.

Attributes

description
optional
user_attr_key

to_dict()

Converts the TextInputWidget instance to a dictionary representation.

Returns
The dictionary representation of the TextInputWidget instance.

Return type
TextInputWidgetJSON

optuna_dashboard.ObjectiveUserAttrRef

class optuna_dashboard.**ObjectiveUserAttrRef**(*key*)

A class representing a reference to a value of *trial.user_attrs*. When combined with *register_objective_form_widgets*, users can tell values that are registered to *trial.user_attrs* during the human-in-the-loop optimization.

Parameters
key (*str*) – The key of *trial.user_attrs* being referenced.

Example

```
from optuna_dashboard import ObjectiveUserAttrRef


user_attr_ref = ObjectiveUserAttrRef(key="key")
```

Methods

to_dict()	Converts the ObjectiveUserAttrRef instance to a dictionary representation.

Attributes

key

to_dict()

Converts the ObjectiveUserAttrRef instance to a dictionary representation.

Returns
The dictionary representation of the ObjectiveUserAttrRef instance.

Return type
UserAttrRefJSON

2.2.2 Preferential Optimization

optuna_dashboard.preferential.create_study	Like `optuna.create_study()`, but for preferential optimization.
optuna_dashboard.preferential.load_study	Like `optuna.load_study()`, but for preferential optimization.
optuna_dashboard.preferential.PreferentialStudy	A Study-like class for preferential optimization.
optuna_dashboard.preferential.samplers.gp.PreferentialGPSampler	Sampler for preferential optimization using Gaussian process.
optuna_dashboard.register_preference_feedba	Register a preference feedback component to the study.

optuna_dashboard.preferential.create_study

optuna_dashboard.preferential.**create_study**(*, *n_generate*, *storage=None*, *sampler=None*, *study_name=None*, *load_if_exists=False*)

Like `optuna.create_study()`, but for preferential optimization.

Example

```
import optuna
from optuna_dashboard.preferential import create_study


study = create_study()
trial = study.ask()
```

Parameters

- **n_generate** (*int*) – The number of active trials to keep. *should_generate()* returns `True` if the number of trials not reported bad and not skipped are less than `n_generate`.
- **storage** (*str | BaseStorage | None*) – Database URL. If this argument is set to None, in-memory storage is used, and the *PreferentialStudy* will not be persistent.
- **sampler** (*BaseSampler | None*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *PreferentialGPSampler* is used. Please note that most Optuna samplers does not work efficiently for preferential optimization.
- **study_name** (*str | None*) – Study's name. If this argument is set to None, a unique name is generated automatically.
- **load_if_exists** (*bool*) – Flag to control the behavior to handle a conflict of study names. In the case where a study named `study_name` already exists in the `storage`, a `DuplicatedStudyError` is raised if `load_if_exists` is set to `False`. Otherwise, the creation of the study is skipped, and the existing one is returned.

Returns

A *PreferentialStudy* object.

Return type

PreferentialStudy

Note: Preferential optimization is an experimental feature (introduced in v0.13.0). The interface may change in newer versions without prior notice.

---

optuna_dashboard.preferential.load_study

optuna_dashboard.preferential.**load_study**(*, *study_name*, *storage*, *sampler=None*)

Like `optuna.load_study()`, but for preferential optimization.

Example

```
import optuna
from optuna_dashboard.preferential import create_study
from optuna_dashboard.preferential import load_study

study = create_study(storage="sqlite:///example.db", study_name="my_study")
study.ask()

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

Parameters

- **study_name** (*str | None*) – Study's name. Each study has a unique name as an identifier. If `None`, checks whether the storage contains a single study, and if so loads that study. `study_name` is required if there are multiple studies in the storage.
- **storage** (*str | BaseStorage*) – Database URL such as `sqlite:///example.db`. Please see also the documentation of `create_study()` for further details.
- **sampler** (*BaseSampler | None*) – A sampler object that implements background algorithm for value suggestion. If `None` is specified, *PreferentialGPSampler* is used. Please note that most Optuna samplers does not work efficiently for preferential optimization.

Returns

A *PreferentialStudy* object.

Return type

PreferentialStudy

Note: Preferential optimization is an experimental feature (introduced in v0.13.0). The interface may change in newer versions without prior notice.

---

optuna_dashboard.preferential.PreferentialStudy

class optuna_dashboard.preferential.**PreferentialStudy**(*study*)

A Study-like class for preferential optimization.

This object provides interfaces to create a new Trial, set/get results of pairwise comparison called preferences.

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

Note: Preferential optimization is an experimental feature (introduced in v0.13.0). The interface may change in newer versions without prior notice.

Methods

add_trial(trial)	Add a trial to the study.
add_trials(trials)	Add trials to the study.
ask([fixed_distributions])	Create a new trial from which hyperparameters can be suggested.
enqueue_trial(params[, user_attrs, ...])	Enqueue a trial with given parameter values.
get_preferences(*[, deepcopy])	Return results of pairwise comparison.
get_trials([deepcopy, states])	Return the trials that is not dominated by other trials.
report_preference(better_trials, worse_trials)	Report results of pairwise comparison.
set_user_attr(key, value)	Set a user attribute to the study.
should_generate()	Return whether the generator should generate a new trial now.

Attributes

best_trials	Return the trials that is not dominated by other trials.
preferences	Return results of pairwise comparison.
study_name	Return the name of the study.
trials	Return the all trials.
user_attrs	Return user attributes of the study.

Parameters
study (*optuna.Study*) –

add_trial(*trial*)

Add a trial to the study.

See also:

See Study.add_trials() for details.

Parameters
trial (*FrozenTrial*) –

Return type
None

add_trials(*trials*)

Add trials to the study.

See also:

See Study.add_trials() for details.

Parameters
trials (*Iterable[FrozenTrial]*) –

Return type
None

ask(*fixed_distributions=None*)

Create a new trial from which hyperparameters can be suggested.

See also:

See Study.ask for details.

Parameters

fixed_distributions (*dict[str, BaseDistribution] | None*) – A dictionary containing the parameter names and parameter's distributions. Each parameter in this dictionary is automatically suggested for the returned trial, even when the suggest method is not explicitly invoked by the user. If this argument is set to `None`, no parameter is automatically suggested.

Returns

A Trial object.

Return type

Trial

property best_trials: list[FrozenTrial]

Return the trials that is not dominated by other trials.

Returns

A list of FrozenTrial object

enqueue_trial(*params*, *user_attrs=None*, *skip_if_exists=False*)

Enqueue a trial with given parameter values.

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

See also:

See **`Study.enqueue_trials`_** for details.

Parameters

- **params** (*dict[str, Any]*) – Parameter values to pass your objective function.
- **user_attrs** (*dict[str, Any] | None*) – A dictionary of user-specific attributes other than `params`.
- **skip_if_exists** (*bool*) – When `True`, prevents duplicate trials from being enqueued again.

Note: This method might produce duplicated trials if called simultaneously by multiple processes at the same time with same `params` dict.

Return type

None

get_preferences(*, *deepcopy=True*)

Return results of pairwise comparison.

Parameters

deepcopy (*bool*) – Flag to control whether to apply `copy.deepcopy()` to the trials. Note that if you set the flag to `False`, you shouldn't mutate any fields of the returned trial. Otherwise the internal state of the study may corrupt and unexpected behavior may happen.

Returns
A list of the pair of FrozenTrial objects. The left trial is better than the right one.

Return type
list[tuple[*FrozenTrial*, *FrozenTrial*]]

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

Return the trials that is not dominated by other trials.

See also:

See `Study.get_trials`_ for details.

Parameters

- **deepcopy** (*bool*) – Flag to control whether to apply `copy.deepcopy()` to the trials. Note that if you set the flag to `False`, you shouldn't mutate any fields of the returned trial. Otherwise the internal state of the study may corrupt and unexpected behavior may happen.
- **states** (*Container[TrialState] | None*) – Trial states to filter on. If `None`, include all states.

Returns
A list of FrozenTrial object

Return type
list[*FrozenTrial*]

`property preferences: list[tuple[FrozenTrial, FrozenTrial]]`

Return results of pairwise comparison.

Returns
A list of the pair of FrozenTrial objects. The left trial is better than the right one.

`report_preference`(*better_trials*, *worse_trials*)

Report results of pairwise comparison.

Parameters

- **better_trials** (*FrozenTrial | list[FrozenTrial]*) – Trials that are better than worse_trials.
- **worse_trials** (*FrozenTrial | list[FrozenTrial]*) – Trials that are worse than better_trials.

Return type
None

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

Set a user attribute to the study.

Parameters

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

Return type
None

See also:

See the tutorial for user attributes on Optuna's documentation.

should_generate()

Return whether the generator should generate a new trial now.

Returns `True` if the number of trials not reported bad and not skipped are less than `n_generate`. Users are recommended to generate a new trial if this method returns `True`, and to wait for human evaluation if this method returns `False`.

Return type
bool

property study_name: str

Return the name of the study.

Returns
A string object

property trials: list[FrozenTrial]

Return the all trials.

See also:

See Study.trials for details.

Returns
A list of FrozenTrial object

property user_attrs: dict[str, Any]

Return user attributes of the study.

See also:

See Study.user_attrs for details.

Returns
A dictionary containing all user attributes

optuna_dashboard.preferential.samplers.gp.PreferentialGPSampler

class optuna_dashboard.preferential.samplers.gp.**PreferentialGPSampler**(*, *kernel=None*, *noise_prior=None*, *independent_sampler=None*, *seed=None*)

Sampler for preferential optimization using Gaussian process.

The sampling algorithm is based on Takeno et al., 2023. This sampler uses BoTorch to optimize acquisition function.

Parameters

- **kernel** (*gpytorch.kernels.Kernel | None*) – Kernel that computes the covariance on the Gaussian process. Defaults to Matern 3/2 Kernel + ARD.
- **noise_prior** (*Prior | None*) – Prior of the observation noise. Defaults to gamma prior.
- **independent_sampler** (*optuna.samplers.BaseSampler | None*) – A `BaseSampler` instance that is used for independent sampling. The parameters not contained in the relative search space are sampled by this sampler. If `None` is specified, `RandomSampler` is used as the default.
- **seed** (*int | None*) – Seed for random number generator.

Methods

after_trial(study, trial, state, values)	Trial post-processing.
before_trial(study, trial)	Trial pre-processing.
infer_relative_search_space(study, trial)	Infer the search space that will be used by relative sampling in the target trial.
reseed_rng()	Reseed sampler's random number generator.
sample_independent(study, trial, param_name, ...)	Sample a parameter for a given distribution.
sample_relative(study, trial, search_space)	Sample parameters in a given search space.

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

Trial post-processing.

This method is called after the objective function returns and right before the trial is finished and its state is stored.

Note: Added in v2.4.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v2.4.0.

Parameters

- **study** (*Study*) – Target study object.
- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **state** (*TrialState*) – Resulting trial state.
- **values** (*Sequence[float] | None*) – Resulting trial values. Guaranteed to not be `None` if trial succeeded.

Return type
None

before_trial(*study*, *trial*)

Trial pre-processing.

This method is called before the objective function is called and right after the trial is instantiated. More precisely, this method is called during trial initialization, just before the `infer_relative_search_space()` call. In other words, it is responsible for pre-processing that should be done before inferring the search space.

Note: Added in v3.3.0 as an experimental feature. The interface may change in newer versions without prior notice. See https://github.com/optuna/optuna/releases/tag/v3.3.0.

Parameters

- **study** (*Study*) – Target study object.
- **trial** (*FrozenTrial*) – Target trial object.

Return type
None

infer_relative_search_space(*study*, *trial*)

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

This method is called right before sample_relative() method, and the search space returned by this method is passed to it. The parameters not contained in the search space will be sampled by using sample_independent() method.

Parameters

- **study** (*Study*) – Target study object.
- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.

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

Return type
dict[str, *BaseDistribution*]

See also:

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

reseed_rng()

Reseed sampler's random number generator.

This method is called by the Study instance if trials are executed in parallel with the option n_jobs>1. In that case, the sampler instance will be replicated including the state of the random number generator, and they may suggest the same values. To prevent this issue, this method assigns a different seed to each random number generator.

Return type
None

sample_independent(*study*, *trial*, *param_name*, *param_distribution*)

Sample a parameter for a given distribution.

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

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

Parameters

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

Returns
A parameter value.

Return type
Any

sample_relative(*study*, *trial*, *search_space*)

Sample parameters in a given search space.

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

Note: The failed trials are ignored by any build-in samplers when they sample new parameters. Thus, failed trials are regarded as deleted in the samplers' perspective.

Parameters

- **study** (*Study*) – Target study object.
- **trial** (*FrozenTrial*) – Target trial object. Take a copy before modifying this object.
- **search_space** (*dict[str, BaseDistribution]*) – The search space returned by `infer_relative_search_space()`.

Returns

A dictionary containing the parameter names and the values.

Return type

dict[str, *Any*]

optuna_dashboard.register_preference_feedback_component

optuna_dashboard.**register_preference_feedback_component**(*study*, *component_type*, *artifact_key=None*)

Register a preference feedback component to the study.

With this feature, you can change the component, displayed on the human feedback pages. By default, the Markdown note (`component_type="note"`) is displayed. If you specify `component_type="artifact"`, the viewer for the specified artifact file will be displayed.

Parameters

- **study** (PreferentialStudy) – The study to register the preference feedback component.
- **component_type** (*OUTPUT_COMPONENT_TYPE*) – The component type, displayed on the human feedback pages (default: `"note"`).
- **user_attr_artifact_key** – This option is required when the `component_type` is `"artifact"`. The user attribute, which is specified this field, must contain the ``artifact``id you want to display on the human feedback page.
- **artifact_key** (*str | None*) –

Return type

None

2.3 Streamlit

optuna_dashboard.streamlit.render_trial_note	Write a trial note to UI with streamlit as a markdown format.
optuna_dashboard.streamlit.render_objective_form_widgets	Render user input widgets to UI with streamlit.
optuna_dashboard.streamlit.render_user_attr_form_widgets	Render user input widgets to UI with streamlit.

2.3.1 optuna_dashboard.streamlit.render_trial_note

optuna_dashboard.streamlit.**render_trial_note**(*study*, *trial*)

Write a trial note to UI with streamlit as a markdown format.

Parameters

- **study** (*Study*) – The optuna study object.
- **trial** (*FrozenTrial*) – The optuna trial object to get note.

Return type
None

2.3.2 optuna_dashboard.streamlit.render_objective_form_widgets

optuna_dashboard.streamlit.**render_objective_form_widgets**(*study*, *trial*, *on_success_callback=None*)

Render user input widgets to UI with streamlit.

Submitted values to the forms are telled to optuna trial object. All submitted values should be float. Multiple widgets correspond to multi-objective optimization.

Parameters

- **study** (*optuna.Study*) – The optuna study object to get widget specification.
- **trial** (*FrozenTrial*) – The optuna trial object to tell user feedbacks.
- **on_success_callback** (*Optional[Callable[[], None]]*) – The callback function which will be executed when feedback submission is succeeded.

Raises

- **ValueError** – If No form widgets registered.
- **ValueError** – If 'output_type' of form widgets is not 'objective'.
- **ValueError** – If any submitted values cannot be converted to float.

Return type
None

2.3.3 optuna_dashboard.streamlit.render_user_attr_form_widgets

optuna_dashboard.streamlit.**render_user_attr_form_widgets**(*study*, *trial*, *on_success_callback=None*)

Render user input widgets to UI with streamlit.

Submitted values to the forms are registered as each trial's user_attrs.

Parameters

- **study** (*optuna.Study*) – The optuna study object to get widget specification.
- **trial** (*FrozenTrial*) – The optuna trial object to save user feedbacks.
- **on_success_callback** (*Optional[Callable[[], None]]*) – The callback function which will be executed when feedback submission is succeeded.

Raises

- **ValueError** – If No form widgets registered.
- **ValueError** – If 'output_type' of form widgets is not 'user_attr'.

Return type

None

CHAPTER

THREE

ERROR MESSAGES

This section lists descriptions and background for common error messages and warnings raised or emitted by Optuna Dashboard.

3.1 Warning Messages

3.1.1 Human-in-the-loop optimization will not work with `_CachedStorage` in Optuna prior to v3.2.

This warning occurs when the storage object associated with the Optuna Study is of the `_CachedStorage` class.

When using `RDBStorage` with Optuna, it is implicitly wrapped with the `_CachedStorage` class for performance improvement. However, there is a bug in the `_CachedStorage` class that prevents Optuna from synchronizing the latest Trial information. This bug is not a problem for the general use case of Optuna, but it is critical for human-in-the-loop optimization.

If you are using a version prior to v3.2, please upgrade to v3.2 or later, use another storage classes, or use a following dirty hack to unwrap `_CachedStorage` class.

```
if isinstance(study._storage, optuna.storages._CachedStorage):
    study._storage = study._storage._backend
```

3.1.2 `set_objective_names()` function is deprecated. Please use `study.set_metric_names()` instead.

`set_objective_names` function has been ported to Optuna. Please use study.set_metric_names() function instead.

Deprecated APIs	Corresponding Active APIs
`optuna_dashboard.set_objective_names(study, ["objective 1", "objective 2"])`	`study.set_metric_names(["objective 1", "objective 2"])`

3.1.3 `upload_artifact()` is deprecated. Please use `optuna.artifacts.upload_artifact()` instead.

`upload_artifact` function has been ported to Optuna. Please use optuna.artifacts.upload_artifact function instead.

Deprecated APIs	Corresponding Active APIs
`optuna_dashboard.artifact.upload_artifact(artifact_backend, trial, fiel_path)`	`optuna.artifacts.upload_artifact(trial, file_path, artifact_store)`

Please note that the order of arguments is different between the deprecated and active APIs.

3.1.4 `FileSystemBackend` is deprecated. Please use `FileSystemArtifactStore` instead.

`FileSystemBackend` class has been ported to Optuna. Please use FileSystemArtifactStore class instead.

Deprecated APIs	Corresponding Active APIs
`optuna_dashboard.artifact.file_system.FileSystemBackend(base_path)`	`optuna.artifacts.FileSystemArtifactStore(base_path)`

3.1.5 `Boto3Backend` is deprecated. Please use `Boto3ArtifactStore` instead.

`Boto3Backend` class has been ported to Optuna. Please use Boto3ArtifactStore class instead.

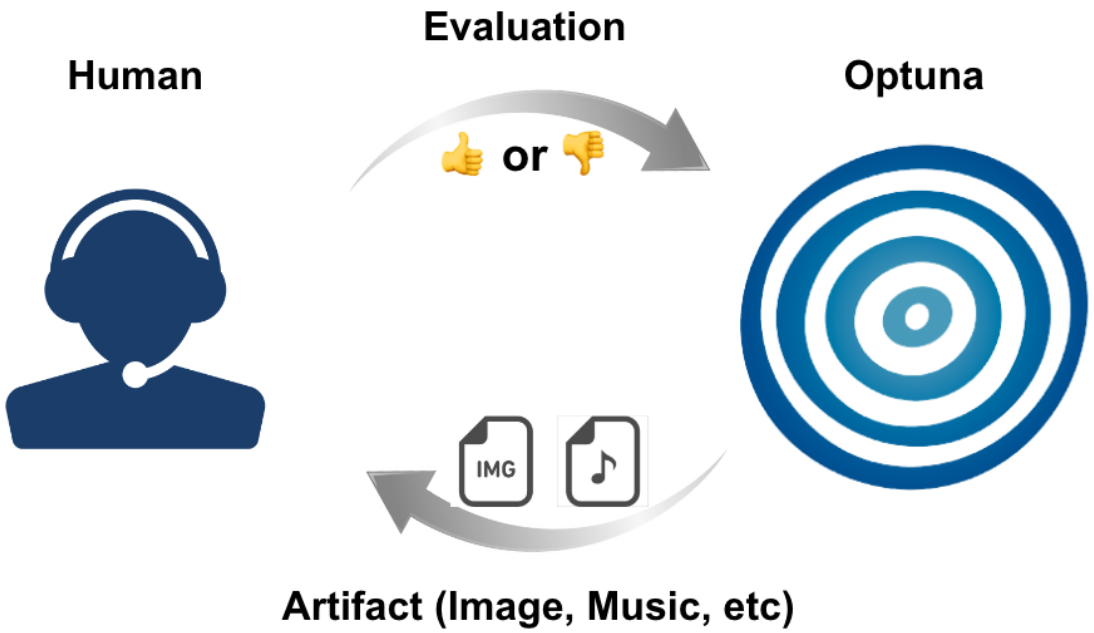
Deprecated APIs	Corresponding Active APIs
`optuna_dashboard.artifact.boto3.Boto3Backend(bucket_name, client=None)`	`optuna.artifacts.Boto3ArtifactStore(bucket_name, client=None)`

CHAPTER

FOUR

TUTORIALS

4.1 Tutorial: Human-in-the-loop Optimization using Objective Form Widgets



In tasks involving image generation, natural language, or speech synthesis, evaluating results mechanically can be tough, and human evaluation becomes crucial. Until now, managing such tasks with Optuna has been challenging. However, the introduction of Optuna Dashboard enables humans and optimization algorithms to work interactively and execute the optimization process.

In this tutorial, we will explain how to optimize hyperparameters to generate a simple image using Optuna Dashboard. While the tutorial focuses on a simple task, the same approach can be applied to for instance optimize more complex images, natural language, and speech.

The tutorial is organized as follows:

- What is human-in-the-loop optimization?

- Main tutorial
  - Theme
  - System architecture
  - Steps
  - Script explanation

4.1.1 What is human-in-the-loop optimization?

Human-in-the-loop (HITL) is a concept where humans play a role in machine learning or artificial intelligence systems. In HITL optimization in particular, humans are part of the optimization process. This is useful when it's difficult for machines to evaluate the results and human evaluation is crucial. In such cases, humans will assess the results instead.

Generally, HITL optimization involves the following steps:

1. An output is computed given the hyperparameters suggested by an optimization algorithm
2. An evaluator (human) evaluates the output

Steps 1 to 2 are repeated to find the best hyperparameters.

HITL optimization is valuable in areas where human judgment is essential, like art and design, since it's hard for machines to evaluate the output. For instance, it can optimize images created by generative models or improve cooking methods and ingredients for foods like coffee.

4.1.2 Main tutorial

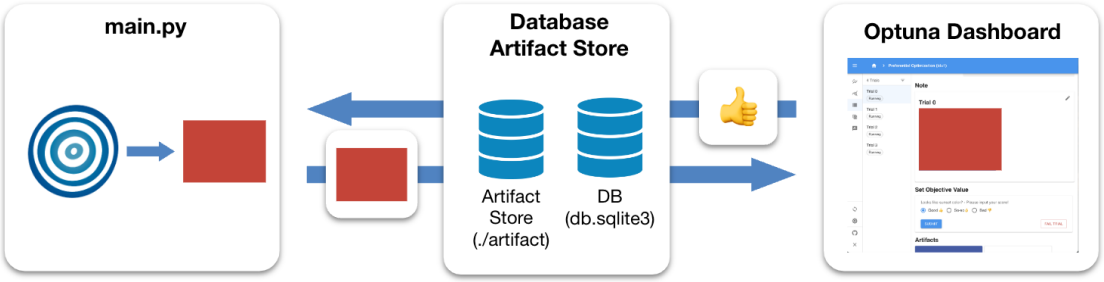
Theme

In this tutorial, we will interactively optimize RGB values between 0 and 255 to generate a color that the user perceives as the "color of the sunset." If someone already knows the RGB hyperparameter characteristics for their ideal "color of the sunset," they can specify those values directly. However, even without knowing such characteristics, interactive optimization allows us to find good hyperparameters. Although the task is simple, this serves as a practical introduction to human-in-the-loop optimization, and can be applied to image generation, natural language generation, speech synthesis, and more.

To implement HITL optimization, you need a way to interactively execute the optimization process, typically through a user interface (UI) or other means. Usually, users would have to implement their own, but with Optuna Dashboard, everything is already set up for you. This is a major advantage of using Optuna Dashboard for this purpose.

System architecture

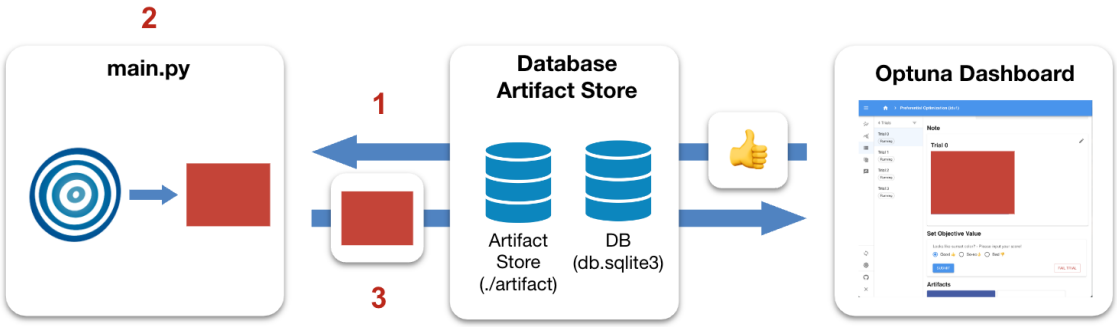
The system architecture for this tutorial's example is as follows:



In HITL optimization using Optuna Dashboard, there are primarily the following components:

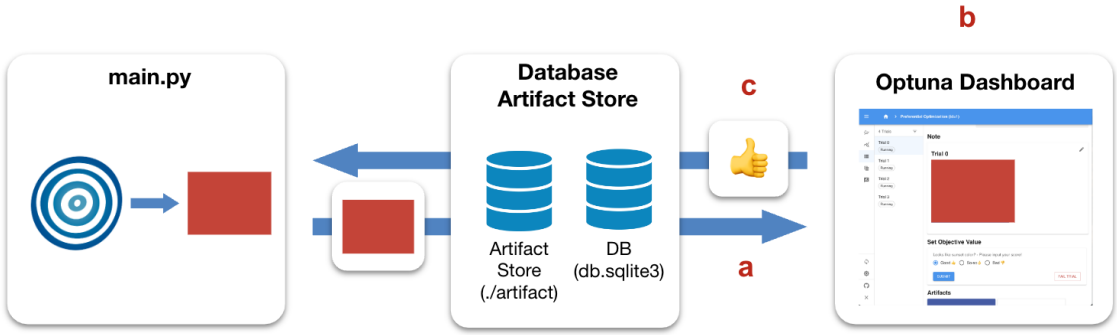
1. Optuna Dashboard for displaying the outputs and making evaluations
2. Database and File Storage to store the experiment's data (Study)
3. Script that samples hyperparameters from Optuna and generates outputs

The DB is the place where the information of the Study is stored. The Artifact Store is a place for storing artifacts (data, files, etc.) for the Optuna Dashboard. In this case, it is used as a storage location for the RGB images.



Our script repeatedly performs these steps:

1. Monitor the Study's state to maintain a constant number of Trials in progress (Running).
2. Sample hyperparameters using the optimization algorithm and generate RGB images.
3. Upload the generated RGB images to the Artifact Store.



Additionally, the evaluator, Optuna Dashboard, and Optuna perform the following processes:

a. Optuna Dashboard retrieves the RGB images uploaded to the Artifact Store and displays the retrieved RGB images to the evaluator

b. The evaluator reviews the displayed RGB images and inputs their evaluation of how close the displayed image is to the "color of the sunset" into the Optuna Dashboard

c. Optuna Dashboard saves the evaluation results in the database

In the example of this tutorial, processes 1-3 and a-c are executed in parallel.

Steps

Given the above system, we carry out HITL optimization as follows:

1. Environment setup
2. Execution of the HITL optimization script
3. Launching Optuna Dashboard
4. Interactive HITL optimization

Environment setup

To run the script used in this tutorial, you need to install following libraries:

```
$ pip install "optuna>=3.3.0" "optuna-dashboard>=0.12.0" pillow
```

You will use SQLite for the storage backend in this tutorial. Ensure that the following library is installed:

- SQLite

Execution of the HITL optimization script

Run a python script below which you copied from main.py

```
$ python main.py
```

Launching Optuna Dashboard

Run this command to launch Optuna Dashboard in a separate process.

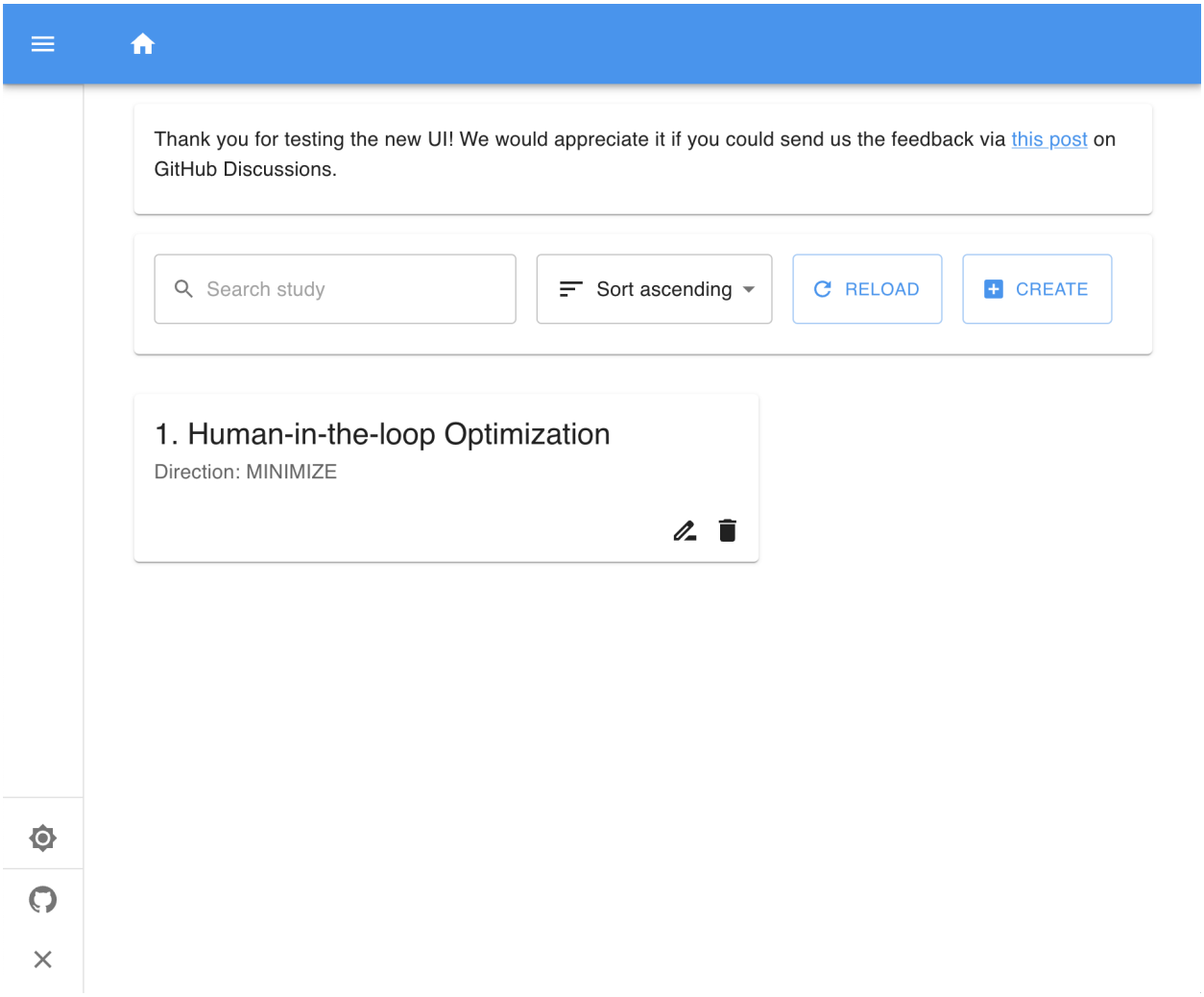
```
$ optuna-dashboard sqlite:///db.sqlite3 --artifact-dir ./artifact
```

In the command, the storage is set to `sqlite:///db.sqlite3` to persist Optuna's trial history. To store the artifacts, `--artifact-dir ./artifact` is specified.

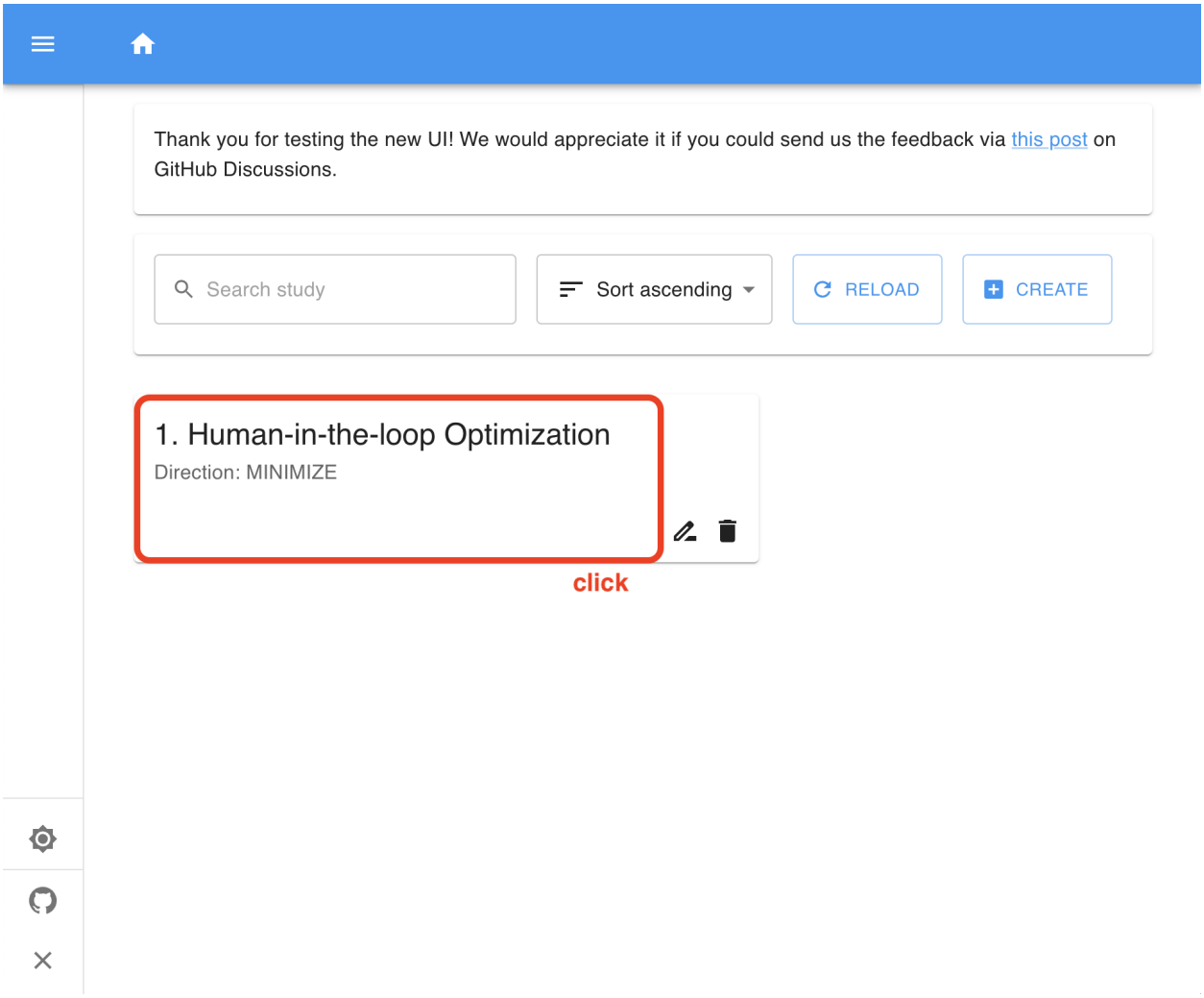
```
Listening on http://127.0.0.1:8080/
Hit Ctrl-C to quit.
```

When you run the command, you will see a message like the one above. Open http://127.0.0.1:8080/dashboard/ in your browser.

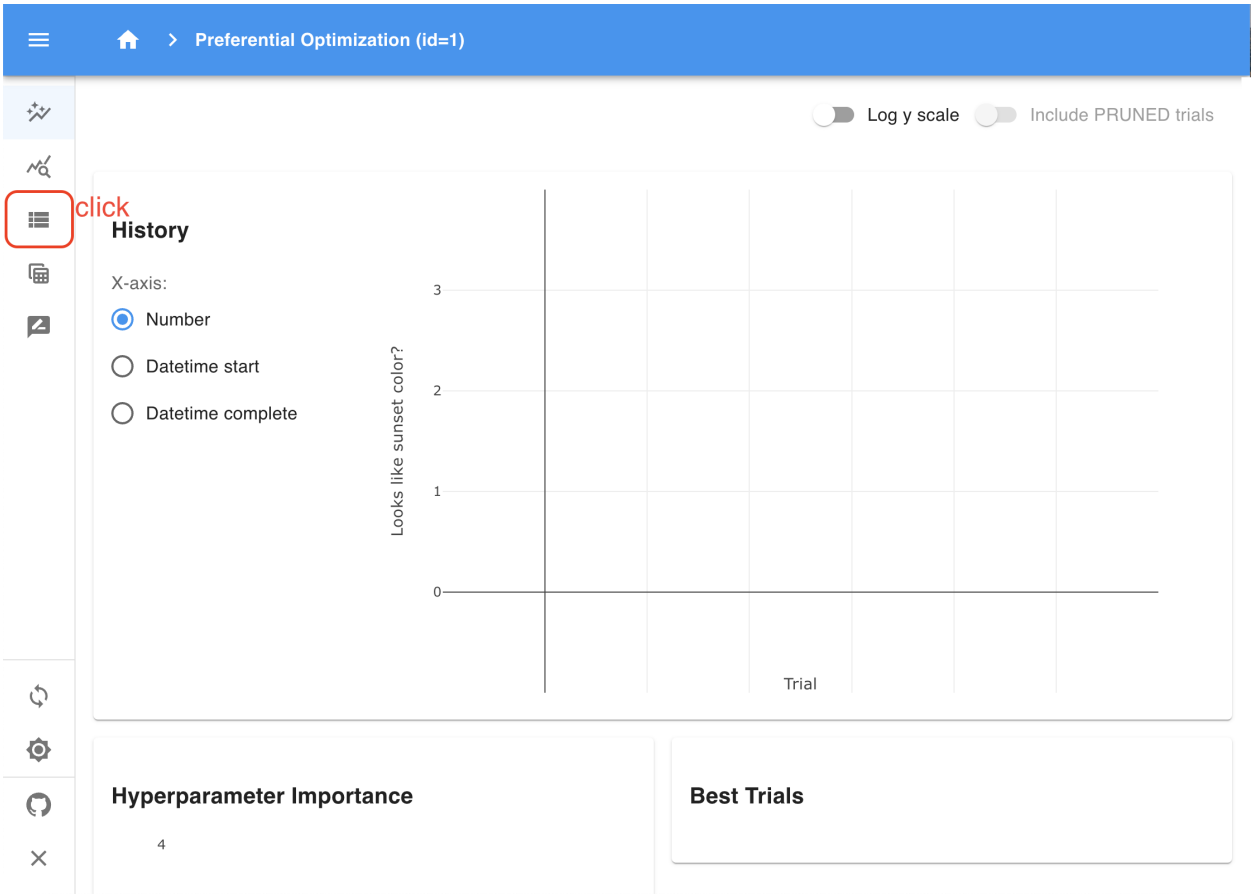
Interactive HITL optimization



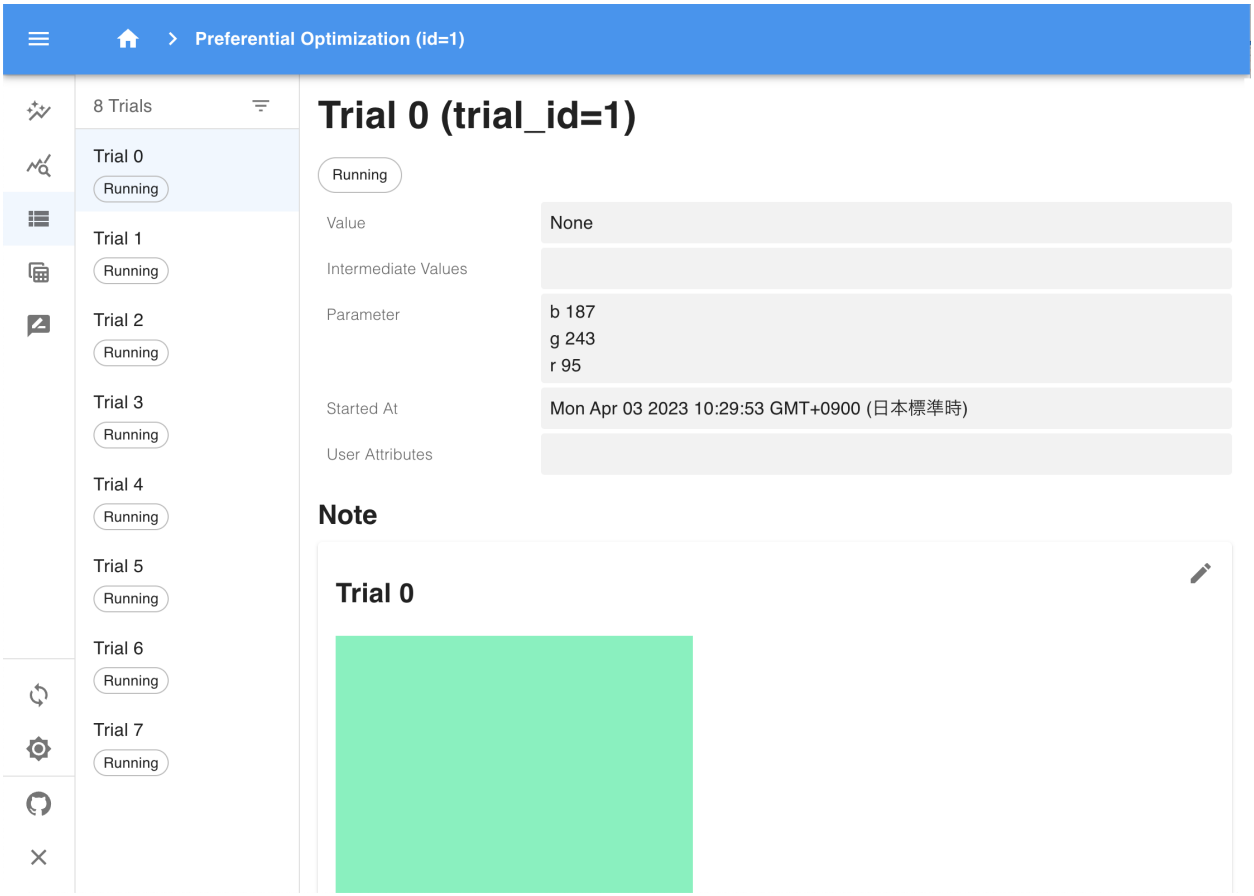
You will see the main screen.



In this example, a study is created with the name "Human-in-the-loop Optimization." Click on it. You will be directed to the page related to that study.



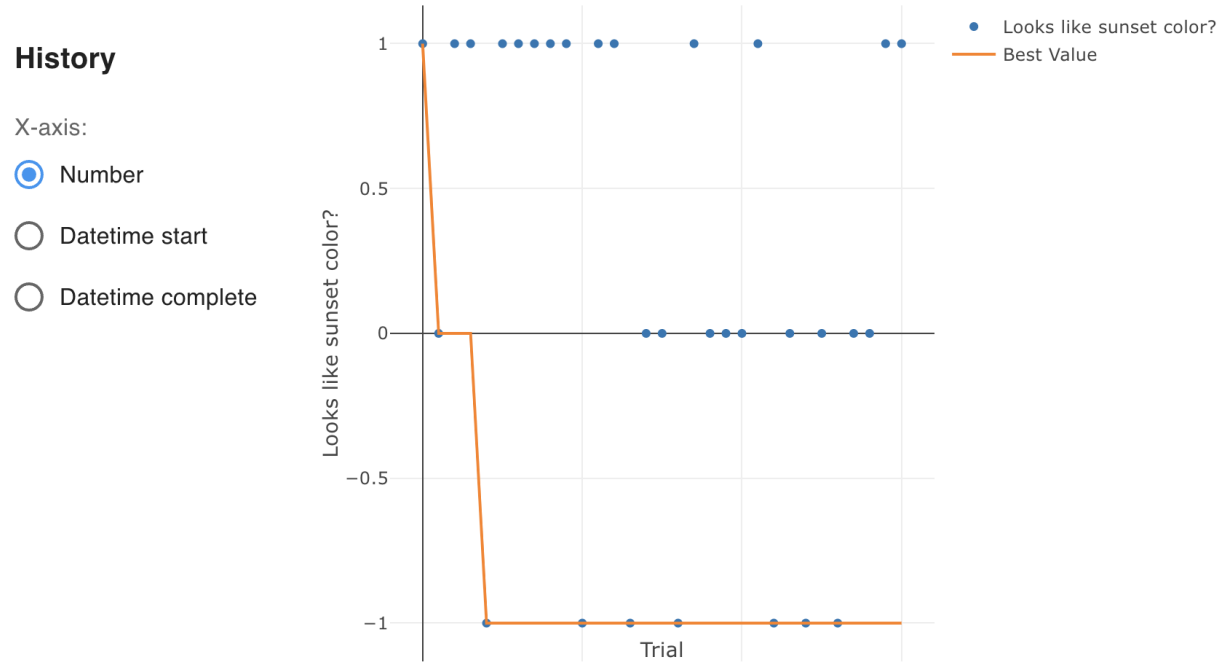
Click the third item in the sidebar. You will see a list of all trials.



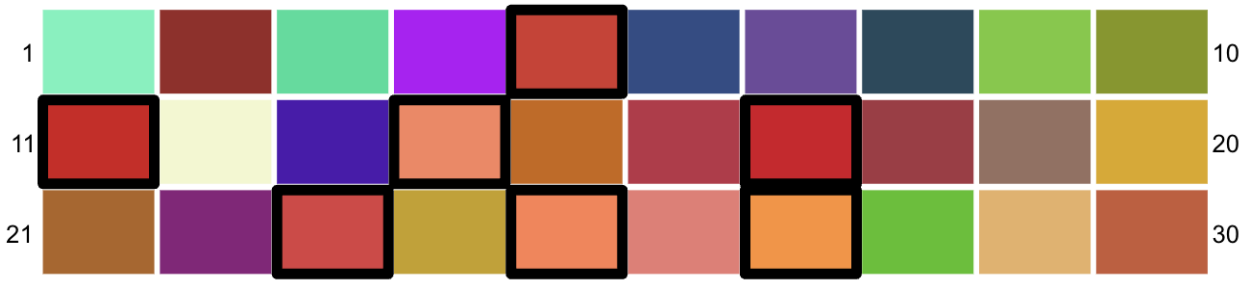
For each trial, you can see its details such as RGB parameter values and importantly, the generated image based on these values.

Let's evaluate some of the images. For the first image, which is far from the "color of the sunset," we rated it as "Bad." For the next image, which is somewhat closer to the "color of the sunset," we rated it as "So-so." Continue this evaluation process for several trials. After evaluating about 30 trials, we should see an improvement.

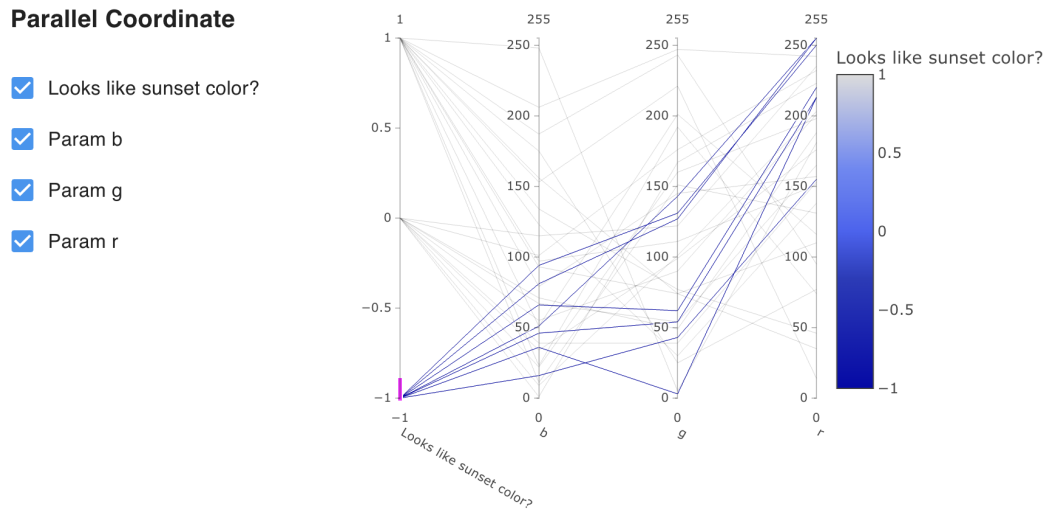
We can review the progress of the HITL optimization through graphs and other visualizations.



Also, this image is an array of images up to 30 trials. The best ones are surrounded by thick lines.



By looking at the History plot, you can see that colors gradually get closer to the "color of the sunset".



Additionally, by looking at the Parallel Coordinate plot, you can get an insight into the relationship between the evaluation and each hyperparameter.

Various other plots are available. Try exploring.

Script explanation

Let's walk through the script we used for the optimization.

```python
def suggest_and_generate_image(
    study: optuna.Study, artifact_store: FileSystemArtifactStore
) -> None:
    # 1. Ask new parameters
    trial = study.ask()
    r = trial.suggest_int("r", 0, 255)
    g = trial.suggest_int("g", 0, 255)
    b = trial.suggest_int("b", 0, 255)

    # 2. Generate image
    image_path = f"tmp/sample-{trial.number}.png"
    image = Image.new("RGB", (320, 240), color=(r, g, b))
    image.save(image_path)

    # 3. Upload Artifact
    artifact_id = upload_artifact(trial, image_path, artifact_store)
    artifact_path = get_artifact_path(trial, artifact_id)

    # 4. Save Note
    note = textwrap.dedent(
        f"""\
    ## Trial {trial.number}

    ![generated-image]({artifact_path})
    """
    )
    save_note(trial, note)
```

In the `suggest_and_generate_image` function, a new Trial is obtained and new hyperparameters are suggested for that Trial. Based on those hyperparameters, an RGB image is generated as an artifact. The generated image is then uploaded to the Artifact Store of the Optuna, and the image is also displayed in the Dashboard's Note. For more information on how to use the Note feature, please refer to the API Reference of *save_note()*.

```python
def start_optimization(artifact_store: FileSystemArtifactStore) -> NoReturn:
    # 1. Create Study
    study = optuna.create_study(
        study_name="Human-in-the-loop Optimization",
        storage="sqlite:///db.sqlite3",
        sampler=optuna.samplers.TPESampler(constant_liar=True, n_startup_trials=5),
        load_if_exists=True,
    )

    # 2. Set an objective name
    study.set_metric_names(["Looks like sunset color?"])
```

(continues on next page)

(continued from previous page)

```

    # 3. Register ChoiceWidget
    register_objective_form_widgets(
        study,
        widgets=[
            ChoiceWidget(
                choices=["Good ", "So-so", "Bad "],
                values=[-1, 0, 1],
                description="Please input your score!",
            ),
        ],
    )

    # 4. Start Human-in-the-loop Optimization
    n_batch = 4
    while True:
        running_trials = study.get_trials(deepcopy=False, states=(TrialState.RUNNING,))
        if len(running_trials) >= n_batch:
            time.sleep(1)  # Avoid busy-loop
            continue
        suggest_and_generate_image(study, artifact_store)
```

The function `start_optimization` defines our loop for HITL optimization to generate an image resembling a sunset color.

- First, at #1, a Study of Optuna is created using TPESampler. Setting `load_if_exists=True` allows a Study to exist and be reused and the experiment to be resumed if it has already been created. The reason for setting `constant_liar=True` in TPESampler is to prevent similar hyperparameters from being sampled even if the trial is executed several times simultaneously (in this example, four times).
- At #2, the name of the objective that the *`ChoiceWidget`* targets is set using the study.set_metric_names function. In this case, the name "Looks like sunset color?" is set.
- At #3, the *`ChoiceWidget`* is registered using the *`register_objective_form_widgets()`* function. This widget is used to ask users for evaluation to find the optimal hyperparameters. In this case, there are three options: "Good ", "So-so", and "Bad ", each with an evaluation value of -1, 0, and 1, respectively. Note that Optuna minimizes objective values by default, so -1 is Good. Other widgets for evaluation are also available, so please refer to the API Reference for details.
- At #4, the `suggest_and_generate_image` function is used to generate an RGB image. Here, the number of currently running (TrialState.RUNNING) trials is periodically checked to ensure that four trials are running simultaneously. The reason why trials are executed in batches like this is that it generally may take a long time to obtain results from trial execution. By performing batch parallel processing, time waiting for the next results can be reduced. In this case, because generating the images is instant, it's not necessary, but demonstrates practices.

```
def main() -> NoReturn:
    tmp_path = os.path.join(os.path.dirname(__file__), "tmp")

    # 1. Create Artifact Store
    artifact_path = os.path.join(os.path.dirname(__file__), "artifact")
    artifact_store = FileSystemArtifactStore(artifact_path)

    if not os.path.exists(artifact_path):
        os.mkdir(artifact_path)
```

(continues on next page)

(continued from previous page)

```
    if not os.path.exists(tmp_path):
        os.mkdir(tmp_path)

    # 2. Run optimize loop
    start_optimization(artifact_store)
```

In the `main` function, at first, the locations of the Artifact Store is set.

- At #1, the FileSystemArtifactStore is created, which is one of the Artifact Store options used in the Optuna. Artifact Store is used to store artifacts (data, files, etc.) generated during Optuna trials. For more information, please refer to the API Reference.
- At #2, *start_optimization()* function, which is described above, is called.

After that, two folders are created, artifact and tmp, and then `start_optimization` function is called to start the HITL optimization using Optuna.

4.2 Tutorial: Preferential Optimization

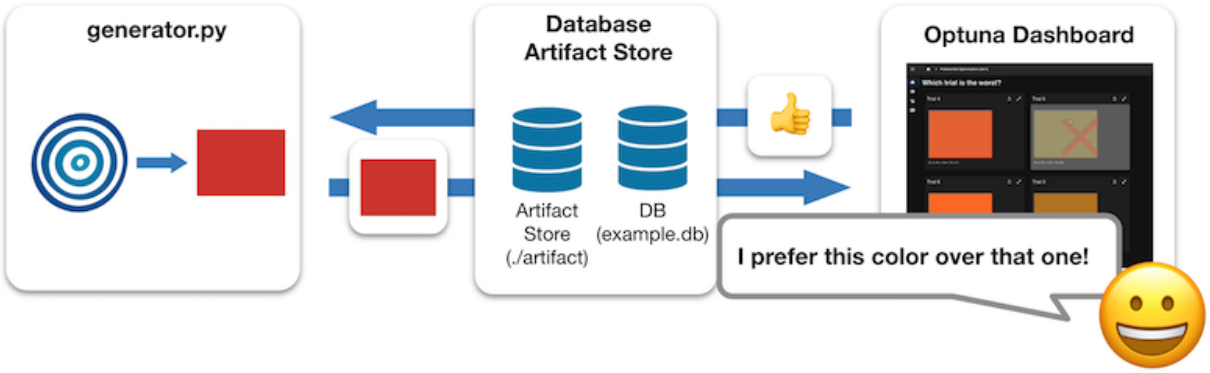
4.2.1 What is Preferential Optimization?

Preferential optimization is a method for optimizing hyperparameters, focusing of human preferences, by determining which trial is superior when comparing a pair. It differs from *human-in-the-loop optimization utilizing objective form widgets*, which relies on absolute evaluations, as it significantly reduces fluctuations in evaluators' criteria, thus ensuring more consistent results.

In this tutorial, we'll interactively optimize RGB values to generate a color resembling a "sunset hue", aligining with the problem setting in *this tutorial*. Familiarity with the tutorial ob objective form widgets may enhance your understanding.

4.2.2 How to Run Preferential Optimization

In preferential optimization, two programs run concurrently: generator.py performing parameter sampling and image generation, and the Optuna Dashboard, offering a user interface for human evaluation.



First, ensure the necessary packages are installed by executing the following command in your terminal:

```
$ pip install "optuna>=3.3.0" "optuna-dashboard[preferential]>=0.13.0b1" pillow
```

Next, execute the Python script, copied from generator.py.

```
$ python generator.py
```

Then, launch Optuna Dashboard in a separate process using the following command.

```
$ optuna-dashboard sqlite:///example.db --artifact-dir ./artifact
```

Here, the storage is configured to `sqlite:///example.db` to retain Optuna's trial history, and `--artifact-dir ./artifact` is specified to store the artifacts (output images).

```
Listening on http://127.0.0.1:8080/
Hit Ctrl-C to quit.
```

Upon executing the command, a message like the above will appear. Open http://127.0.0.1:8080/dashboard/ in your browser to view the Optuna Dashboard:

Fig. 1: Select the least sunset-like color from four trials to record human preferences.

4.2.3 Script Explanation

First, we specify the SQLite database URL and initialize the artifact store to house the images produced during the trial.

```
STORAGE_URL = "sqlite:///example.db"
artifact_path = os.path.join(os.path.dirname(__file__), "artifact")
artifact_store = FileSystemArtifactStore(base_path=artifact_path)
os.makedirs(artifact_path, exist_ok=True)
```

Within the `main()` function, creating dedicated `Study` and `Sampler` objects since preferential optimization relies on the comparison results between trials, lacking absolute evaluation values for each one.

Then, the component to be displayed on the human feedback pages is registered via *register_preference_feedback_component()*. The generated images are uploaded to the artifact store, and their `artifact_id` is stored in the trial user attribute (e.g., `trial.user_attrs["rgb_image"]`), enabling the Optuna Dashboard to display images on the evaluation feedback page.

```
from optuna_dashboard import register_preference_feedback_component
from optuna_dashboard.preferential import create_study
from optuna_dashboard.preferential.samplers.gp import PreferentialGPSampler

study = create_study(
    n_generate=4,
    study_name="Preferential Optimization",
    storage=STORAGE_URL,
    sampler=PreferentialGPSampler(),
    load_if_exists=True,
)
# Change the component, displayed on the human feedback pages.
# By default (component_type="note"), the Trial's Markdown note is displayed.
```

(continues on next page)

(continued from previous page)

```
user_attr_key = "rgb_image"
register_preference_feedback_component(study, "artifact", user_attr_key)
```

Following this, we create a loop that continuously checks if new trials should be generated, awaiting human evaluation if not. Within the while loop, new trials are generated if the condition *should_generate()* returns `True`. For each trial, RGB values are sampled, an image is generated with these values, saved temporarily. Then the image is uploaded to the artifact store, and finally, the `artifact_id` is stored to the key, which is specified via *register_preference_feedback_component()*.

```
while True:
    # If study.should_generate() returns False, the generator waits for human evaluation.
    if not study.should_generate():
        time.sleep(0.1)  # Avoid busy-loop
        continue

    trial = study.ask()
    # Ask new parameters
    r = trial.suggest_int("r", 0, 255)
    g = trial.suggest_int("g", 0, 255)
    b = trial.suggest_int("b", 0, 255)

    # Generate an image
    image_path = os.path.join(tmpdir, f"sample-{trial.number}.png")
    image = Image.new("RGB", (320, 240), color=(r, g, b))
    image.save(image_path)

    # Upload Artifact and set artifact_id to trial.user_attrs["rgb_image"].
    artifact_id = upload_artifact(trial, image_path, artifact_store)
    trial.set_user_attr(user_attr_key, artifact_id)
```

CHAPTER

FIVE

LICENSE

This software is licensed under the MIT license and uses the codes from SQLAlchemy (MIT) project, see LICENSE for more information.

CHAPTER

SIX

LINKS

- Github
- PyPI

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

S

T

U

W