Tune

Han Wang

Mar 19, 2023

CONTENTS

1 Introduction **1**
- 1.1 Installation . . . 1
- 1.2 Quick Start . . . 1
- 1.3 Design Philosophy . . . 1
- 1.4 Current Focuses . . . 2
- 1.5 Collaboration . . . 2

2 Top Level API Reference **3**
- 2.1 The Space Concept . . . 3
  - 2.1.1 Space . . . 3
  - 2.1.2 TuningParametersTemplate . . . 4
  - 2.1.3 Grid . . . 6
  - 2.1.4 Choice . . . 7
  - 2.1.5 TransitionChoice . . . 7
  - 2.1.6 Rand . . . 7
  - 2.1.7 RandInt . . . 8
- 2.2 General Non-Iterative Problems . . . 8
- 2.3 Level 2 Optimizers . . . 10
  - 2.3.1 Hyperopt . . . 10
  - 2.3.2 Optuna . . . 10
- 2.4 General Iterative Problems . . . 11
  - 2.4.1 Successive Halving . . . 11
  - 2.4.2 Hyperband . . . 11
  - 2.4.3 Continuous ASHA . . . 12
- 2.5 For Scikit-Learn . . . 13
- 2.6 For Tensorflow Keras . . . 15

3 Complete API Reference **19**
- 3.1 tune . . . 19
  - 3.1.1 tune.api . . . 19
  - 3.1.2 tune.concepts . . . 23
  - 3.1.3 tune.iterative . . . 40
  - 3.1.4 tune.noniterative . . . 43
  - 3.1.5 tune.constants . . . 47
  - 3.1.6 tune.exceptions . . . 47
- 3.2 tune_hyperopt . . . 47
  - 3.2.1 tune_hyperopt.optimizer . . . 47
- 3.3 tune_optuna . . . 48
  - 3.3.1 tune_optuna.optimizer . . . 48
- 3.4 tune_sklearn . . . 48

3.4.1 tune_sklearn.objective . . . 48
3.4.2 tune_sklearn.suggest . . . 49
3.4.3 tune_sklearn.utils . . . 51
3.5 tune_tensorflow . . . 51
3.5.1 tune_tensorflow.objective . . . 51
3.5.2 tune_tensorflow.spec . . . 52
3.5.3 tune_tensorflow.suggest . . . 53
3.5.4 tune_tensorflow.utils . . . 55
3.6 tune_notebook . . . 55
3.6.1 tune_notebook.monitors . . . 55
3.7 tune_test . . . 56
3.7.1 tune_test.local_optmizer . . . 56

4 Short Tutorials **59**
4.1 Search Space . . . 59
4.1.1 Simple Cases . . . 59
4.1.2 Grid Search . . . 59
4.1.3 Random Expressions . . . 60
4.1.4 Random Search . . . 66
4.1.5 Space Operations, Conditional Search and Hybrid Search . . . 67
4.2 Non-Iterative Tuning Guide . . . 69
4.2.1 Hello World . . . 69
4.2.2 Configuration . . . 70
4.2.3 Tuning Examples . . . 71
4.2.4 Realtime Monitoring . . . 75
4.2.5 Early Stopping . . . 77
4.3 Non-Iterative Objective . . . 80
4.3.1 Interfaceless . . . 80
4.3.2 Decorator Approach . . . 80
4.3.3 Interface Approach . . . 81
4.3.4 Factory Method . . . 81
4.4 Non-Iterative Optimizers . . . 82
4.4.1 Use Directly . . . 83
4.4.2 Use Top Level API . . . 83
4.4.3 Factory Method . . . 84
4.5 Tune Dataset . . . 85
4.6 Checkpoint . . . 89

Python Module Index **91**

Index **93**

CHAPTER

ONE

INTRODUCTION

Tune is an abstraction layer for general parameter tuning. It is built on Fugue so it can seamlessly run on any backend supported by Fugue, such as Spark, Dask and local.

1.1 Installation

```
pip install tune
```

It's recommended to also install Scikit-Learn (for all compatible models tuning) and Hyperopt (to enable Bayesian Optimization

```
pip install tune[hyperopt,sklearn]
```

1.2 Quick Start

To quickly start, please go through these tutorials on Kaggle:

1. Search Space
2. Non-iterative Problems, such as Scikit-Learn model tuning
3. Iterative Problems, such as Keras model tuning

1.3 Design Philosophy

Tune does not follow Scikit-Learn's model selection APIs and does not provide distributed backend for it. **We believe that parameter tuning is a general problem that is not only for machine learning**, so our abstractions are built from ground up, the lower level APIs do not assume the objective is a machine learning model, while the higher level APIs are dedicated to solve specific problems, such as Scikit-Learn compatible model tuning and Keras model tuning.

Although we didn't base our solution on any of HyperOpt, Optuna, Ray Tune and Nevergrad etc., we are truly inspired by these wonderful solutions and their design. We also integrated with many of them for deeper level optimizations.

Tuning problems are never easy, here are our goals:

- Provide the simplest and most intuitive APIs for major tuning cases. We always start from real tuning cases, figure out the minimal requirement for each of them and then determine the layers of abstraction. Read this tutorial, you can see how minimal the interfaces can be.

- Be scale agnostic and platform agnostic. We want you to worry less about *distributed computing*, and just focus on the tuning logic itself. Built on Fugue, Tune let you develop your tuning process iteratively. You can test with small spaces on local machine, and then switch to larger spaces and run distributedly with no code change. It can effectively save time and cost and make the process fun and rewarding. And to run any tuning logic distributedly, you only need a core framework itself (Spark, Dask, etc.) and you do not need a database, a queue service or even an embeded cluster.
- Be highly extendable and flexible on lower level. For example
  - you can extend on Fugue level, for example create an execution engine for Prefect to run the tuning jobs as a Prefect workflow
  - you can integrate third party optimizers and use Tune just as a distributed orchestrator.
  - you can start external instances (e.g. EC2 instances) for different training subtasks and to fully utilize your cloud
  - you can combine with distributed training as long as your have enough compute resource

1.4 Current Focuses

Here are our current focuses:

- A flexible space design and can describe a hybrid space of grid search, random search and second level optimization such as bayesian optimization
- Integrate with 3rd party tuning frameworks. We have integrated HyperOpt and Optuna. And Nevergrad is on the way.
- Create generalized and distributed versions of Successive Halving, Hyperband and Asynchronous Successive Halving.

1.5 Collaboration

We are looking for collaborators, if you are interested, please let us know.

Please join our Slack channel.

CHAPTER

TWO

TOP LEVEL API REFERENCE

2.1 The Space Concept

2.1.1 Space

class Space(**args*, ***kwargs*)

Bases: `object`

Search space object

Important: Please read *Space Tutorial*.

Parameters **kwargs** (*Any*) – parameters in the search space

```
Space(a=1, b=1)  # static space
Space(a=1, b=Grid(1,2), c=Grid("a", "b"))  # grid search
Space(a=1, b=Grid(1,2), c=Rand(0, 1))  # grid search + level 2 search
Space(a=1, b=Grid(1,2), c=Rand(0, 1)).sample(10, sedd=0)  # grid + random search

# union
Space(a=1, b=Grid(2,3)) + Space(b=Rand(1,5)).sample(10)

# cross product
Space(a=1, b=Grid(2,3)) * Space(c=Rand(1,5), d=Grid("a","b"))

# combo (grid + random + level 2)
space1 = Space(a=1, b=Grid(2,4))
space2 = Space(b=RandInt(10, 20))
space3 = Space(c=Rand(0,1)).sample(10)
space = (space1 + space2) * space3
```

```
assert Space(a=1, b=Rand(0,1)).has_stochastic
assert not Space(a=1, b=Rand(0,1)).sample(10).has_stochastic
assert not Space(a=1, b=Grid(0,1)).has_stochastic
assert not Space(a=1, b=1).has_stochastic

# get all configurations
space = Space(a=Grid(2,4), b=Rand(0,1)).sample(100)
```

(continues on next page)

(continued from previous page)

```
for conf in space:
    print(conf)
all_conf = list(space)
```

property has_stochastic
Whether the space contains any *StochasticExpression*

sample(*n*, *seed=None*)
Draw random samples from the current space. Please read *Space Tutorial*.

Parameters

- **n** (*int*) – number of samples to draw
- **seed** (*Optional[Any]*) – random seed, defaults to None

Returns a new Space containing all samples

Return type *tune.concepts.space.spaces.Space*

Note:

- it only applies to *StochasticExpression*
- if *has_stochastic()* is False, then it will return the original space
- After sampling, no *StochasticExpression* will exist in the new space.

2.1.2 TuningParametersTemplate

class TuningParametersTemplate(*raw*)
Bases: object

Parameter template to extract tuning parameter expressions from nested data structure

Parameters **raw** (*Dict[str, Any]*) – the dictionary of input parameters.

Note: Please use *to_template()* to initialize this class.

```
# common cases
to_template(dict(a=1, b=1))
to_template(dict(a=Rand(0, 1), b=1))

# expressions may nest in dicts or arrays
template = to_template(
    dict(a=dict(x1=Rand(0, 1), x2=Rand(3,4)), b=[Grid("a", "b")]))

assert [Rand(0, 1), Rand(3, 4), Grid("a", "b")] == template.params
assert dict(
    p0=Rand(0, 1), p1=Rand(3, 4), p2=Grid("a", "b")
) == template.params_dict
assert dict(a=1, x2=3), b=["a"]) == template.fill([1, 3, "a"])
assert dict(a=1, x2=3), b=["a"]) == template.fill_dict(
```

(continues on next page)

(continued from previous page)

```
    dict(p2="a", p1=3, p0=1)
)
```

concat(*other*)
Concatenate with another template and generate a new template.

Note: The other template must not have any key existed in this template, otherwise `ValueError` will be raised

---

Returns the merged template

Parameters **other** (tune.concepts.space.parameters.TuningParametersTemplate) –

Return type *tune.concepts.space.parameters.TuningParametersTemplate*

static decode(*data*)
Retrieve the template from a base64 string

Parameters **data** (*str*) –

Return type *tune.concepts.space.parameters.TuningParametersTemplate*

property empty: bool
Whether the template contains any tuning expression

encode()
Convert the template to a base64 string

Return type str

fill(*params*)
Fill the original data structure with values

Parameters

- **params** (*List[Any]*) – the list of values to be filled into the original data structure, in depth-first order
- **copy** – whether to return a deeply copied paramters, defaults to False

Returns the original data structure filled with values

Return type Dict[str, Any]

fill_dict(*params*)
Fill the original data structure with dictionary of values

Parameters

- **params** (*Dict[str, Any]*) – the dictionary of values to be filled into the original data structure, keys must be p0, p1, p2, ...
- **copy** – whether to return a deeply copied paramters, defaults to False

Returns the original data structure filled with values

Return type Dict[str, Any]

property has_grid: bool
Whether the template contains grid expressions

property has_stochastic: bool
Whether the template contains stochastic expressions

property params: List[*tune.concepts.space.parameters.TuningParameterExpression*]
Get all tuning parameter expressions in depth-first order

property params_dict: Dict[str, *tune.concepts.space.parameters.TuningParameterExpression*]
Get all tuning parameter expressions in depth-first order, with correspondent made-up new keys p0, p1, p2, ...

product_grid()
cross product all grid parameters

Yield new templates with the grid paramters filled

Return type Iterable[*tune.concepts.space.parameters.TuningParametersTemplate*]

```
assert [dict(a=1,b=Rand(0,1)), dict(a=2,b=Rand(0,1))] ==␣
↪list(to_template(dict(a=Grid(1,2),b=Rand(0,1))).product_grid())
```

sample(*n*, *seed=None*)
sample all stochastic parameters

Parameters

- **n** (*int*) – number of samples, must be a positive integer
- **seed** (*Optional[Any]*) – random seed defaulting to None. It will take effect if it is not None.

Yield new templates with the grid paramters filled

Return type Iterable[*tune.concepts.space.parameters.TuningParametersTemplate*]

```
assert [dict(a=1.1,b=Grid(0,1)), dict(a=1.5,b=Grid(0,1))] ==␣
↪list(to_template(dict(a=Rand(1,2),b=Grid(0,1))).sample(2,0))
```

property simple_value: Dict[str, Any]
If the template contains no tuning expression, it's simple and it will return parameters dictionary, otherwise, `ValueError` will be raised

property template: Dict[str, Any]
The template dictionary, all tuning expressions will be replaced by `None`

2.1.3 Grid

class Grid(**args*)
Bases: *tune.concepts.space.parameters.TuningParameterExpression*

Grid search, every value will be used. Please read *Space Tutorial*.

Parameters **args** (*Any*) – values for the grid search

2.1.4 Choice

class Choice(**args*)

Bases: *tune.concepts.space.parameters.StochasticExpression*

A random choice of values. Please read *Space Tutorial*.

Parameters **args** (*Any*) – values to choose from

generate(*seed=None*)

Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type Any

property jsondict: Dict[str, Any]

Dict representation of the expression that is json serializable

property values: List[Any]

values to choose from

2.1.5 TransitionChoice

class TransitionChoice(**args*)

Bases: *tune.concepts.space.parameters.Choice*

An ordered random choice of values. Please read *Space Tutorial*.

Parameters **args** (*Any*) – values to choose from

property jsondict: Dict[str, Any]

Dict representation of the expression that is json serializable

2.1.6 Rand

class Rand(*low*, *high*, *q=None*, *log=False*, *include_high=True*)

Bases: *tune.concepts.space.parameters.RandBase*

Continuous uniform random variables. Please read *Space Tutorial*.

Parameters

- **low** (*float*) – range low bound (inclusive)
- **high** (*float*) – range high bound (exclusive)
- **q** (*Optional[float]*) – step between adjacent values, if set, the value will be rounded using `q`, defaults to None
- **log** (*bool*) – whether to do uniform sampling in log space, defaults to False. If True, `low` must be positive and lower values get higher chance to be sampled
- **include_high** (*bool*) –

generate(*seed=None*)

Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type float

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

2.1.7 RandInt

class RandInt(*low*, *high*, *q=1*, *log=False*, *include_high=True*)
Bases: *tune.concepts.space.parameters.RandBase*

Uniform distributed random integer values. Please read *Space Tutorial*.

Parameters

- **low** (*int*) – range low bound (inclusive)
- **high** (*int*) – range high bound (exclusive)
- **log** (*bool*) – whether to do uniform sampling in log space, defaults to False. If True, `low` must be >=1 and lower values get higher chance to be sampled
- **q** (*int*) –
- **include_high** (*bool*) –

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type float

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

2.2 General Non-Iterative Problems

suggest_for_noniterative_objective(*objective*, *space*, *df=None*, *df_name='__tune__df_'*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *local_optimizer=None*, *logger=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *distributed=None*, *shuffle_candidates=True*, *execution_engine=None*, *execution_engine_conf=None*)
Given non-iterative `objective`, `space` and (optional) dataframe, suggest the best parameter combinations.

Important: Please read *Non-Iterative Tuning Guide*

Parameters

- **objective** (*Any*) – a simple python function or *NonIterativeObjectiveFunc* compatible object, please read *Non-Iterative Objective Explained*
- **space** (*tune.concepts.space.spaces.Space*) – search space, please read *Space Tutorial*
- **df** (*Optional[Any]*) – Pandas, Spark, Dask or any dataframe that can be converted to Fugue `DataFrame`, defaults to None
- **df_name** (*str*) – dataframe name, defaults to the value of `TUNE_DATASET_DF_DEFAULT_NAME`

- **temp_path** (*str*) – temp path for serialized dataframe partitions. It can be empty if you preset using TUNE_OBJECT_FACTORY.set_temp_path(). For details, read *TuneDataset Tutorial*, defaults to ""
- **partition_keys** (*Optional[List[str]]*) – partition keys for df, defaults to None. For details, please read *TuneDataset Tutorial*
- **top_n** (*int*) – number of best results to return, defaults to 1. If <=*0* all results will be returned
- **local_optimizer** (*Optional[Any]*) – an object that can be converted to *NonIterativeObjectiveLocalOptimizer*, please read *Non-Iterative Optimizers*, defaults to None
- **logger** (*Optional[Any]*) – MetricLogger object or a function producing it, defaults to None
- **monitor** (*Optional[Any]*) – realtime monitor, defaults to None. Read *Monitoring Guide*
- **stopper** (*Optional[Any]*) – early stopper, defaults to None. Read *Early Stopping Guide*
- **stop_check_interval** (*Optional[Any]*) – an object that can be converted to timedelta, defaults to None. For details, read to_timedelta()
- **distributed** (*Optional[bool]*) – whether to use the exeuction engine to run different trials distributedly, defaults to None. If None, it's equal to True.
- **shuffle_candidates** (*bool*) – whether to shuffle the candidate configurations, defaults to True. This is no effect on final result.
- **execution_engine** (*Optional[Any]*) – Fugue ExecutionEngine like object, defaults to None. If None, NativeExecutionEngine will be used, the task will be running on local machine.
- **execution_engine_conf** (*Optional[Any]*) – Parameters like object, defaults to None

Returns a list of best results

Return type List[*tune.concepts.flow.report.TrialReport*]

optimize_noniterative(*objective*, *dataset*, *optimizer=None*, *distributed=None*, *logger=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **optimizer** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **logger** (*Optional[Any]*) –
- **monitor** (*Optional[Any]*) –
- **stopper** (*Optional[Any]*) –
- **stop_check_interval** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

2.3 Level 2 Optimizers

2.3.1 Hyperopt

class HyperoptLocalOptimizer(*max_iter*, *seed=0*, *kwargs_func=None*)
Bases: *tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer*

Parameters

- **max_iter** (*int*) –
- **seed** (*int*) –
- **kwargs_func** (*Optional[Callable[[tune.noniterative.objective.NonIterativeObjectiveFunc, tune.concepts.flow.trial.Trial], Dict[str, Any]]]*) –

run(*func*, *trial*, *logger*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **logger** (*Any*) –

Return type *tune.concepts.flow.report.TrialReport*

2.3.2 Optuna

class OptunaLocalOptimizer(*max_iter*, *create_study=None*)
Bases: *tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer*

Parameters

- **max_iter** (*int*) –
- **create_study** (*Optional[Callable[[], optuna.study.study.Study]]*) –

run(*func*, *trial*, *logger*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **logger** (*Any*) –

Return type *tune.concepts.flow.report.TrialReport*

2.4 General Iterative Problems

2.4.1 Successive Halving

suggest_by_sha(*objective*, *space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **objective** (*Any*) –
- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

optimize_by_sha(*objective*, *dataset*, *plan*, *checkpoint_path=''*, *distributed=None*, *monitor=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **plan** (*List[Tuple[float, int]]*) –
- **checkpoint_path** (*str*) –
- **distributed** (*Optional[bool]*) –
- **monitor** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

2.4.2 Hyperband

suggest_by_hyperband(*objective*, *space*, *plans*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **objective** (*Any*) –
- **space** (tune.concepts.space.spaces.Space) –

- **plans** (*List[List[Tuple[float, int]]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

optimize_by_hyperband(*objective*, *dataset*, *plans*, *checkpoint_path=''*, *distributed=None*, *monitor=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **plans** (*List[List[Tuple[float, int]]]*) –
- **checkpoint_path** (*str*) –
- **distributed** (*Optional[bool]*) –
- **monitor** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

2.4.3 Continuous ASHA

suggest_by_continuous_asha(*objective*, *space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **objective** (*Any*) –
- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

optimize_by_continuous_asha(*objective*, *dataset*, *plan*, *checkpoint_path=''*, *always_checkpoint=False*, *study_early_stop=None*, *trial_early_stop=None*, *monitor=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **plan** (*List[Tuple[float, int]]*) –
- **checkpoint_path** (*str*) –
- **always_checkpoint** (*bool*) –
- **study_early_stop** (*Optional[Callable[[List[Any], List[*tune.iterative.asha.RungHeap*]], bool]]*) –
- **trial_early_stop** (*Optional[Callable[[*tune.concepts.flow.report.TrialReport*, List[*tune.concepts.flow.report.TrialReport*], List[*tune.iterative.asha.RungHeap*]], bool]]*) –
- **monitor** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

2.5 For Scikit-Learn

sk_space(*model*, ***params*)

Parameters

- **model** (*str*) –
- **params** (*Dict[str, Any]*) –

Return type *tune.concepts.space.spaces.Space*

suggest_sk_models_by_cv(*space*, *train_df*, *scoring*, *cv=5*, *temp_path=''*, *feature_prefix=''*, *label_col='label'*, *save_model=False*, *partition_keys=None*, *top_n=1*, *local_optimizer=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **train_df** (*Any*) –
- **scoring** (*str*) –
- **cv** (*int*) –
- **temp_path** (*str*) –
- **feature_prefix** (*str*) –
- **label_col** (*str*) –
- **save_model** (*bool*) –

- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **local_optimizer** (*Optional[tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer]*) –
- **monitor** (*Optional[Any]*) –
- **stopper** (*Optional[Any]*) –
- **stop_check_interval** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_sk_models(*space*, *train_df*, *test_df*, *scoring*, *temp_path=''*, *feature_prefix=''*, *label_col='label'*, *save_model=False*, *partition_keys=None*, *top_n=1*, *local_optimizer=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (*tune.concepts.space.spaces.Space*) –
- **train_df** (*Any*) –
- **test_df** (*Any*) –
- **scoring** (*str*) –
- **temp_path** (*str*) –
- **feature_prefix** (*str*) –
- **label_col** (*str*) –
- **save_model** (*bool*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **local_optimizer** (*Optional[tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer]*) –
- **monitor** (*Optional[Any]*) –
- **stopper** (*Optional[Any]*) –
- **stop_check_interval** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

2.6 For Tensorflow Keras

class KerasTrainingSpec(*params*, *dfs*)

Bases: object

Parameters

- **params** (*Any*) –
- **dfs** (*Dict[str, Any]*) –

compile_model(***add_kwargs*)

Parameters **add_kwargs** (*Any*) –

Return type keras.engine.training.Model

compute_sort_metric(***add_kwargs*)

Parameters **add_kwargs** (*Any*) –

Return type float

property dfs: Dict[str, Any]

finalize()

Return type None

fit(***add_kwargs*)

Parameters **add_kwargs** (*Any*) –

Return type keras.callbacks.History

generate_sort_metric(*metric*)

Parameters **metric** (*float*) –

Return type float

get_compile_params()

Return type Dict[str, Any]

get_fit_metric(*history*)

Parameters **history** (*keras.callbacks.History*) –

Return type float

get_fit_params()

Return type Tuple[List[Any], Dict[str, Any]]

get_model()

Return type keras.engine.training.Model

load_checkpoint(*fs*, *model*)

Parameters

- **fs** (*fs.base.FS*) –
- **model** (*keras.engine.training.Model*) –

Return type None

property params: *tune.concepts.space.parameters.TuningParametersTemplate*

save_checkpoint(*fs*, *model*)

Parameters

- **fs** (*fs.base.FS*) –
- **model** (*keras.engine.training.Model*) –

Return type None

keras_space(*model*, ***params*)

Parameters

- **model** (*Any*) –
- **params** (*Any*) –

Return type *tune.concepts.space.spaces.Space*

suggest_keras_models_by_continuous_asha(*space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (*tune.concepts.space.spaces.Space*) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_keras_models_by_hyperband(*space*, *plans*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **plans** (*List[List[Tuple[float, int]]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_keras_models_by_sha(*space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

CHAPTER

THREE

COMPLETE API REFERENCE

3.1 tune

3.1.1 tune.api

tune.api.factory

class TuneObjectFactory

Bases: object

get_path_or_temp(*path*)

Parameters **path** (*str*) –

Return type str

make_dataset(*dag*, *dataset*, *df=None*, *df_name='__tune__df_'*, *test_df=None*, *test_df_name='__tune__df__validation_'*, *partition_keys=None*, *shuffle=True*, *temp_path=''*)

Parameters

- **dag** (*fugue.workflow.workflow.FugueWorkflow*) –
- **dataset** (*Any*) –
- **df** (*Optional[Any]*) –
- **df_name** (*str*) –
- **test_df** (*Optional[Any]*) –
- **test_df_name** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **shuffle** (*bool*) –
- **temp_path** (*str*) –

Return type *tune.concepts.dataset.TuneDataset*

set_temp_path(*path*)

Parameters **path** (*str*) –

Return type None

tune.api.optimize

optimize_by_continuous_asha(*objective*, *dataset*, *plan*, *checkpoint_path=''*, *always_checkpoint=False*, *study_early_stop=None*, *trial_early_stop=None*, *monitor=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **plan** (*List[Tuple[float, int]]*) –
- **checkpoint_path** (*str*) –
- **always_checkpoint** (*bool*) –
- **study_early_stop** (*Optional[Callable[[List[Any], List[*tune.iterative.asha.RungHeap*]], bool]]*) –
- **trial_early_stop** (*Optional[Callable[[*tune.concepts.flow.report.TrialReport*, List[*tune.concepts.flow.report.TrialReport*], List[*tune.iterative.asha.RungHeap*]], bool]]*) –
- **monitor** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

optimize_by_hyperband(*objective*, *dataset*, *plans*, *checkpoint_path=''*, *distributed=None*, *monitor=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **plans** (*List[List[Tuple[float, int]]]*) –
- **checkpoint_path** (*str*) –
- **distributed** (*Optional[bool]*) –
- **monitor** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

optimize_by_sha(*objective*, *dataset*, *plan*, *checkpoint_path=''*, *distributed=None*, *monitor=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **plan** (*List[Tuple[float, int]]*) –
- **checkpoint_path** (*str*) –
- **distributed** (*Optional[bool]*) –
- **monitor** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

optimize_noniterative(*objective*, *dataset*, *optimizer=None*, *distributed=None*, *logger=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*)

Parameters

- **objective** (*Any*) –
- **dataset** (tune.concepts.dataset.TuneDataset) –
- **optimizer** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **logger** (*Optional[Any]*) –
- **monitor** (*Optional[Any]*) –
- **stopper** (*Optional[Any]*) –
- **stop_check_interval** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

tune.api.suggest

suggest_by_continuous_asha(*objective*, *space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **objective** (*Any*) –
- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_by_hyperband(*objective*, *space*, *plans*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **objective** (*Any*) –
- **space** (tune.concepts.space.spaces.Space) –
- **plans** (*List[List[Tuple[float, int]]]*) –

- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_by_sha(*objective*, *space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **objective** (*Any*) –
- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_for_noniterative_objective(*objective*, *space*, *df=None*, *df_name='__tune__df_'*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *local_optimizer=None*, *logger=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *distributed=None*, *shuffle_candidates=True*, *execution_engine=None*, *execution_engine_conf=None*)

Given non-iterative `objective`, `space` and (optional) dataframe, suggest the best parameter combinations.

Important: Please read *Non-Iterative Tuning Guide*

Parameters

- **objective** (*Any*) – a simple python function or *NonIterativeObjectiveFunc* compatible object, please read *Non-Iterative Objective Explained*
- **space** (tune.concepts.space.spaces.Space) – search space, please read *Space Tutorial*

- **df** (*Optional[Any]*) – Pandas, Spark, Dask or any dataframe that can be converted to Fugue DataFrame, defaults to None
- **df_name** (*str*) – dataframe name, defaults to the value of TUNE_DATASET_DF_DEFAULT_NAME
- **temp_path** (*str*) – temp path for serialized dataframe partitions. It can be empty if you preset using TUNE_OBJECT_FACTORY.set_temp_path(). For details, read *TuneDataset Tutorial*, defaults to ""
- **partition_keys** (*Optional[List[str]]*) – partition keys for df, defaults to None. For details, please read *TuneDataset Tutorial*
- **top_n** (*int*) – number of best results to return, defaults to 1. If <=*0* all results will be returned
- **local_optimizer** (*Optional[Any]*) – an object that can be converted to *NonIterativeObjectiveLocalOptimizer*, please read *Non-Iterative Optimizers*, defaults to None
- **logger** (*Optional[Any]*) – **|LoggerLikeObject|**, defaults to None
- **monitor** (*Optional[Any]*) – realtime monitor, defaults to None. Read *Monitoring Guide*
- **stopper** (*Optional[Any]*) – early stopper, defaults to None. Read *Early Stopping Guide*
- **stop_check_interval** (*Optional[Any]*) – an object that can be converted to timedelta, defaults to None. For details, read to_timedelta()
- **distributed** (*Optional[bool]*) – whether to use the exeuction engine to run different trials distributedly, defaults to None. If None, it's equal to True.
- **shuffle_candidates** (*bool*) – whether to shuffle the candidate configurations, defaults to True. This is no effect on final result.
- **execution_engine** (*Optional[Any]*) – Fugue ExecutionEngine like object, defaults to None. If None, NativeExecutionEngine will be used, the task will be running on local machine.
- **execution_engine_conf** (*Optional[Any]*) – Parameters like object, defaults to None

Returns a list of best results

Return type List[*tune.concepts.flow.report.TrialReport*]

3.1.2 tune.concepts

tune.concepts.flow

tune.concepts.flow.judge

class Monitor

Bases: object

finalize()

Return type None

initialize()

Return type None

on_get_budget(*trial*, *rung*, *budget*)

Parameters

- **trial** (tune.concepts.flow.trial.Trial) –
- **rung** (*int*) –
- **budget** (*float*) –

Return type None

on_judge(*decision*)

Parameters **decision** (tune.concepts.flow.judge.TrialDecision) –

Return type None

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type None

class NoOpTrailJudge(*monitor=None*)

Bases: *tune.concepts.flow.judge.TrialJudge*

Parameters **monitor** (*Optional[Monitor]*) –

can_accept(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

get_budget(*trial*, *rung*)

Parameters

- **trial** (tune.concepts.flow.trial.Trial) –
- **rung** (*int*) –

Return type float

judge(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type *tune.concepts.flow.judge.TrialDecision*

class RemoteTrialJudge(*entrypoint*)

Bases: *tune.concepts.flow.judge.TrialJudge*

Parameters **entrypoint** (*Callable[[str, Dict[str, Any]], Any]*) –

can_accept(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

get_budget(*trial*, *rung*)

Parameters

- **trial** (tune.concepts.flow.trial.Trial) –
- **rung** (*int*) –

Return type float

judge(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type *tune.concepts.flow.judge.TrialDecision*

property report: Optional[*tune.concepts.flow.report.TrialReport*]

class TrialCallback(*judge*)

Bases: object

Parameters **judge** (tune.concepts.flow.judge.TrialJudge) –

entrypoint(*name*, *kwargs*)

Parameters **kwargs** (*Dict[str, Any]*) –

Return type Any

class TrialDecision(*report*, *budget*, *should_checkpoint*, *reason=''*, *metadata=None*)

Bases: object

Parameters

- **report** (tune.concepts.flow.report.TrialReport) –
- **budget** (*float*) –
- **should_checkpoint** (*bool*) –
- **reason** (*str*) –
- **metadata** (*Optional[Dict[str, Any]]*) –

property budget: float

property metadata: Dict[str, Any]

property reason: str

property report: *tune.concepts.flow.report.TrialReport*

property should_checkpoint: bool

property should_stop: bool

property trial: *tune.concepts.flow.trial.Trial*

property trial_id: str

class TrialJudge(*monitor=None*)

Bases: object

Parameters **monitor** (*Optional[Monitor]*) –

can_accept(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

get_budget(*trial*, *rung*)

Parameters

- **trial** (tune.concepts.flow.trial.Trial) –
- **rung** (*int*) –

Return type float

judge(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type *tune.concepts.flow.judge.TrialDecision*

property monitor: *tune.concepts.flow.judge.Monitor*

reset_monitor(*monitor=None*)

Parameters **monitor** (*Optional[*tune.concepts.flow.judge.Monitor*]*) –

Return type None

tune.concepts.flow.report

class TrialReport(*trial*, *metric*, *params=None*, *metadata=None*, *cost=1.0*, *rung=0*, *sort_metric=None*, *log_time=None*)

Bases: object

The result from running the objective. It is immutable.

Parameters

- **trial** (tune.concepts.flow.trial.Trial) – the original trial sent to the objective
- **metric** (*Any*) – the raw metric from the objective output
- **params** (*Any*) – updated parameters based on the trial input, defaults to None. If none, it means the params from the trial was not updated, otherwise it is an object convertible to TuningParametersTemplate by *to_template()*
- **metadata** (*Optional[Dict[str, Any]]*) – metadata from the objective output, defaults to None
- **cost** (*float*) – cost to run the objective, defaults to 1.0
- **rung** (*int*) – number of rungs in the current objective, defaults to 0. This is for iterative problems
- **sort_metric** (*Any*) – the metric for comparison, defaults to None. It must be smaller better. If not set, it implies the metric is sort_metric and it is smaller better
- **log_time** (*Any*) – the time generating this report, defaults to None. If None, current time will be used

Attention: This class is not for users to construct directly.

copy()
Copy the current object.

Returns the copied object

Return type *tune.concepts.flow.report.TrialReport*

Note: This is shallow copy, but it is also used by *__deepcopy__* of this object. This is because we disable deepcopy of TrialReport.

---

property cost: float
The cost to run the objective

fill_dict(*data*)
Fill a row of *StudyResult* with the report information

Parameters **data** (*Dict[str, Any]*) – a row (as dict) from *StudyResult*

Returns the updated `data`

Return type Dict[str, Any]

generate_sort_metric(*min_better*, *digits*)
Construct a new report object with the new derived``sort_metric``

Parameters

- **min_better** (*bool*) – whether the current *metric()* is smaller better
- **digits** (*int*) – number of digits to keep in `sort_metric`

Returns a new object with the updated value

Return type *tune.concepts.flow.report.TrialReport*

property log_time: datetime.datetime
The time generating this report

property metadata: Dict[str, Any]
The metadata from the objective output

property metric: float
The raw metric from the objective output

property params: *tune.concepts.space.parameters.TuningParametersTemplate*
The parameters used by the objective to generate the *metric()*

reset_log_time()
Reset *log_time()* to now

Return type *tune.concepts.flow.report.TrialReport*

property rung: int
The number of rungs in the current objective, defaults to 0. This is for iterative problems

property sort_metric: float
The metric for comparison

property trial: *tune.concepts.flow.trial.Trial*
The original trial sent to the objective

property trial_id: str
tune.concepts.flow.trial.Trial.trial_id()

with_cost(*cost*)
Construct a new report object with the new `cost`

Parameters **cost** (*float*) – new cost

Returns a new object with the updated value

Return type *tune.concepts.flow.report.TrialReport*

with_rung(*rung*)
Construct a new report object with the new `rung`

Parameters **rung** (*int*) – new rung

Returns a new object with the updated value

Return type *tune.concepts.flow.report.TrialReport*

with_sort_metric(*sort_metric*)
Construct a new report object with the new `sort_metric`

Parameters **sort_metric** (*Any*) – new sort_metric

Returns a new object with the updated value

Return type *tune.concepts.flow.report.TrialReport*

class TrialReportHeap(*min_heap*)
Bases: `object`

Parameters **min_heap** (*bool*) –

pop()

Return type *tune.concepts.flow.report.TrialReport*

push(*report*)

Parameters **report** (`tune.concepts.flow.report.TrialReport`) –

Return type None

values()

Return type Iterable[*tune.concepts.flow.report.TrialReport*]

class TrialReportLogger(*new_best_only=False*)
Bases: `object`

Parameters **new_best_only** (*bool*) –

property best: Optional[*tune.concepts.flow.report.TrialReport*]

log(*report*)

Parameters **report** (`tune.concepts.flow.report.TrialReport`) –

Return type None

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type bool

tune.concepts.flow.trial

class Trial(*trial_id*, *params*, *metadata=None*, *keys=None*, *dfs=None*)

Bases: object

The input data collection for running an objective. It is immutable.

Parameters

- **trial_id** (*str*) – the unique id for a trial
- **params** (*Any*) – parameters for tuning, an object convertible to TuningParametersTemplate by *to_template()*
- **metadata** (*Optional[Dict[str, Any]]*) – metadata for tuning, defaults to None. It is set during the construction of *TuneDataset*
- **keys** (*Optional[List[str]]*) – partitions keys of the *TuneDataset*, defaults to None
- **dfs** (*Optional[Dict[str, Any]]*) – dataframes extracted from *TuneDataset*, defaults to None

Attention: This class is not for users to construct directly. Use *Space* instead.

copy()

Copy the current object.

Returns the copied object

Return type *tune.concepts.flow.trial.Trial*

Note: This is shallow copy, but it is also used by *__deepcopy__* of this object. This is because we disable deepcopy of Trial.

property dfs: Dict[str, Any]

Dataframes extracted from *TuneDataset*

property keys: List[str]

Partitions keys of the *TuneDataset*

property metadata: Dict[str, Any]

Metadata of the trial

property params: *tune.concepts.space.parameters.TuningParametersTemplate*

Parameters for tuning

property trial_id: str

The unique id of this trial

with_dfs(*dfs*)

Set dataframes for the trial, a new Trial object will be constructed and with the new dfs

Parameters **dfs** (*Dict[str, Any]*) – dataframes to attach to the trial

Return type *tune.concepts.flow.trial.Trial*

with_params(*params*)
Set parameters for the trial, a new Trial object will be constructed and with the new `params`

Parameters **params** (*Any*) – parameters for tuning

Return type *tune.concepts.flow.trial.Trial*

tune.concepts.space

tune.concepts.space.parameters

class Choice(**args*)
Bases: *tune.concepts.space.parameters.StochasticExpression*

A random choice of values. Please read *Space Tutorial*.

Parameters **args** (*Any*) – values to choose from

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type Any

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

property values: List[Any]
values to choose from

class FuncParam(*func*, **args*, ***kwargs*)
Bases: `object`

Function paramter. It defers the function call after all its parameters are no longer tuning parameters

Parameters

- **func** (*Callable*) – function to generate parameter value
- **args** (*Any*) – list arguments
- **kwargs** (*Any*) – key-value arguments

```
s = Space(a=1, b=FuncParam(lambda x, y: x + y, x=Grid(0, 1), y=Grid(3, 4)))
assert [
    dict(a=1, b=3),
    dict(a=1, b=4),
    dict(a=1, b=4),
    dict(a=1, b=5),
] == list(s)
```

class Grid(**args*)
Bases: *tune.concepts.space.parameters.TuningParameterExpression*

Grid search, every value will be used. Please read *Space Tutorial*.

Parameters **args** (*Any*) – values for the grid search

class NormalRand(*mu*, *sigma*, *q=None*)
Bases: *tune.concepts.space.parameters.RandBase*

Continuous normally distributed random variables. Please read *Space Tutorial*.

Parameters

- **mu** (*float*) – mean of the normal distribution
- **sigma** (*float*) – standard deviation of the normal distribution
- **q** (*Optional[float]*) – step between adjacent values, if set, the value will be rounded using q, defaults to None

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type float

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

class NormalRandInt(*mu*, *sigma*, *q=1*)
Bases: *tune.concepts.space.parameters.RandBase*

Normally distributed random integer values. Please read *Space Tutorial*.

Parameters

- **mu** (*int*) – mean of the normal distribution
- **sigma** (*float*) – standard deviation of the normal distribution
- **q** (*int*) –

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type int

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

class Rand(*low*, *high*, *q=None*, *log=False*, *include_high=True*)
Bases: *tune.concepts.space.parameters.RandBase*

Continuous uniform random variables. Please read *Space Tutorial*.

Parameters

- **low** (*float*) – range low bound (inclusive)
- **high** (*float*) – range high bound (exclusive)
- **q** (*Optional[float]*) – step between adjacent values, if set, the value will be rounded using q, defaults to None
- **log** (*bool*) – whether to do uniform sampling in log space, defaults to False. If True, `low` must be positive and lower values get higher chance to be sampled
- **include_high** (*bool*) –

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call `seed()` , defaults to None

Return type float

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

class RandBase(*q=None*, *log=False*)
Bases: *tune.concepts.space.parameters.StochasticExpression*

Base class for continuous random variables. Please read *Space Tutorial*.

Parameters

- **q** (*Optional[float]*) – step between adjacent values, if set, the value will be rounded using q, defaults to None
- **log** (*bool*) – whether to do uniform sampling in log space, defaults to False. If True, lower values get higher chance to be sampled

class RandInt(*low*, *high*, *q=1*, *log=False*, *include_high=True*)
Bases: *tune.concepts.space.parameters.RandBase*

Uniform distributed random integer values. Please read *Space Tutorial*.

Parameters

- **low** (*int*) – range low bound (inclusive)
- **high** (*int*) – range high bound (exclusive)
- **log** (*bool*) – whether to do uniform sampling in log space, defaults to False. If True, low must be >=1 and lower values get higher chance to be sampled
- **q** (*int*) –
- **include_high** (*bool*) –

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call seed() , defaults to None

Return type float

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

class StochasticExpression
Bases: *tune.concepts.space.parameters.TuningParameterExpression*

Stochastic search base class. Please read *Space Tutorial*.

generate(*seed=None*)
Return a randomly chosen value.

Parameters **seed** (*Optional[Any]*) – if set, it will be used to call seed() , defaults to None

Return type Any

generate_many(*n*, *seed=None*)
Generate n randomly chosen values

Parameters

- **n** (*int*) – number of random values to generate
- **seed** (*Optional[Any]*) – random seed, defaults to None

Returns a list of values

Return type List[Any]

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

class TransitionChoice(**args*)
Bases: *tune.concepts.space.parameters.Choice*

An ordered random choice of values. Please read *Space Tutorial*.

Parameters **args** (*Any*) – values to choose from

property jsondict: Dict[str, Any]
Dict representation of the expression that is json serializable

class TuningParameterExpression
Bases: object

Base class of all tuning parameter expressions

class TuningParametersTemplate(*raw*)
Bases: object

Parameter template to extract tuning parameter expressions from nested data structure

Parameters **raw** (*Dict[str, Any]*) – the dictionary of input parameters.

Note: Please use *to_template()* to initialize this class.

```
# common cases
to_template(dict(a=1, b=1))
to_template(dict(a=Rand(0, 1), b=1))

# expressions may nest in dicts or arrays
template = to_template(
    dict(a=dict(x1=Rand(0, 1), x2=Rand(3,4)), b=[Grid("a", "b")]))

assert [Rand(0, 1), Rand(3, 4), Grid("a", "b")] == template.params
assert dict(
    p0=Rand(0, 1), p1=Rand(3, 4), p2=Grid("a", "b")
) == template.params_dict
assert dict(a=1, x2=3), b=["a"]) == template.fill([1, 3, "a"])
assert dict(a=1, x2=3), b=["a"]) == template.fill_dict(
    dict(p2="a", p1=3, p0=1)
)
```

concat(*other*)
Concatenate with another template and generate a new template.

Note: The other template must not have any key existed in this template, otherwise ValueError will be raised

---

Returns the merged template

Parameters **other** (tune.concepts.space.parameters.TuningParametersTemplate) –

Return type *tune.concepts.space.parameters.TuningParametersTemplate*

static decode(*data*)
Retrieve the template from a base64 string

Parameters **data** (*str*) –

Return type *tune.concepts.space.parameters.TuningParametersTemplate*

property empty: bool
Whether the template contains any tuning expression

encode()
Convert the template to a base64 string

Return type str

fill(*params*)
Fill the original data structure with values

Parameters

- **params** (*List[Any]*) – the list of values to be filled into the original data structure, in depth-first order
- **copy** – whether to return a deeply copied paramters, defaults to False

Returns the original data structure filled with values

Return type Dict[str, Any]

fill_dict(*params*)
Fill the original data structure with dictionary of values

Parameters

- **params** (*Dict[str, Any]*) – the dictionary of values to be filled into the original data structure, keys must be p0, p1, p2, ...
- **copy** – whether to return a deeply copied paramters, defaults to False

Returns the original data structure filled with values

Return type Dict[str, Any]

property has_grid: bool
Whether the template contains grid expressions

property has_stochastic: bool
Whether the template contains stochastic expressions

property params: List[*tune.concepts.space.parameters.TuningParameterExpression*]
Get all tuning parameter expressions in depth-first order

property params_dict: Dict[str,
***tune.concepts.space.parameters.TuningParameterExpression*]**
Get all tuning parameter expressions in depth-first order, with correspondent made-up new keys p0, p1, p2, ...

product_grid()
cross product all grid parameters

Yield new templates with the grid paramters filled

Return type Iterable[*tune.concepts.space.parameters.TuningParametersTemplate*]

```
assert [dict(a=1,b=Rand(0,1)), dict(a=2,b=Rand(0,1))] ==
↪list(to_template(dict(a=Grid(1,2),b=Rand(0,1))).product_grid())
```

sample(*n*, *seed=None*)
sample all stochastic parameters

Parameters

- **n** (*int*) – number of samples, must be a positive integer
- **seed** (*Optional[Any]*) – random seed defaulting to None. It will take effect if it is not None.

Yield new templates with the grid paramters filled

Return type Iterable[*tune.concepts.space.parameters.TuningParametersTemplate*]

```
assert [dict(a=1.1,b=Grid(0,1)), dict(a=1.5,b=Grid(0,1))] ==␣
↪list(to_template(dict(a=Rand(1,2),b=Grid(0,1))).sample(2,0))
```

property simple_value: Dict[str, Any]
If the template contains no tuning expression, it's simple and it will return parameters dictionary, otherwise, `ValueError` will be raised

property template: Dict[str, Any]
The template dictionary, all tuning expressions will be replaced by `None`

to_template(*data*)
Convert an oject to `TuningParametersTemplate`

Parameters **data** (*Any*) – data object (`dict` or `TuningParametersTemplate` or `str` (encoded string))

Returns the template object

Return type *tune.concepts.space.parameters.TuningParametersTemplate*

tune.concepts.space.spaces

class Space(**args*, ***kwargs*)
Bases: `object`

Search space object

Important: Please read *Space Tutorial*.

Parameters **kwargs** (*Any*) – parameters in the search space

```
Space(a=1, b=1)  # static space
Space(a=1, b=Grid(1,2), c=Grid("a", "b"))  # grid search
Space(a=1, b=Grid(1,2), c=Rand(0, 1))  # grid search + level 2 search
Space(a=1, b=Grid(1,2), c=Rand(0, 1)).sample(10, sedd=0)  # grid + random search

# union
Space(a=1, b=Grid(2,3)) + Space(b=Rand(1,5)).sample(10)

# cross product
Space(a=1, b=Grid(2,3)) * Space(c=Rand(1,5), d=Grid("a","b"))
```

(continues on next page)

(continued from previous page)

```
# combo (grid + random + level 2)
space1 = Space(a=1, b=Grid(2,4))
space2 = Space(b=RandInt(10, 20))
space3 = Space(c=Rand(0,1)).sample(10)
space = (space1 + space2) * space3
```

```
assert Space(a=1, b=Rand(0,1)).has_stochastic
assert not Space(a=1, b=Rand(0,1)).sample(10).has_stochastic
assert not Space(a=1, b=Grid(0,1)).has_stochastic
assert not Space(a=1, b=1).has_stochastic

# get all configurations
space = Space(a=Grid(2,4), b=Rand(0,1)).sample(100)
for conf in space:
    print(conf)
all_conf = list(space)
```

property has_stochastic
Whether the space contains any *StochasticExpression*

sample(*n*, *seed=None*)
Draw random samples from the current space. Please read *Space Tutorial*.

Parameters

- **n** (*int*) – number of samples to draw
- **seed** (*Optional[Any]*) – random seed, defaults to None

Returns a new Space containing all samples

Return type *tune.concepts.space.spaces.Space*

Note:

- it only applies to *StochasticExpression*
- if *has_stochastic()* is False, then it will return the original space
- After sampling, no *StochasticExpression* will exist in the new space.

---

tune.concepts.checkpoint

class Checkpoint(*fs*)
Bases: `object`

An abstraction for tuning checkpoint

Parameters **fs** (*fs.base.FS*) – the file system

> **Attention:** Normally you don't need to create a checkpoint by yourself, please read *Checkpoint Tutorial* if you want to understand how it works.

create()
Create a new checkpoint

Return type *tune.concepts.checkpoint.NewCheckpoint*

property latest: fs.base.FS
latest checkpoint folder

Raises **AssertionError** – if there was no checkpoint

class NewCheckpoint(*checkpoint*)
Bases: object

A helper class for adding new checkpoints

Parameters **checkpoint** (tune.concepts.checkpoint.Checkpoint) – the parent checkpoint

Attention: Do not construct this class directly, please read *Checkpoint Tutorial* for details

tune.concepts.dataset

class StudyResult(*dataset*, *result*)
Bases: object

A collection of the input *TuneDataset* and the tuning result

Parameters

- **dataset** (tune.concepts.dataset.TuneDataset) – input dataset for tuning
- **result** (*fugue.workflow.workflow.WorkflowDataFrame*) – tuning result as a dataframe

Attention: Do not construct this class directly.

next_tune_dataset(*best_n=0*)
Convert the result back to a new *TuneDataset* to be used by the next steps.

Parameters **best_n** (*int*) – top n result to extract, defaults to 0 (entire result)

Returns a new dataset for tuning

Return type *tune.concepts.dataset.TuneDataset*

result(*best_n=0*)
Get the top n results sorted by *tune.concepts.flow.report.TrialReport.sort_metric()*

Parameters **best_n** (*int*) – number of result to get, defaults to 0. if <=*0* then it will return the entire result

Returns result subset

Return type fugue.workflow.workflow.WorkflowDataFrame

union_with(*other*)
Union with another result set and update itself

Parameters **other** (tune.concepts.dataset.StudyResult) – the other result dataset

Return type None

> **Note:** This method also removes duplicated reports based on *tune.concepts.flow.trial.Trial.trial_id()*. Each trial will have only the best report in the updated result

class TuneDataset(*data*, *dfs*, *keys*)

Bases: object

A Fugue WorkflowDataFrame with metadata representing all dataframes required for a tuning task.

Parameters

- **data** (*fugue.workflow.workflow.WorkflowDataFrame*) – the Fugue WorkflowDataFrame containing all required dataframes
- **dfs** (*List[str]*) – the names of the dataframes
- **keys** (*List[str]*) – the common partition keys of all dataframes

> **Attention:** Do not construct this class directly, please read *TuneDataset Tutorial* to find the right way

property data: fugue.workflow.workflow.WorkflowDataFrame

the Fugue WorkflowDataFrame containing all required dataframes

property dfs: List[str]

All dataframe names (you can also find them part of the column names of *data()*)

property keys: List[str]

Partition keys (columns) of *data()*

split(*weights*, *seed*)

Split the dataset randomly to small partitions. This is useful for some algorithms such as Hyperband, because it needs different subset to run successive halvings with different parameters.

Parameters

- **weights** (*List[float]*) – a list of numeric values. The length represents the number of splitd partitions, and the values represents the proportion of each partition
- **seed** (*Any*) – random seed for the split

Returns a list of sub-datasets

Return type List[*tune.concepts.dataset.TuneDataset*]

```
# randomly split the data to two partitions 25% and 75%
dataset.split([1, 3], seed=0)
# same because weights will be normalized
dataset.split([10, 30], seed=0)
```

class TuneDatasetBuilder(*space*, *path=''*)

Bases: object

Builder of *TuneDataset*, for details please read *TuneDataset Tutorial*

Parameters

- **space** (tune.concepts.space.spaces.Space) – searching space, see *Space Tutorial*
- **path** (*str*) – temp path to store searialized dataframe partitions , defaults to “”

add_df(*name*, *df*, *how=''*)
Add a dataframe to the dataset

Parameters

- **name** (*str*) – name of the dataframe, it will also create a __tune_df__<name> column in the dataset dataframe
- **df** (*fugue.workflow.workflow.WorkflowDataFrame*) – the dataframe to add.
- **how** (*str*) – join type, can accept semi, left_semi, anti, left_anti, inner, left_outer, right_outer, full_outer, cross

Returns the builder itself

Return type *tune.concepts.dataset.TuneDatasetBuilder*

Note: For the first dataframe you add, how should be empty. From the second dataframe you add, how must be set.

Note: If df is prepartitioned, the partition key will be used to join with the added dataframes. Read *TuneDataset Tutorial* for more details

---

add_dfs(*dfs*, *how=''*)
Add multiple dataframes with the same join type

Parameters

- **dfs** (*fugue.workflow.workflow.WorkflowDataFrames*) – dictionary like dataframe collection. The keys will be used as the dataframe names
- **how** (*str*) – join type, can accept semi, left_semi, anti, left_anti, inner, left_outer, right_outer, full_outer, cross

Returns the builder itself

Return type *tune.concepts.dataset.TuneDatasetBuilder*

build(*wf*, *batch_size=1*, *shuffle=True*, *trial_metadata=None*)
Build *TuneDataset*, for details please read *TuneDataset Tutorial*

Parameters

- **wf** (*fugue.workflow.workflow.FugueWorkflow*) – the workflow associated with the dataset
- **batch_size** (*int*) – how many configurations as a batch, defaults to 1
- **shuffle** (*bool*) – whether to shuffle the entire dataset, defaults to True. This is to make the tuning process more even, it will look better. It should have slight benefit on speed, no effect on result.
- **trial_metadata** (*Optional[Dict[str, Any]]*) – metadata to pass to each *Trial*, defaults to None

Returns the dataset for tuning

Return type *tune.concepts.dataset.TuneDataset*

3.1.3 tune.iterative

tune.iterative.asha

class **ASHAJudge**(*schedule*, *always_checkpoint=False*, *study_early_stop=None*, *trial_early_stop=None*, *monitor=None*)

Bases: *tune.concepts.flow.judge.TrialJudge*

Parameters

- **schedule** (*List[Tuple[float, int]]*) –
- **always_checkpoint** (*bool*) –
- **study_early_stop** (*Optional[Callable[[List[Any], List[tune.iterative.asha.RungHeap]], bool]]*) –
- **trial_early_stop** (*Optional[Callable[[tune.concepts.flow.report.TrialReport, List[tune.concepts.flow.report.TrialReport], List[tune.iterative.asha.RungHeap]], bool]]*) –
- **monitor** (*Optional[tune.concepts.flow.judge.Monitor]*) –

property always_checkpoint: bool

can_accept(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

get_budget(*trial*, *rung*)

Parameters

- **trial** (tune.concepts.flow.trial.Trial) –
- **rung** (*int*) –

Return type float

judge(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type *tune.concepts.flow.judge.TrialDecision*

property schedule: List[Tuple[float, int]]

class **RungHeap**(*n*)

Bases: object

Parameters **n** (*int*) –

property best: float

property bests: List[float]

property capacity: int

property full: bool

push(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type bool

values()

Return type Iterable[*tune.concepts.flow.report.TrialReport*]

tune.iterative.objective

class IterativeObjectiveFunc

Bases: object

copy()

Return type *tune.iterative.objective.IterativeObjectiveFunc*

property current_trial: *tune.concepts.flow.trial.Trial*

finalize()

Return type None

generate_sort_metric(*value*)

Parameters **value** (*float*) –

Return type float

initialize()

Return type None

load_checkpoint(*fs*)

Parameters **fs** (*fs.base.FS*) –

Return type None

run(*trial*, *judge*, *checkpoint_basedir_fs*)

Parameters

- **trial** (tune.concepts.flow.trial.Trial) –
- **judge** (tune.concepts.flow.judge.TrialJudge) –
- **checkpoint_basedir_fs** (*fs.base.FS*) –

Return type None

run_single_iteration()

Return type *tune.concepts.flow.report.TrialReport*

run_single_rung(*budget*)

Parameters **budget** (*float*) –

Return type *tune.concepts.flow.report.TrialReport*

property rung: int

save_checkpoint(*fs*)

Parameters **fs** (*fs.base.FS*) –

Return type None

validate_iterative_objective(*func*, *trial*, *budgets*, *validator*, *continuous=False*, *checkpoint_path=''*, *monitor=None*)

Parameters

- **func** (tune.iterative.objective.IterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **budgets** (*List[float]*) –
- **validator** (*Callable[[List[tune.concepts.flow.report.TrialReport]], None]*) –
- **continuous** (*bool*) –
- **checkpoint_path** (*str*) –
- **monitor** (*Optional[tune.concepts.flow.judge.Monitor]*) –

Return type None

tune.iterative.sha

tune.iterative.study

class IterativeStudy(*objective*, *checkpoint_path*)

Bases: object

Parameters

- **objective** (tune.iterative.objective.IterativeObjectiveFunc) –
- **checkpoint_path** (*str*) –

optimize(*dataset*, *judge*)

Parameters

- **dataset** (tune.concepts.dataset.TuneDataset) –
- **judge** (tune.concepts.flow.judge.TrialJudge) –

Return type *tune.concepts.dataset.StudyResult*

3.1.4 tune.noniterative

tune.noniterative.convert

noniterative_objective(*func=None*, *min_better=True*)

Parameters

- **func** (*Optional[Callable]*) –
- **min_better** (*bool*) –

Return type Callable[[Any], *tune.noniterative.objective.NonIterativeObjectiveFunc*]

to_noniterative_objective(*obj*, *min_better=True*, *global_vars=None*, *local_vars=None*)

Parameters

- **obj** (*Any*) –
- **min_better** (*bool*) –
- **global_vars** (*Optional[Dict[str, Any]]*) –
- **local_vars** (*Optional[Dict[str, Any]]*) –

Return type *tune.noniterative.objective.NonIterativeObjectiveFunc*

tune.noniterative.objective

class NonIterativeObjectiveFunc

Bases: object

generate_sort_metric(*value*)

Parameters **value** (*float*) –

Return type float

run(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type *tune.concepts.flow.report.TrialReport*

safe_run(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type *tune.concepts.flow.report.TrialReport*

class NonIterativeObjectiveLocalOptimizer

Bases: object

property distributable: bool

run(*func*, *trial*, *logger*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **logger** (*Any*) –

Return type *tune.concepts.flow.report.TrialReport*

run_monitored_process(*func*, *trial*, *stop_checker*, *logger*, *interval='60sec'*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **stop_checker** (*Callable[[], bool]*) –
- **logger** (*Any*) –
- **interval** (*Any*) –

Return type *tune.concepts.flow.report.TrialReport*

validate_noniterative_objective(*func*, *trial*, *validator*, *optimizer=None*, *logger=None*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **validator** (*Callable[[*tune.concepts.flow.report.TrialReport*], None]*) –
- **optimizer** (*Optional[*tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer*]*) –
- **logger** (*Optional[Any]*) –

Return type None

tune.noniterative.stopper

class NonIterativeStopper(*log_best_only=False*)

Bases: *tune.concepts.flow.judge.TrialJudge*

Parameters **log_best_only** (*bool*) –

can_accept(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

get_reports(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type List[*tune.concepts.flow.report.TrialReport*]

judge(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type *tune.concepts.flow.judge.TrialDecision*

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type bool

should_stop(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

property updated: bool

class NonIterativeStopperCombiner(*left*, *right*, *is_and*)

Bases: *tune.noniterative.stopper.NonIterativeStopper*

Parameters

- **left** (tune.noniterative.stopper.NonIterativeStopper) –
- **right** (tune.noniterative.stopper.NonIterativeStopper) –
- **is_and** (*bool*) –

get_reports(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type List[*tune.concepts.flow.report.TrialReport*]

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type bool

should_stop(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

class SimpleNonIterativeStopper(*partition_should_stop*, *log_best_only=False*)

Bases: *tune.noniterative.stopper.NonIterativeStopper*

Parameters

- **partition_should_stop** (*Callable[[*tune.concepts.flow.report.TrialReport*, bool, List[*tune.concepts.flow.report.TrialReport*]], bool]*) –
- **log_best_only** (*bool*) –

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type bool

should_stop(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type bool

class TrialReportCollection(*new_best_only=False*)

Bases: *tune.concepts.flow.report.TrialReportLogger*

Parameters **new_best_only** (*bool*) –

log(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type None

property reports: List[*tune.concepts.flow.report.TrialReport*]

n_samples(*n*)

Parameters **n** (*int*) –

Return type *tune.noniterative.stopper.SimpleNonIterativeStopper*

n_updates(*n*)

Parameters **n** (*int*) –

Return type *tune.noniterative.stopper.SimpleNonIterativeStopper*

no_update_period(*period*)

Parameters **period** (*Any*) –

Return type *tune.noniterative.stopper.SimpleNonIterativeStopper*

small_improvement(*threshold*, *updates*)

Parameters

- **threshold** (*float*) –
- **updates** (*int*) –

Return type *tune.noniterative.stopper.SimpleNonIterativeStopper*

tune.noniterative.study

class NonIterativeStudy(*objective*, *optimizer*)
Bases: object

Parameters

- **objective** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **optimizer** (tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer) –

optimize(*dataset*, *distributed=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *logger=None*)

Parameters

- **dataset** (tune.concepts.dataset.TuneDataset) –
- **distributed** (*Optional[bool]*) –
- **monitor** (*Optional[*tune.concepts.flow.judge.Monitor*]*) –
- **stopper** (*Optional[*tune.noniterative.stopper.NonIterativeStopper*]*) –
- **stop_check_interval** (*Optional[Any]*) –
- **logger** (*Optional[Any]*) –

Return type *tune.concepts.dataset.StudyResult*

3.1.5 tune.constants

3.1.6 tune.exceptions

exception TuneCompileError
Bases: fugue.exceptions.FugueWorkflowCompileError

exception TuneInterrupted
Bases: *tune.exceptions.TuneRuntimeError*

exception TuneRuntimeError
Bases: fugue.exceptions.FugueWorkflowRuntimeError

3.2 tune_hyperopt

3.2.1 tune_hyperopt.optimizer

class HyperoptLocalOptimizer(*max_iter*, *seed=0*, *kwargs_func=None*)
Bases: *tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer*

Parameters

- **max_iter** (*int*) –
- **seed** (*int*) –

- **kwargs_func** (*Optional[Callable[[tune.noniterative.objective.NonIterativeObjectiveFunc, tune.concepts.flow.trial.Trial], Dict[str, Any]]]*) –

run(*func*, *trial*, *logger*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **logger** (*Any*) –

Return type *tune.concepts.flow.report.TrialReport*

3.3 tune_optuna

3.3.1 tune_optuna.optimizer

class OptunaLocalOptimizer(*max_iter*, *create_study=None*)

Bases: *tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer*

Parameters

- **max_iter** (*int*) –
- **create_study** (*Optional[Callable[[], optuna.study.study.Study]]*) –

run(*func*, *trial*, *logger*)

Parameters

- **func** (tune.noniterative.objective.NonIterativeObjectiveFunc) –
- **trial** (tune.concepts.flow.trial.Trial) –
- **logger** (*Any*) –

Return type *tune.concepts.flow.report.TrialReport*

3.4 tune_sklearn

3.4.1 tune_sklearn.objective

class SKCVObjective(*scoring*, *cv=5*, *feature_prefix=''*, *label_col='label'*, *checkpoint_path=None*)

Bases: *tune_sklearn.objective.SKObjective*

Parameters

- **scoring** (*Any*) –
- **cv** (*int*) –
- **feature_prefix** (*str*) –
- **label_col** (*str*) –

- **checkpoint_path** (*Optional[str]*) –

Return type None

run(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type *tune.concepts.flow.report.TrialReport*

class SKObjective(*scoring*, *feature_prefix=''*, *label_col='label'*, *checkpoint_path=None*)

Bases: *tune.noniterative.objective.NonIterativeObjectiveFunc*

Parameters

- **scoring** (*Any*) –
- **feature_prefix** (*str*) –
- **label_col** (*str*) –
- **checkpoint_path** (*Optional[str]*) –

Return type None

generate_sort_metric(*value*)

Parameters **value** (*float*) –

Return type float

run(*trial*)

Parameters **trial** (tune.concepts.flow.trial.Trial) –

Return type *tune.concepts.flow.report.TrialReport*

3.4.2 tune_sklearn.suggest

suggest_sk_models(*space*, *train_df*, *test_df*, *scoring*, *temp_path=''*, *feature_prefix=''*, *label_col='label'*, *save_model=False*, *partition_keys=None*, *top_n=1*, *local_optimizer=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **train_df** (*Any*) –
- **test_df** (*Any*) –
- **scoring** (*str*) –
- **temp_path** (*str*) –
- **feature_prefix** (*str*) –
- **label_col** (*str*) –
- **save_model** (*bool*) –
- **partition_keys** (*Optional[List[str]]*) –

- **top_n** (*int*) –
- **local_optimizer** (*Optional[tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer]*) –
- **monitor** (*Optional[Any]*) –
- **stopper** (*Optional[Any]*) –
- **stop_check_interval** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_sk_models_by_cv(*space*, *train_df*, *scoring*, *cv=5*, *temp_path=''*, *feature_prefix=''*, *label_col='label'*, *save_model=False*, *partition_keys=None*, *top_n=1*, *local_optimizer=None*, *monitor=None*, *stopper=None*, *stop_check_interval=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (*tune.concepts.space.spaces.Space*) –
- **train_df** (*Any*) –
- **scoring** (*str*) –
- **cv** (*int*) –
- **temp_path** (*str*) –
- **feature_prefix** (*str*) –
- **label_col** (*str*) –
- **save_model** (*bool*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **local_optimizer** (*Optional[tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer]*) –
- **monitor** (*Optional[Any]*) –
- **stopper** (*Optional[Any]*) –
- **stop_check_interval** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

3.4.3 tune_sklearn.utils

sk_space(*model*, ***params*)

Parameters

- **model** (*str*) –
- **params** (*Dict[str, Any]*) –

Return type *tune.concepts.space.spaces.Space*

to_sk_model(*obj*)

Parameters **obj** (*Any*) –

Return type Type

to_sk_model_expr(*model*)

Parameters **model** (*Any*) –

Return type Any

3.5 tune_tensorflow

3.5.1 tune_tensorflow.objective

class KerasObjective(*type_dict*)

Bases: *tune.iterative.objective.IterativeObjectiveFunc*

Parameters **type_dict** (*Dict[str, Type[*tune_tensorflow.spec.KerasTrainingSpec*]]*) –

Return type None

copy()

Return type *tune_tensorflow.objective.KerasObjective*

finalize()

Return type None

generate_sort_metric(*value*)

Parameters **value** (*float*) –

Return type float

initialize()

Return type None

load_checkpoint(*fs*)

Parameters **fs** (*fs.base.FS*) –

Return type None

property model: keras.engine.training.Model

run_single_rung(*budget*)

Parameters **budget** (*float*) –

Return type *tune.concepts.flow.report.TrialReport*

save_checkpoint(*fs*)

Parameters **fs** (*fs.base.FS*) –

Return type None

property spec: *tune_tensorflow.spec.KerasTrainingSpec*

3.5.2 tune_tensorflow.spec

class KerasTrainingSpec(*params*, *dfs*)

Bases: object

Parameters

- **params** (*Any*) –
- **dfs** (*Dict[str, Any]*) –

compile_model(***add_kwargs*)

Parameters **add_kwargs** (*Any*) –

Return type keras.engine.training.Model

compute_sort_metric(***add_kwargs*)

Parameters **add_kwargs** (*Any*) –

Return type float

property dfs: Dict[str, Any]

finalize()

Return type None

fit(***add_kwargs*)

Parameters **add_kwargs** (*Any*) –

Return type keras.callbacks.History

generate_sort_metric(*metric*)

Parameters **metric** (*float*) –

Return type float

`get_compile_params()`

Return type Dict[str, Any]

`get_fit_metric(history)`

Parameters **history** (*keras.callbacks.History*) –

Return type float

`get_fit_params()`

Return type Tuple[List[Any], Dict[str, Any]]

`get_model()`

Return type keras.engine.training.Model

`load_checkpoint(fs, model)`

Parameters

- **fs** (*fs.base.FS*) –
- **model** (*keras.engine.training.Model*) –

Return type None

`property params: tune.concepts.space.parameters.TuningParametersTemplate`

`save_checkpoint(fs, model)`

Parameters

- **fs** (*fs.base.FS*) –
- **model** (*keras.engine.training.Model*) –

Return type None

3.5.3 tune_tensorflow.suggest

`suggest_keras_models_by_continuous_asha(space, plan, train_df=None, temp_path='', partition_keys=None, top_n=1, monitor=None, execution_engine=None, execution_engine_conf=None)`

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –

- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_keras_models_by_hyperband(*space*, *plans*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **plans** (*List[List[Tuple[float, int]]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

suggest_keras_models_by_sha(*space*, *plan*, *train_df=None*, *temp_path=''*, *partition_keys=None*, *top_n=1*, *monitor=None*, *distributed=None*, *execution_engine=None*, *execution_engine_conf=None*)

Parameters

- **space** (tune.concepts.space.spaces.Space) –
- **plan** (*List[Tuple[float, int]]*) –
- **train_df** (*Optional[Any]*) –
- **temp_path** (*str*) –
- **partition_keys** (*Optional[List[str]]*) –
- **top_n** (*int*) –
- **monitor** (*Optional[Any]*) –
- **distributed** (*Optional[bool]*) –
- **execution_engine** (*Optional[Any]*) –
- **execution_engine_conf** (*Optional[Any]*) –

Return type List[*tune.concepts.flow.report.TrialReport*]

3.5.4 tune_tensorflow.utils

extract_keras_spec(*params*, *type_dict*)

Parameters

- **params** (tune.concepts.space.parameters.TuningParametersTemplate) –
- **type_dict** (*Dict[str, Any]*) –

Return type Type[*tune_tensorflow.spec.KerasTrainingSpec*]

keras_space(*model*, ***params*)

Parameters

- **model** (*Any*) –
- **params** (*Any*) –

Return type *tune.concepts.space.spaces.Space*

to_keras_spec(*obj*)

Parameters **obj** (*Any*) –

Return type Type[*tune_tensorflow.spec.KerasTrainingSpec*]

to_keras_spec_expr(*spec*)

Parameters **spec** (*Any*) –

Return type str

3.6 tune_notebook

3.6.1 tune_notebook.monitors

class NotebookSimpleChart(*interval='1sec'*, *best_only=True*, *always_update=False*)

Bases: *tune.concepts.flow.judge.Monitor*

Parameters

- **interval** (*Any*) –
- **best_only** (*bool*) –
- **always_update** (*bool*) –

finalize()

Return type None

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type None

plot(*df*)

Parameters **df** (*pandas.core.frame.DataFrame*) –

Return type None

class NotebookSimpleHist(*interval='1sec'*)
Bases: *tune_notebook.monitors.NotebookSimpleChart*

Parameters **interval** (*Any*) –

plot(*df*)

Parameters **df** (*pandas.core.frame.DataFrame*) –

Return type None

class NotebookSimpleRungs(*interval='1sec'*)
Bases: *tune_notebook.monitors.NotebookSimpleChart*

Parameters **interval** (*Any*) –

plot(*df*)

Parameters **df** (*pandas.core.frame.DataFrame*) –

Return type None

class NotebookSimpleTimeSeries(*interval='1sec'*)
Bases: *tune_notebook.monitors.NotebookSimpleChart*

Parameters **interval** (*Any*) –

plot(*df*)

Parameters **df** (*pandas.core.frame.DataFrame*) –

Return type None

class PrintBest
Bases: *tune.concepts.flow.judge.Monitor*

on_report(*report*)

Parameters **report** (tune.concepts.flow.report.TrialReport) –

Return type None

3.7 tune_test

3.7.1 tune_test.local_optmizer

class NonIterativeObjectiveLocalOptimizerTests
Bases: object

DataFrame level general test suite. All new DataFrame types should pass this test suite.

class Tests(*methodName='runTest'*)
Bases: unittest.case.TestCase

make_optimizer(***kwargs*)

Parameters **kwargs** (*Any*) –
Return type *tune.noniterative.objective.NonIterativeObjectiveLocalOptimizer*

test_choice()

test_optimization()

test_optimization_dummy()

test_optimization_nested_param()

test_rand()

test_randint()

test_transition_choice()

CHAPTER

FOUR

SHORT TUTORIALS

4.1 Search Space

THIS IS THE MOST IMPORTANT CONCEPT OF TUNE, MUST READ

Tune defines its own searching space concept and different expressions. It inherits the Fugue philosophy: one expression for all frameworks. For the underlying optimizers (e.g. HyperOpt, Optuna), tune unifies the behaviors. For example `Rand(1.0, 5.0, q=1.5)` will uniformly search on `[1.0 , 2.5, 4.0]` no matter you use HyperOpt or Optuna as the underlying optimizer.

In Tune, spaces are predefined before search, it is opposite to Optuna where you get variables inside objectives during runtime. In this way, your space definition is totally separated from objective definition, and your objectives may be just simple python functions independent from Tune.

[1]:
```
from tune import Space, Grid, Rand, RandInt, Choice
import pandas as pd
```

4.1.1 Simple Cases

The simplest cases are spaces with only static variables. So the spaces will always generate single configuration.

[2]:
```
space = Space(a=1, b=1)
print(list(space))
```

```
[{'a': 1, 'b': 1}]
```

4.1.2 Grid Search

You can replace the static variables to `Grid` expression. We will cross product all grid expressions in the space, so you see in the second example, it generates 6 configurations.

[3]:
```
print(list(Space(a=1, b=Grid("a","b"))))
print(list(Space(a=Grid(1,2), b=Grid("x","y","z"))))
```

```
[{'a': 1, 'b': 'a'}, {'a': 1, 'b': 'b'}]
[{'a': 1, 'b': 'x'}, {'a': 1, 'b': 'y'}, {'a': 1, 'b': 'z'}, {'a': 2, 'b': 'x'}, {'a': 2,
↪ 'b': 'y'}, {'a': 2, 'b': 'z'}]
```

4.1.3 Random Expressions

Random search requires `.sample` method after you define the original space to specify how many random combinations you want to draw from the expression.

Choice

`Choice` refers to discrete **unordered** set of values. So `Choice(1, 2, 3)` is equivalent to `Choide(2, 1, 3)`. When you do random sampling from `Choice`, every value has equal chance. When you do advanced search such as Bayesian Optimization, it also assumes no relation between values.

[4]:
```
space = Space(a=1, b=Choice("aa", "bb", "cc")).sample(2, seed=1)
print(list(space))
```

```
[{'a': 1, 'b': 'bb'}, {'a': 1, 'b': 'aa'}]
```

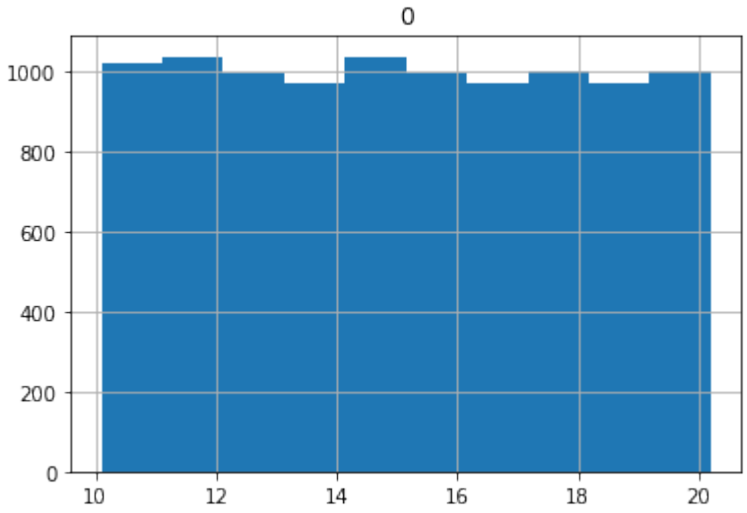
Rand

`Rand` is the most common expression for a variable. It refers to sampling from a range of value.

Rand(low, high)

uniformly search between `[low, high)`

[5]:
```
samples = Rand(10.1, 20.2).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```



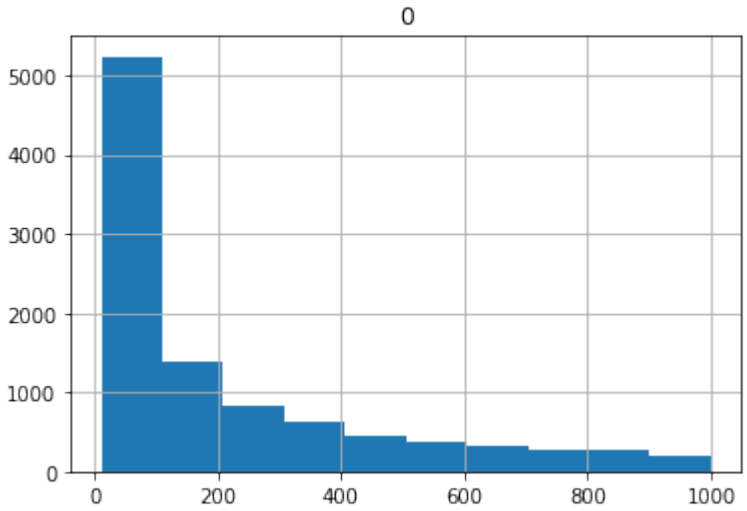
Rand(low, high, log=True)

search in the log space, but still in `[low, high)` so the smaller values get higher chance to be selected.

For log space searching, low must be greater or equal to 1.

The algorithm: `exp(uniform(log(low), log(high)))`

[6]:
```
samples = Rand(10.1, 1000, log=True).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```



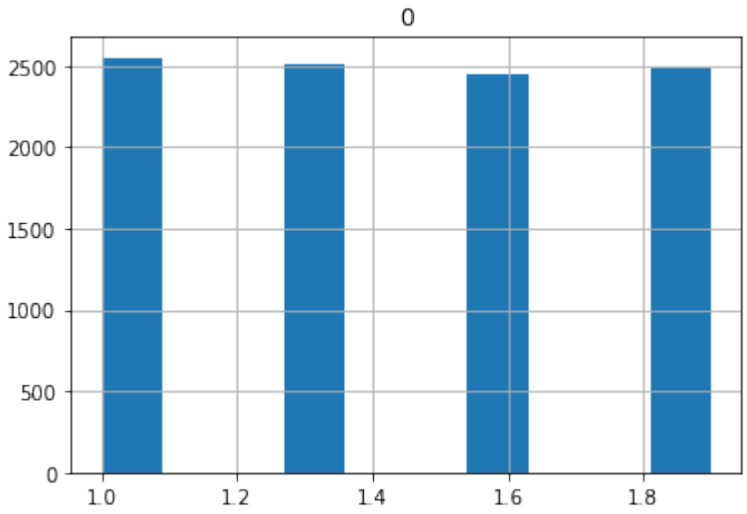
Rand(low, high, q, include_high)

uniformly search between `low` and `high` with step `q`. `include_high` (default `True`) indicates whether the high value can be a candidate.

[7]:
```
print(Rand(-1.0,4.0,q=2.5).generate_many(10, seed=0))
print(Rand(-1.0,4.0,q=2.5,include_high=False).generate_many(10, seed=0))

samples = Rand(1.0,2.0,q=0.3).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```

```
[1.5, 4.0, 1.5, 1.5, 1.5, 1.5, 1.5, 4.0, 4.0, 1.5]
[1.5, 1.5, 1.5, 1.5, -1.0, 1.5, -1.0, 1.5, 1.5, -1.0]
```

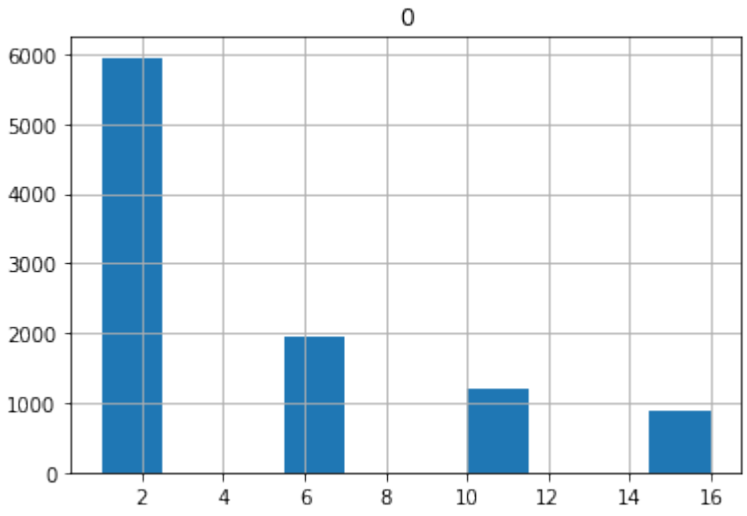


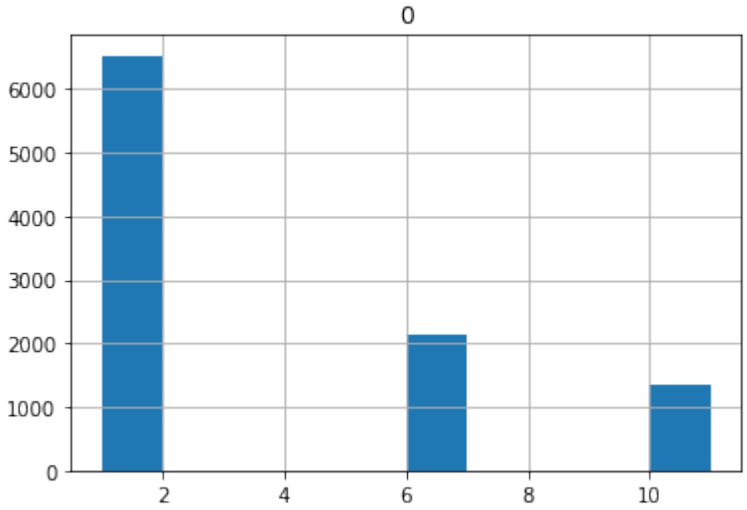
Rand(low, high, q, include_high, log=True)

search between `low` and `high` with step `q` in log space. `include_high` (default `True`) indicates whether the high value can be a candidate.

[8]:
```
samples = Rand(1.0,16.0,q=5, log=True).generate_many(10000, seed=0)
pd.DataFrame(samples).hist()

samples = Rand(1.0,16.0,q=5, log=True, include_high=False).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```





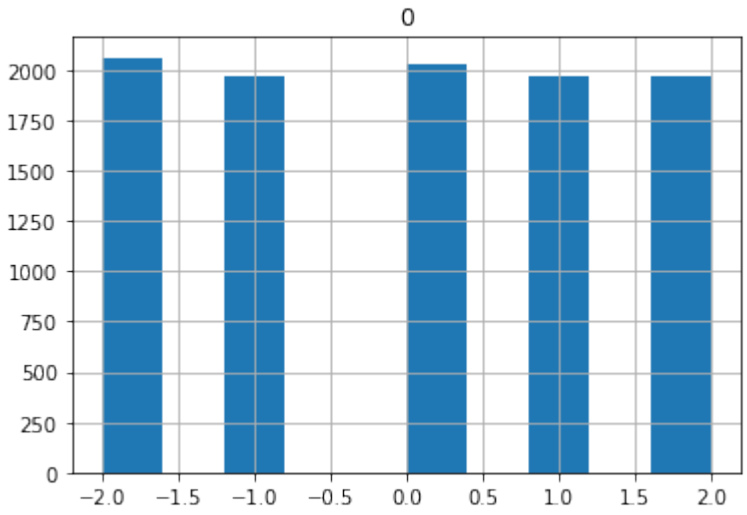
RandInt

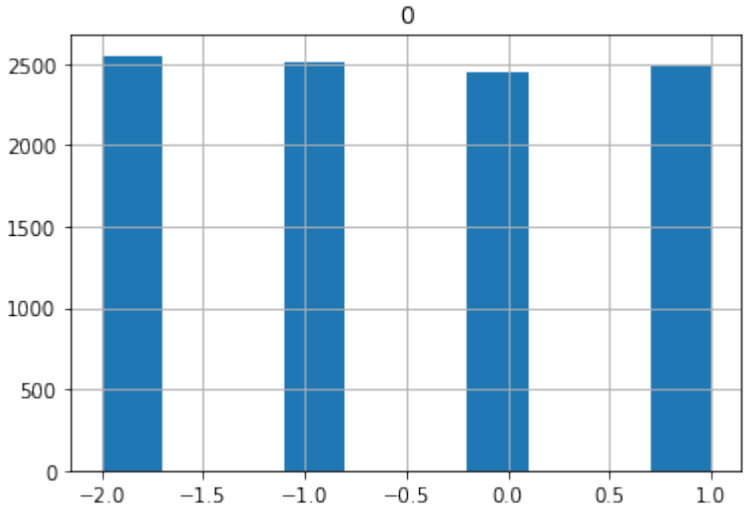
`RandInt` can be considered as a special case of `Rand` where the `low`, `high` and `q` are all integers

RandInt(low, high, include_high)

[9]:
```
samples = RandInt(-2,2).generate_many(10000, seed=0)
pd.DataFrame(samples).hist()

samples = RandInt(-2,2,include_high=False).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```



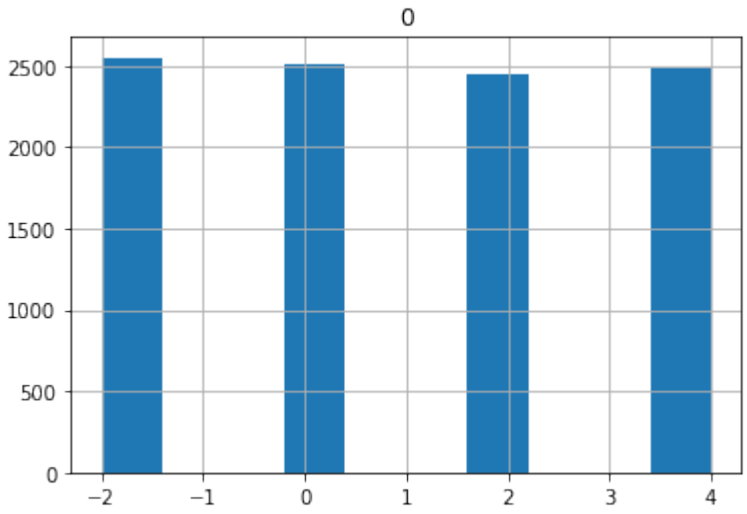


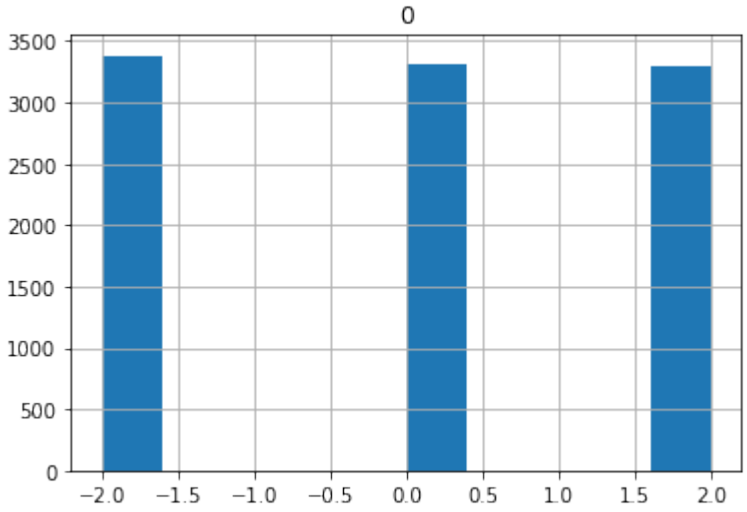
RandInt(low, high, include_high, q)

Search starting from `low` with step `q` to `high`

[10]:
```
samples = RandInt(-2,4,q=2).generate_many(10000, seed=0)
pd.DataFrame(samples).hist()

samples = RandInt(-2,4,include_high=False,q=2).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```





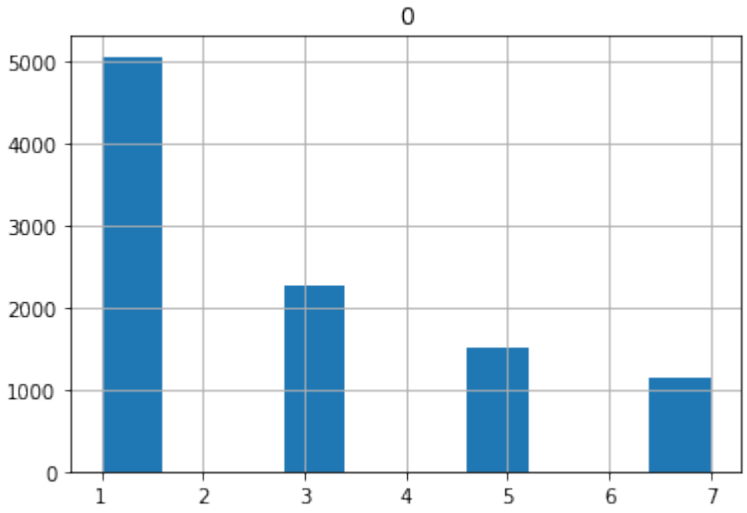
RandInt(low, high, include_high, q, log)

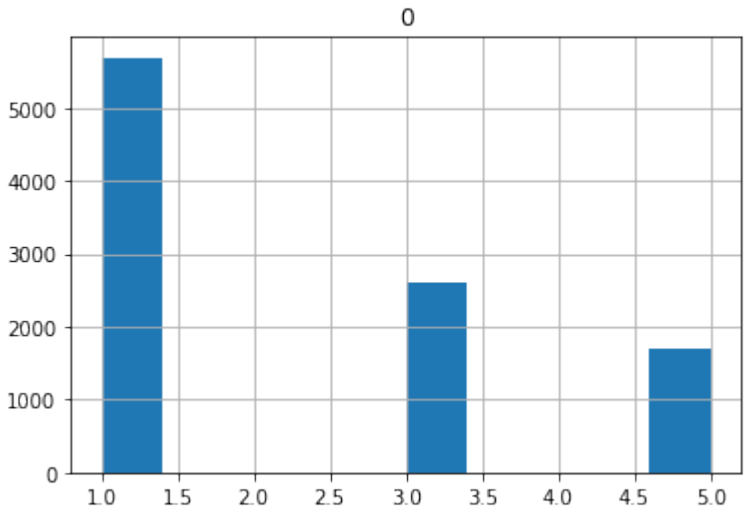
Search starting from `low` with step `q` to `high`. The difference is it's in log space, so lower values get higher chance.

Also for log searching space, `low` must be >=1

[11]:
```
samples = RandInt(1,7,q=2,log=True).generate_many(10000, seed=0)
pd.DataFrame(samples).hist()

samples = RandInt(1,7,include_high=False,q=2,log=True).generate_many(10000, seed=0)
pd.DataFrame(samples).hist();
```





4.1.4 Random Search

In Tune, you have two options to search on random expressions

As Level 1 Search

Level 1 means before execution. So given a combination of random expressions, we draw certain number of parameter combinations before execution. So the system will only deal with static parameters during runtime.

Grid search is also Level 1 search, and Level 1 search determines max parallelism. To also treat random expressions as Level 1, we must use `.sample`

[12]:
```
space = Space(a=Rand(0,1), b=Choice("x", "y")).sample(10, seed=0)
list(space)
```

[12]:
```
[{'a': 0.5488135039273248, 'b': 'x'},
 {'a': 0.7151893663724195, 'b': 'y'},
 {'a': 0.6027633760716439, 'b': 'y'},
 {'a': 0.5448831829968969, 'b': 'x'},
 {'a': 0.4236547993389047, 'b': 'x'},
 {'a': 0.6458941130666561, 'b': 'y'},
 {'a': 0.4375872112626925, 'b': 'y'},
 {'a': 0.8917730007820798, 'b': 'y'},
 {'a': 0.9636627605010293, 'b': 'y'},
 {'a': 0.3834415188257777, 'b': 'x'}]
```

If in space, you have both grid and random expressions, `.sample` will only apply to random samples, and then cross product with all grid combinations

[13]:
```
space = Space(a=Grid(0,1), b=Rand(0,1), c=Grid("a", "b"), d=Rand(0,1)).sample(3, seed=1)
list(space)  # 2*2 *3 configs
```

[13]:
```
[{'a': 0, 'b': 0.417022004702574, 'c': 'a', 'd': 0.30233257263183977},
 {'a': 0, 'b': 0.417022004702574, 'c': 'b', 'd': 0.30233257263183977},
 {'a': 1, 'b': 0.417022004702574, 'c': 'a', 'd': 0.30233257263183977},
```

(continues on next page)

(continued from previous page)

```
 {'a': 1, 'b': 0.417022004702574, 'c': 'b', 'd': 0.30233257263183977},
 {'a': 0, 'b': 0.7203244934421581, 'c': 'a', 'd': 0.14675589081711304},
 {'a': 0, 'b': 0.7203244934421581, 'c': 'b', 'd': 0.14675589081711304},
 {'a': 1, 'b': 0.7203244934421581, 'c': 'a', 'd': 0.14675589081711304},
 {'a': 1, 'b': 0.7203244934421581, 'c': 'b', 'd': 0.14675589081711304},
 {'a': 0, 'b': 0.00011437481734488664, 'c': 'a', 'd': 0.0923385947687978},
 {'a': 0, 'b': 0.00011437481734488664, 'c': 'b', 'd': 0.0923385947687978},
 {'a': 1, 'b': 0.00011437481734488664, 'c': 'a', 'd': 0.0923385947687978},
 {'a': 1, 'b': 0.00011437481734488664, 'c': 'b', 'd': 0.0923385947687978}]
```

As Level 2 Search

Level 2 search happens during runtime, and bases on each level 1 search candidate. A common scenario is that we want to do grid search on one parameter, and do Bayesian Optimization on another parameter. Then we can parallelize on the choices of the first parameter and do sequential Bayesian Optimization on the second parameter.

We will use 3rd party solutions for Level 2 search, such as HyperOpt and Optuna. To pass random expression to Level 2, we simply don't use `.sample`

[14]:
```
space = Space(a=Grid(0,1), b=Rand(0,1), c=Grid("a", "b"), d=Rand(0,1))
list(space)  # 2*2 configs, each of the config still contains the Rand expression
```

[14]:
```
[{'a': 0, 'b': Rand(low=0, high=1, q=None, log=False, include_high=True), 'c': 'a', 'd':␣
↪Rand(low=0, high=1, q=None, log=False, include_high=True)},
 {'a': 0, 'b': Rand(low=0, high=1, q=None, log=False, include_high=True), 'c': 'b', 'd':␣
↪Rand(low=0, high=1, q=None, log=False, include_high=True)},
 {'a': 1, 'b': Rand(low=0, high=1, q=None, log=False, include_high=True), 'c': 'a', 'd':␣
↪Rand(low=0, high=1, q=None, log=False, include_high=True)},
 {'a': 1, 'b': Rand(low=0, high=1, q=None, log=False, include_high=True), 'c': 'b', 'd':␣
↪Rand(low=0, high=1, q=None, log=False, include_high=True)}]
```

4.1.5 Space Operations, Conditional Search and Hybrid Search

Almost all popular tuning frameworks support conditional search. Tune approaches conditional search in a totally different way.

Instead using if-else at runtime or using nested dictionaries to represent conditions, we introduce space operations:

[15]:
```
space1 = Space(a=1, b=Grid(2,3))
space2 = Space(c=Grid("a","b"))

union_space = space1 + space2
print(list(union_space))

product_space = space1 * space2
print(list(product_space))
```

```
[{'a': 1, 'b': 2}, {'a': 1, 'b': 3}, {'c': 'a'}, {'c': 'b'}]
[{'a': 1, 'b': 2, 'c': 'a'}, {'a': 1, 'b': 2, 'c': 'b'}, {'a': 1, 'b': 3, 'c': 'a'}, {'a
↪': 1, 'b': 3, 'c': 'b'}]
```

Operator + will **union** the configurations from two spaces, it can solve most of the conditional search problems

Operator * will **cross product** the configurations from two spaces, it can solve most of the hybrid search problems

Conditional Search

[16]:
```
space1 = Space(model="LogisticRegression")
space2 = Space(model="RandomForestClassifier", max_depth=Grid(3,4))
space3 = Space(model="XGBClassifier", n_estimators=Grid(10,100,1000))

sweep = sum([space1, space2, space3])  # sum is another way to union
list(sweep)
```

[16]:
```
[{'model': 'LogisticRegression'},
 {'model': 'RandomForestClassifier', 'max_depth': 3},
 {'model': 'RandomForestClassifier', 'max_depth': 4},
 {'model': 'XGBClassifier', 'n_estimators': 10},
 {'model': 'XGBClassifier', 'n_estimators': 100},
 {'model': 'XGBClassifier', 'n_estimators': 1000}]
```

All 3 models have a parameter `random_state`, we want also want to do a grid search on it for every model. We just use *

[17]:
```
sweep_with_random_state = sweep * Space(random_state=Grid(0,1))
list(sweep_with_random_state)
```

[17]:
```
[{'model': 'LogisticRegression', 'random_state': 0},
 {'model': 'LogisticRegression', 'random_state': 1},
 {'model': 'RandomForestClassifier', 'max_depth': 3, 'random_state': 0},
 {'model': 'RandomForestClassifier', 'max_depth': 3, 'random_state': 1},
 {'model': 'RandomForestClassifier', 'max_depth': 4, 'random_state': 0},
 {'model': 'RandomForestClassifier', 'max_depth': 4, 'random_state': 1},
 {'model': 'XGBClassifier', 'n_estimators': 10, 'random_state': 0},
 {'model': 'XGBClassifier', 'n_estimators': 10, 'random_state': 1},
 {'model': 'XGBClassifier', 'n_estimators': 100, 'random_state': 0},
 {'model': 'XGBClassifier', 'n_estimators': 100, 'random_state': 1},
 {'model': 'XGBClassifier', 'n_estimators': 1000, 'random_state': 0},
 {'model': 'XGBClassifier', 'n_estimators': 1000, 'random_state': 1}]
```

Hybrid Search (Grid + Random + Bayesian Optimization)

For `XGBClassifier`, we want to do a hybrid search: grid search on `random_state`, random search on `n_estimators` and Level 2 (Bayesian Optimization) search on `learning_rate`

[18]:
```
xgb = Space(model="XGBClassifier", learning_rate=Rand(0,1), random_state=Grid(0,1)) *␣
↪Space(n_estimators=RandInt(10,1000)).sample(3, seed=0)
list(xgb)
```

[18]:
```
[{'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 0, 'n_estimators': 553},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 0, 'n_estimators': 718},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 0, 'n_estimators': 607},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 1, 'n_estimators': 553},
```

(continues on next page)

(continued from previous page)

```
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 1, 'n_estimators': 718},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 1, 'n_estimators': 607}]
```

Hybrid search and conditional search can also be used together

[19]:
```
list(Space(model="LogisticRegression")+xgb)
```

[19]:
```
[{'model': 'LogisticRegression'},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 0, 'n_estimators': 553},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 0, 'n_estimators': 718},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 0, 'n_estimators': 607},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 1, 'n_estimators': 553},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 1, 'n_estimators': 718},
 {'model': 'XGBClassifier', 'learning_rate': Rand(low=0, high=1, q=None, log=False,␣
↪include_high=True), 'random_state': 1, 'n_estimators': 607}]
```

[ ]:

4.2 Non-Iterative Tuning Guide

4.2.1 Hello World

Let's do a hybrid parameter tuning with grid search + random search, and run it distributedly

[1]:
```
def objective(a, b) -> float:
    return a**2 + b**2
```

[2]:
```
from tune import Space, Grid, Rand, RandInt, Choice

space = Space(a=Grid(-1,0,1), b=Rand(-10,10)).sample(100, seed=0)
```

[4]:
```
from tune import suggest_for_noniterative_objective

result = suggest_for_noniterative_objective(objective, space, top_n=1)[0]
print(result.sort_metric, result)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
0.1909396653178624 {'trial': {'trial_id': '58c94f4f-011e-53da-a85b-7e696ced6600', 'params
↪': {'a': 0, 'b': 0.43696643500143395}, 'metadata': {}, 'keys': []}, 'metric': 0.
↪1909396653178624, 'params': {'a': 0, 'b': 0.43696643500143395}, 'metadata': {}, 'cost':
↪ 1.0, 'rung': 0, 'sort_metric': 0.1909396653178624, 'log_time': datetime.datetime(2021,
↪ 10, 6, 23, 35, 53, 24547)}
```

Now run it distributedly, let's use dask as as the example

[6]:
```
from fugue_dask import DaskExecutionEngine

result = suggest_for_noniterative_objective(
    objective, space, top_n=1,
    execution_engine = DaskExecutionEngine
)[0]

print(result.sort_metric, result)
```

```
0.1909396653178624 {'trial': {'trial_id': '58c94f4f-011e-53da-a85b-7e696ced6600', 'params
↪': {'a': 0, 'b': 0.43696643500143395}, 'metadata': {}, 'keys': []}, 'metric': 0.
↪1909396653178624, 'params': {'a': 0, 'b': 0.43696643500143395}, 'metadata': {}, 'cost':
↪ 1.0, 'rung': 0, 'sort_metric': 0.1909396653178624, 'log_time': datetime.datetime(2021,
↪ 10, 6, 23, 36, 16, 996725)}
```

In order to use `tune` in a more elegant and easier way, let's firstly see how to configure the system.

4.2.2 Configuration

Configuring the system is not necessary but it has great benefit for simpifying your following works.

`suggest_for_noniterative_objective` and `optimize_noniterative` have a lot of parameters due to the complexity of tuning operations. But `tune` let you do global configuration so you don't need to repeat the same configuration for every tuning task.

Customize Optimizer Converter

[7]:
```
from tune import TUNE_OBJECT_FACTORY
from tune import NonIterativeObjectiveLocalOptimizer
from tune_hyperopt import HyperoptLocalOptimizer
from tune_optuna import OptunaLocalOptimizer
import optuna

optuna.logging.disable_default_handler()

def to_optimizer(obj):
    if isinstance(obj, NonIterativeObjectiveLocalOptimizer):
        return obj
    if obj is None or "hyperopt"==obj:
        return HyperoptLocalOptimizer(max_iter=20, seed=0)
    if "optuna" == obj:
        return OptunaLocalOptimizer(max_iter=20)
    raise NotImplementedError

# make default level 2 optimizer HyperoptLocalOptimizer, so you will not need to set␣
↪again
TUNE_OBJECT_FACTORY.set_noniterative_local_optimizer_converter(to_optimizer)
```

Customize Monitor

Monitor is to collect and render information in real time, there are builtin monitors, you can also create your own.

[9]:
```
from typing import Optional

from tune import TUNE_OBJECT_FACTORY
from tune import Monitor
from tune_notebook import (
    NotebookSimpleHist,
    NotebookSimpleRungs,
    NotebookSimpleTimeSeries,
    PrintBest,
)

def to_monitor(obj) -> Optional[Monitor]:
    if obj is None:
        return None
    if isinstance(obj, Monitor):
        return obj
    if isinstance(obj, str):
        if obj == "hist":
            return NotebookSimpleHist()
        if obj == "rungs":
            return NotebookSimpleRungs()
        if obj == "ts":
            return NotebookSimpleTimeSeries()
        if obj == "text":
            return PrintBest()
    raise NotImplementedError(obj)

TUNE_OBJECT_FACTORY.set_monitor_converter(to_monitor)
```

Set Temp Path For Tuning

Temp path can be used to store serialized partitions or checkpoints. Most top level API usage requires a valid temporary path. We can use factory method to set a global value.

Notice if you want to tune distributedly, you should set the path to a distributed file system, for example s3.

[10]:
```
TUNE_OBJECT_FACTORY.set_temp_path("/tmp")
```

4.2.3 Tuning Examples

Sometimes, your objective function requires a input dataframe. There are two ways to use dataframes in general:

	Pros	Cons
Take them as real dataframes, for example pandas dataframes.	Simple and intuitive	Either the datas ize can't scale or you have to couple with a distributed solution such as Spark
Take them from parameters, for example paths as parameters.	You have the full control how and when and whether to load the data. More scalable.	More code to make it work

In general, the second way is a better idea. But if your case can fit in the first scenario, then `tune` has a simple solution letting you take the pandas dataframes as input.

[11]:
```
from sklearn.datasets import load_diabetes
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import numpy as np

diabetes = load_diabetes(as_frame=True)["frame"]

def evaluate(train_df:pd.DataFrame, **kwargs) -> float:
    x, y = train_df.drop("target", axis=1), train_df["target"]
    model = RandomForestRegressor(**kwargs)
    # pay attention here, score is larger better so we return the negative value
    return -np.mean(cross_val_score(model, x, y, scoring="neg_mean_absolute_error",␣
↪cv=4))

evaluate(diabetes)
```

[11]:
```
46.646344389844394
```

With the given `diabetes` dataset and the objective function `evaluate` let's tune it in different ways

Hybrid Tuning

[13]:
```
# Grid search only
space = Space(n_estimators=Grid(100,200), random_state=0)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df"
)[0]

print(result.sort_metric, result)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
46.63103787878788 {'trial': {'trial_id': '5d719fa7-9537-58b1-86cd-fa69a4e75272', 'params
↪': {'n_estimators': 100, 'random_state': 0}, 'metadata': {}, 'keys': []}, 'metric': 46.
↪63103787878788, 'params': {'n_estimators': 100, 'random_state': 0}, 'metadata': {},
↪'cost': 1.0, 'rung': 0, 'sort_metric': 46.63103787878788, 'log_time': datetime.
↪datetime(2021, 10, 6, 23, 37, 11, 450017)}
```

[14]:
```
# grid + random
space = Space(n_estimators=Grid(100,200), max_depth=RandInt(2,10), random_state=0).
↪sample(3, seed=0)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df"
)[0]

print(result.sort_metric, result)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
46.52677715635581 {'trial': {'trial_id': '0a53519f-576b-5a9f-8ef9-4a7e7f69de1a', 'params
↪': {'n_estimators': 200, 'max_depth': 6, 'random_state': 0}, 'metadata': {}, 'keys':␣
↪[]}, 'metric': 46.52677715635581, 'params': {'n_estimators': 200, 'max_depth': 6,
↪'random_state': 0}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 46.
↪52677715635581, 'log_time': datetime.datetime(2021, 10, 6, 23, 37, 26, 492058)}
```

[16]:
```
# random + bayesian optimization (hyperopt is used by default)
space = Space(n_estimators=RandInt(50,200))* Space(max_depth=RandInt(2,10), random_
↪state=0).sample(2, seed=0)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df"
)[0]

print(result.sort_metric, result)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    local_optimizer="optuna" # switch to optuna for bayesian optimization
)[0]

print(result.sort_metric, result)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
46.419699856089416 {'trial': {'trial_id': '52919031-4f17-58d2-8cfc-e4a1d0e4555a', 'params
↪': {'n_estimators': 175, 'max_depth': 6, 'random_state': 0}, 'metadata': {}, 'keys':␣
↪[]}, 'metric': 46.419699856089416, 'params': {'n_estimators': 175, 'max_depth': 6,
↪'random_state': 0}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 46.
↪419699856089416, 'log_time': datetime.datetime(2021, 10, 6, 23, 38, 37, 355059)}
46.41622613826187 {'trial': {'trial_id': '52919031-4f17-58d2-8cfc-e4a1d0e4555a', 'params
↪': {'n_estimators': 176, 'max_depth': 6, 'random_state': 0}, 'metadata': {}, 'keys':␣
↪[]}, 'metric': 46.41622613826187, 'params': {'n_estimators': 176, 'max_depth': 6,
↪'random_state': 0}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 46.
↪41622613826187, 'log_time': datetime.datetime(2021, 10, 6, 23, 39, 9, 442020)}
```

Partition And Train And Tune

This is a very important feature of `tune`. Sometimes, partitioning the data and train and tune small independent models separately can generate better result. This is not necessarily true, but at least we make it very simple for you to try. You only need to specify `partition_keys`. And with a distributed engine, all independent tasks are fully parallelized.

[17]:
```
space = Space(n_estimators=Grid(50,200), max_depth=RandInt(2,10), random_state=0).
↪sample(2, seed=0)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    partition_keys = ["sex"]  # for male and females, we train and tune separately
)

for r in result:
    print(r.trial.keys, r.sort_metric, r)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
[0.0506801187398187] 42.48208345425722 {'trial': {'trial_id': '83f593dd-a3a2-5ac0-b389-
↪ee19f8cc1134', 'params': {'n_estimators': 200, 'max_depth': 8, 'random_state': 0},
↪'metadata': {}, 'keys': [0.0506801187398187]}, 'metric': 42.48208345425722, 'params': {
↪'n_estimators': 200, 'max_depth': 8, 'random_state': 0}, 'metadata': {}, 'cost': 1.0,
↪'rung': 0, 'sort_metric': 42.48208345425722, 'log_time': datetime.datetime(2021, 10, 6,
↪ 23, 40, 38, 579320)}
[-0.044641636506989] 46.66399292343497 {'trial': {'trial_id': '1759366d-de55-5418-b1b5-
↪48cf91f529a0', 'params': {'n_estimators': 50, 'max_depth': 8, 'random_state': 0},
↪'metadata': {}, 'keys': [-0.044641636506989]}, 'metric': 46.66399292343497, 'params': {
↪'n_estimators': 50, 'max_depth': 8, 'random_state': 0}, 'metadata': {}, 'cost': 1.0,
↪'rung': 0, 'sort_metric': 46.66399292343497, 'log_time': datetime.datetime(2021, 10, 6,
↪ 23, 40, 33, 356186)}
```

Distributed Tuning

`tune` is based on Fugue so it can run seamlessly using all Fugue supported execution engines and in the same way Fugue uses them.

[18]:
```
# This space is a combination of grid and random search
# all level 1 searches, so it can be fully distributed
space = Space(n_estimators=Grid(50,200), max_depth=RandInt(2,10), random_state=0).
↪sample(2, seed=0)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    partition_keys = ["sex"],
    execution_engine = DaskExecutionEngine  # this makes the tuning process distributed
)

for r in result:
    print(r.trial.keys, r.sort_metric, r)
```

```
[0.0506801187398187] 42.79742975473356 {'trial': {'trial_id': '0f2053de-71b2-514d-b4ff-
↪8495b93a042b', 'params': {'n_estimators': 200, 'max_depth': 6, 'random_state': 0},
↪'metadata': {}, 'keys': [0.0506801187398187]}, 'metric': 42.79742975473356, 'params': {
↪'n_estimators': 200, 'max_depth': 6, 'random_state': 0}, 'metadata': {}, 'cost': 1.0,
↪'rung': 0, 'sort_metric': 42.79742975473356, 'log_time': datetime.datetime(2021, 10, 6,
↪ 23, 40, 57, 795165)}
[-0.044641636506989] 47.480845528260254 {'trial': {'trial_id': '46da77b5-089d-57b9-8036-
↪0ca2e3646fdb', 'params': {'n_estimators': 200, 'max_depth': 6, 'random_state': 0},
↪'metadata': {}, 'keys': [-0.044641636506989]}, 'metric': 47.480845528260254, 'params':
↪{'n_estimators': 200, 'max_depth': 6, 'random_state': 0}, 'metadata': {}, 'cost': 1.0,
↪'rung': 0, 'sort_metric': 47.480845528260254, 'log_time': datetime.datetime(2021, 10,␣
↪6, 23, 41, 0, 714602)}
```

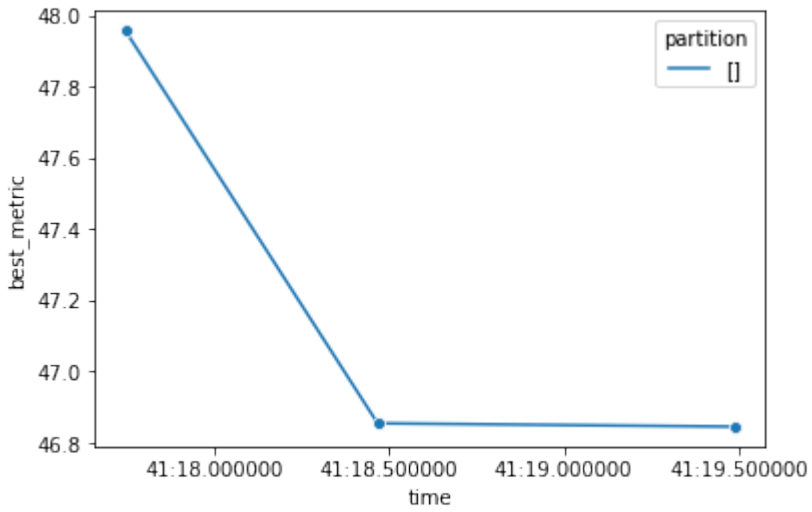
4.2.4 Realtime Monitoring

Fugue framework can let workers communicate with driver in realtime (see this). So `tune` leverages this feature for monitoring and iterative problems.

[19]:
```
space = Space(n_estimators=RandInt(1,20), max_depth=RandInt(2,10), random_state=0).
↪sample(100, seed=0)

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    monitor="ts"
)

for r in result:
    print(r.trial.keys, r.sort_metric, r)
```



```
[] 46.84555314021837 {'trial': {'trial_id': '2c9456ad-f8a7-56df-9195-3266ffabd941',
↪'params': {'n_estimators': 20, 'max_depth': 3, 'random_state': 0}, 'metadata': {},
↪'keys': []}, 'metric': 46.84555314021837, 'params': {'n_estimators': 20, 'max_depth':␣
↪3, 'random_state': 0}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 46.
↪84555314021837, 'log_time': datetime.datetime(2021, 10, 6, 23, 41, 19, 488640)}
[] 46.84555314021837 {'trial': {'trial_id': '2c9456ad-f8a7-56df-9195-3266ffabd941',
↪'params': {'n_estimators': 20, 'max_depth': 3, 'random_state': 0}, 'metadata': {},
↪'keys': []}, 'metric': 46.84555314021837, 'params': {'n_estimators': 20, 'max_depth':␣
↪3, 'random_state': 0}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 46.
↪84555314021837, 'log_time': datetime.datetime(2021, 10, 6, 23, 41, 23, 761028)}
```

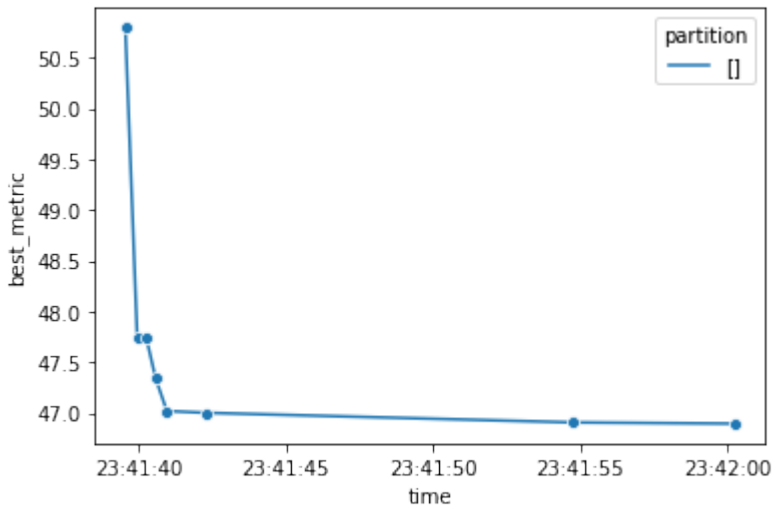
To enable monitoring on a distributed engine, you must also enable remote call back. Without shortcut, you have to set multiple configs. Here is an example with the `fuggle` package who sets the shortcuts for callbacks on Kaggle, it's as simple as one config: `callback: True`

[20]:
```
space = Space(n_estimators=RandInt(1,20), max_depth=RandInt(2,10), random_state=0, n_
↪jobs=1).sample(200, seed=0)

callback_conf = {
    "fugue.rpc.server": "fugue.rpc.flask.FlaskRPCServer",
    "fugue.rpc.flask_server.host": "0.0.0.0",
    "fugue.rpc.flask_server.port": "1234",
    "fugue.rpc.flask_server.timeout": "2 sec",
}

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    monitor="ts",
    execution_engine = DaskExecutionEngine,
    execution_engine_conf=callback_conf
)

for r in result:
    print(r.trial.keys, r.sort_metric, r)
```



```
[] 46.89339381813802 {'trial': {'trial_id': 'af51195c-3da6-59e5-a4ab-9802041ab314',
↪'params': {'n_estimators': 20, 'max_depth': 5, 'random_state': 0, 'n_jobs': 1},
↪'metadata': {}, 'keys': []}, 'metric': 46.89339381813802, 'params': {'n_estimators':␣
↪20, 'max_depth': 5, 'random_state': 0, 'n_jobs': 1}, 'metadata': {}, 'cost': 1.0, 'rung
↪': 0, 'sort_metric': 46.89339381813802, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪42, 0, 265059)}
[] 46.89339381813802 {'trial': {'trial_id': 'af51195c-3da6-59e5-a4ab-9802041ab314',
↪'params': {'n_estimators': 20, 'max_depth': 5, 'random_state': 0, 'n_jobs': 1},
↪'metadata': {}, 'keys': []}, 'metric': 46.89339381813802, 'params': {'n_estimators':␣
↪20, 'max_depth': 5, 'random_state': 0, 'n_jobs': 1}, 'metadata': {}, 'cost': 1.0, 'rung
↪': 0, 'sort_metric': 46.89339381813802, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪42, 0, 265059)}
```

For the shortcuts of monitoring

1. `ts` to monitor the up-to-date best metric collected
2. `hist` to motitor the histogram of metrics collected

4.2.5 Early Stopping

When you enable monitoring, you often see the curve flattens quickly, so it can save significant time if it can stop trying the remaining trials. To do early stopping, it is required to enable callbacks for distributed engine (for monitoring, if you don't monitor, you don't need to enable callback).

In `tune`, you can also combine stoppers with logical operators

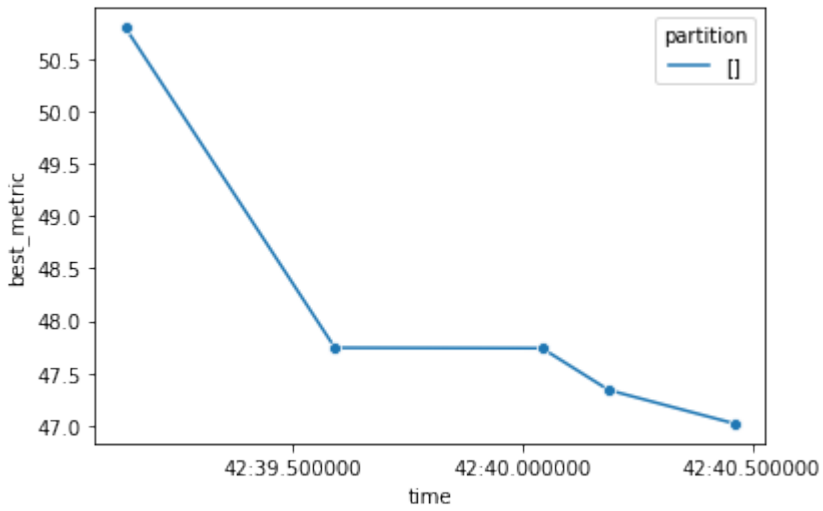
[21]:
```
from tune import small_improvement, n_updates

space = Space(n_estimators=RandInt(1,20), max_depth=RandInt(2,10), random_state=0, n_
↪jobs=1).sample(200, seed=0)

callback_conf = {
    "fugue.rpc.server": "fugue.rpc.flask.FlaskRPCServer",
    "fugue.rpc.flask_server.host": "0.0.0.0",
    "fugue.rpc.flask_server.port": "1234",
    "fugue.rpc.flask_server.timeout": "2 sec",
}

result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    monitor="ts",
    # stop if at least 5 updates on best
    # AND the last update on best improved less than 0.1 (abs value)
    stopper= n_updates(5) & small_improvement(0.1,1),
    execution_engine = DaskExecutionEngine,
    execution_engine_conf=callback_conf
)

for r in result:
    print(r.trial.keys, r.sort_metric, r)
```




```
[] 47.01773216903467 {'trial': {'trial_id': 'f84ce5f5-207b-5ab7-a81a-be80879d5431',
↪'params': {'n_estimators': 19, 'max_depth': 4, 'random_state': 0, 'n_jobs': 1},
↪'metadata': {}, 'keys': []}, 'metric': 47.01773216903467, 'params': {'n_estimators':␣
↪19, 'max_depth': 4, 'random_state': 0, 'n_jobs': 1}, 'metadata': {}, 'cost': 1.0, 'rung
↪': 0, 'sort_metric': 47.01773216903467, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪42, 40, 406040)}
[] 47.01773216903467 {'trial': {'trial_id': 'f84ce5f5-207b-5ab7-a81a-be80879d5431',
↪'params': {'n_estimators': 19, 'max_depth': 4, 'random_state': 0, 'n_jobs': 1},
↪'metadata': {}, 'keys': []}, 'metric': 47.01773216903467, 'params': {'n_estimators':␣
↪19, 'max_depth': 4, 'random_state': 0, 'n_jobs': 1}, 'metadata': {}, 'cost': 1.0, 'rung
↪': 0, 'sort_metric': 47.01773216903467, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪42, 40, 406040)}
```

The above example combined a warmup period `n_updates(5)` and improvement check `small_improvement(0.1, 1)` so it does not stop too early or too late.

You can also customize a simple stopper

[22]:
```
from typing import List
from tune.noniterative.stopper import SimpleNonIterativeStopper
from tune import TrialReport

def less_than(v: float) -> SimpleNonIterativeStopper:
    def func(current: TrialReport, updated: bool, reports: List[TrialReport]):
        return current.sort_metric <= v

    return SimpleNonIterativeStopper(func, log_best_only=True)
```

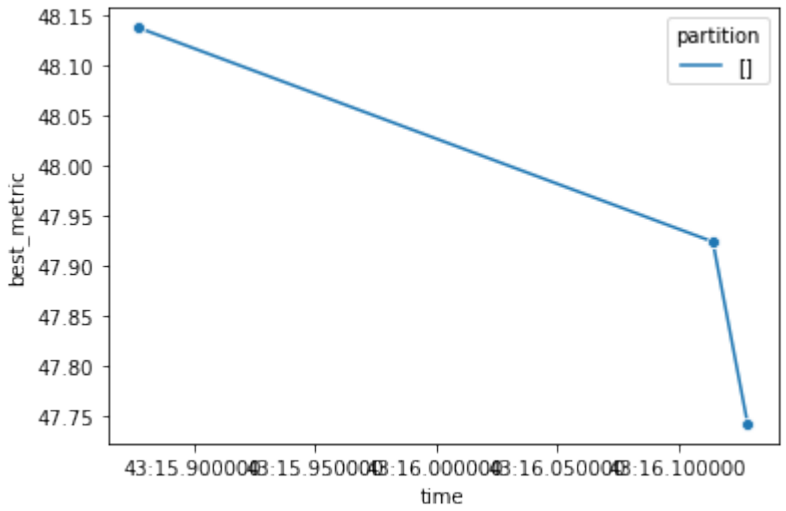
[23]:
```
result = suggest_for_noniterative_objective(
    evaluate, space, top_n=1,
    df = diabetes, df_name = "train_df",
    monitor="ts",
    stopper= less_than(49),
    execution_engine = DaskExecutionEngine,
    execution_engine_conf=callback_conf
```

(continues on next page)

(continued from previous page)

```
)

for r in result:
    print(r.trial.keys, r.sort_metric, r)
```




```
[] 47.74170052753941 {'trial': {'trial_id': 'b9ab0d11-991d-53d2-ad41-246dcbe23c22',
↪'params': {'n_estimators': 17, 'max_depth': 2, 'random_state': 0, 'n_jobs': 1},
↪'metadata': {}, 'keys': []}, 'metric': 47.74170052753941, 'params': {'n_estimators':␣
↪17, 'max_depth': 2, 'random_state': 0, 'n_jobs': 1}, 'metadata': {}, 'cost': 1.0, 'rung
↪': 0, 'sort_metric': 47.74170052753941, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪43, 15, 891806)}
[] 47.74170052753941 {'trial': {'trial_id': 'b9ab0d11-991d-53d2-ad41-246dcbe23c22',
↪'params': {'n_estimators': 17, 'max_depth': 2, 'random_state': 0, 'n_jobs': 1},
↪'metadata': {}, 'keys': []}, 'metric': 47.74170052753941, 'params': {'n_estimators':␣
↪17, 'max_depth': 2, 'random_state': 0, 'n_jobs': 1}, 'metadata': {}, 'cost': 1.0, 'rung
↪': 0, 'sort_metric': 47.74170052753941, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪43, 15, 891806)}
```

The stopper will try to do graceful stop, so after the stop criteria, some running trials may still finish in with a distributed engine and report back, that is normal. If you want to stop faster, for example set: `stop_check_interval: "5sec"`. But if you have a lot of workers, the frequent check may be a burden on the driver side, it also depends on how heavy compute your custom stopper is using.

Notice: You must create new stoppers everytime you call `suggest_for_noniterative_objective` because `SimpleNonIterativeStopper` is stateful.

```
[ ]:
```

4.3 Non-Iterative Objective

Non-Iterative Objective refers to the objective functions with single iteration. They do not report progress during the execution to get a pruning decision.

4.3.1 Interfaceless

The simplest way to construct a Tune compatible non-iterative objective is to wirte a native python function with type annotations.

[3]:
```
from typing import Tuple, Dict, Any

def objective1(a, b) -> float:
    return a**2 + b**2

def objective2(a, b) -> Tuple[float, Dict[str, Any]]:
    return a**2 + b**2, {"metadata":"x"}
```

If you function as `float` or `Tuple[float, Dict[str, Any]]` as output annotation, they are valid non-iterative objectives for `tune`

`Tuple[float, Dict[str, Any]]` is to return both the metric and metadata.

The following code demos how it works on the backend to convert your simple functions to `tune` compatible objects. You normally don't need to do that by yourself.

[5]:
```
from tune import to_noniterative_objective, Trial

f1 = to_noniterative_objective(objective1)
f2 = to_noniterative_objective(objective2, min_better=False)

trial = Trial("id", params=dict(a=1,b=1))
report1 = f1.safe_run(trial)
report2 = f2.safe_run(trial)

print(type(f1))
print(report1.metric, report1.sort_metric, report1.metadata)
print(report2.metric, report2.sort_metric, report2.metadata)
```

```
<class 'tune.noniterative.convert._NonIterativeObjectiveFuncWrapper'>
2.0 2.0 {}
2.0 -2.0 {'metadata': 'x'}
```

4.3.2 Decorator Approach

It is equivalent to use decorator on top of the functions. But now your functions depend on `tune` package.

[7]:
```
from tune import noniterative_objective

@noniterative_objective
def objective_3(a, b) -> float:
    return a**2 + b**2
```

(continues on next page)

(continued from previous page)

```
@noniterative_objective(min_better=False)
def objective_4(a, b) -> Tuple[float, Dict[str, Any]]:
    return a**2 + b**2, {"metadata":"x"}

report3 = objective_3.safe_run(trial)
report4 = objective_4.safe_run(trial)

print(report3.metric, report3.sort_metric, report3.metadata)
print(report4.metric, report4.sort_metric, report4.metadata)
```

```
2.0 2.0 {}
2.0 -2.0 {'metadata': 'x'}
```

4.3.3 Interface Approach

With interface approach, you can access all properties of a trial. Also you can use more flexible logic to generate sort metric.

[9]:
```
from tune import NonIterativeObjectiveFunc, TrialReport

class Objective(NonIterativeObjectiveFunc):
    def generate_sort_metric(self, value: float) -> float:
        return - value * 10

    def run(self, trial: Trial) -> TrialReport:
        params = trial.params.simple_value
        metric = params["a"]**2 + params["b"]**2
        return TrialReport(trial, metric, metadata=dict(m="x"))

report = Objective().safe_run(trial)
print(report.metric, report.sort_metric, report.metadata)
```

```
2.0 -20.0 {'m': 'x'}
```

4.3.4 Factory Method

Almost all higher level APIs of `tune` are using `TUNE_OBJECT_FACTORY` to convert various objects to `NonIterativeObjectiveFunc`.

[10]:
```
from tune import TUNE_OBJECT_FACTORY

assert isinstance(TUNE_OBJECT_FACTORY.make_noniterative_objective(objective1),␣
↪NonIterativeObjectiveFunc)
assert isinstance(TUNE_OBJECT_FACTORY.make_noniterative_objective(objective_4),␣
↪NonIterativeObjectiveFunc)
assert isinstance(TUNE_OBJECT_FACTORY.make_noniterative_objective(Objective()),␣
↪NonIterativeObjectiveFunc)
```

That is why in the higher level APIs, you can just pass in a very simple python function as objective but `tune` is still able to recognize.

Actually you can make it even more flexible by configuring the factory.

[11]:
```
def to_obj(obj):
    if obj == "test":
        return to_noniterative_objective(objective1, min_better=False)
    if isinstance(obj, NonIterativeObjectiveFunc):
        return obj
    raise NotImplementedError

TUNE_OBJECT_FACTORY.set_noniterative_objective_converter(to_obj)  # user to_obj to␣
↪replace the built-in default converter

assert isinstance(TUNE_OBJECT_FACTORY.make_noniterative_objective("test"),␣
↪NonIterativeObjectiveFunc)
```

If you customize in this way, then you can pass in `test` to the higher level tuning APIs, and it will be recognized as a compatible objective.

This is a common approach in Fugue projects. It enables you to use mostly primitive data types to represent what you want to do. For advanced users, if you spend some time on such configuration (one time effort), you will find the code is even simpler and less dependent on `fugue` and `tune`.

[ ]:
```
```

4.4 Non-Iterative Optimizers

AKA Level 2 optimizers, are unified 3rd party solutions for random expressions. Look at this space:

[1]:
```
from tune import Space, Grid, Rand

space = Space(a=Grid(1,2), b=Rand(0,1))
list(space)
```

[1]:
```
[{'a': 1, 'b': Rand(low=0, high=1, q=None, log=False, include_high=True)},
 {'a': 2, 'b': Rand(low=0, high=1, q=None, log=False, include_high=True)}]
```

`Grid` is for level 1 optimization, all level 1 parameters will be converted to static values before execution. And level 2 parameters will be optimized during runtime using level 2 optimizers. So for the above example, if we have a Spark cluster and Hyperopt, then we can use Hyperot to search for the best `b` on each of the 2 configurations. And the 2 jobs are parallelized by Spark.

[3]:
```
from tune import noniterative_objective, Trial

@noniterative_objective
def objective(a ,b) -> float:
    return a**2 + b**2

trial = Trial("dummy", params=list(space)[0])
```

4.4.1 Use Directly

Notice normally you don't use them directly, instead you should use them through top level APIs. This is just to demo how they work.

Hyperopt

[5]:
```
from tune_hyperopt import HyperoptLocalOptimizer

hyperopt_optimizer = HyperoptLocalOptimizer(max_iter=200, seed=0)
report = hyperopt_optimizer.run(objective, trial)

print(report.sort_metric, report)
```

```
1.0000000001665414 {'trial': {'trial_id': 'dummy', 'params': {'a': 1, 'b': 1.
↪2905089873156781e-05}, 'metadata': {}, 'keys': []}, 'metric': 1.0000000001665414,
↪'params': {'a': 1, 'b': 1.2905089873156781e-05}, 'metadata': {}, 'cost': 1.0, 'rung':␣
↪0, 'sort_metric': 1.0000000001665414, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪30, 51, 970344)}
```

Optuna

[7]:
```
from tune_optuna import OptunaLocalOptimizer
import optuna

optuna.logging.disable_default_handler()

optuna_optimizer = OptunaLocalOptimizer(max_iter=200)
report = optuna_optimizer.run(objective, trial)

print(report.sort_metric, report)
```

```
1.0000000003655019 {'trial': {'trial_id': 'dummy', 'params': {'a': 1, 'b': 1.
↪9118105424729645e-05}, 'metadata': {}, 'keys': []}, 'metric': 1.0000000003655019,
↪'params': {'a': 1, 'b': 1.9118105424729645e-05}, 'metadata': {}, 'cost': 1.0, 'rung':␣
↪0, 'sort_metric': 1.0000000003655019, 'log_time': datetime.datetime(2021, 10, 6, 23,␣
↪31, 26, 6566)}
```

As you see, we have unified the interfaces for using these frameworks. In addition, we also unified the semantic of the random expressions, so the random sampling behavior will be highly consistent on different 3rd party solutions.

4.4.2 Use Top Level API

In the following example, we directly use the entire `space` where you can mix grid search, random search and Bayesian Optimization.

[8]:
```
from tune import suggest_for_noniterative_objective

report = suggest_for_noniterative_objective(
    objective, space, top_n=1,
```

(continues on next page)

(continued from previous page)

```
    local_optimizer=hyperopt_optimizer
)[0]

print(report.sort_metric, report)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
1.0000000001665414 {'trial': {'trial_id': '971ef4a5-71a9-5bf2-b2a4-f0f1acd02b78', 'params
↪': {'a': 1, 'b': 1.2905089873156781e-05}, 'metadata': {}, 'keys': []}, 'metric': 1.
↪0000000001665414, 'params': {'a': 1, 'b': 1.2905089873156781e-05}, 'metadata': {},
↪'cost': 1.0, 'rung': 0, 'sort_metric': 1.0000000001665414, 'log_time': datetime.
↪datetime(2021, 10, 6, 23, 31, 43, 784128)}
```

You can also provide only random expressions in space, and use in the same way so it looks like a common case similar to the examples

[14]:
```
report = suggest_for_noniterative_objective(
    objective, Space(a=Rand(-1,1), b=Rand(-100,100)), top_n=1,
    local_optimizer=optuna_optimizer
)[0]

print(report.sort_metric, report)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
0.04085386621249434 {'trial': {'trial_id': '45179c01-7358-5546-8f41-d7c6f120523f',
↪'params': {'a': 0.01604913454189394, 'b': 0.20148521408021614}, 'metadata': {}, 'keys':
↪ []}, 'metric': 0.04085386621249434, 'params': {'a': 0.01604913454189394, 'b': 0.
↪20148521408021614}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 0.
↪04085386621249434, 'log_time': datetime.datetime(2021, 10, 6, 23, 34, 47, 379901)}
```

4.4.3 Factory Method

In the above example, if we don't set `local_optimizer`, then the default level 2 optimizer will be used which can't handle a configuration with random expressions.

So we have a nice way to make certain optimizer the default one.

[10]:
```
from tune import NonIterativeObjectiveLocalOptimizer, TUNE_OBJECT_FACTORY

def to_optimizer(obj):
    if isinstance(obj, NonIterativeObjectiveLocalOptimizer):
        return obj
    if obj is None or "hyperopt"==obj:
        return HyperoptLocalOptimizer(max_iter=200, seed=0)
    if "optuna" == obj:
        return OptunaLocalOptimizer(max_iter=200)
    raise NotImplementedError

TUNE_OBJECT_FACTORY.set_noniterative_local_optimizer_converter(to_optimizer)
```

Now Hyperopt becomes the default level 2 optimizer, and you can switch to Optuna by specifying a string parameter

[16]:
```
report = suggest_for_noniterative_objective(
    objective, Space(a=Rand(-1,1), b=Rand(-100,100)), top_n=1
)[0]  # using hyperopt

print(report.sort_metric, report)

report = suggest_for_noniterative_objective(
    objective, Space(a=Rand(-1,1), b=Rand(-100,100)), top_n=1,
    local_optimizer="optuna"
)[0]  # using hyperopt

print(report.sort_metric, report)
```

```
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
NativeExecutionEngine doesn't respect num_partitions ROWCOUNT
```

```
0.02788888054657708 {'trial': {'trial_id': '45179c01-7358-5546-8f41-d7c6f120523f',
↪'params': {'a': -0.13745463941867586, 'b': -0.09484251498594332}, 'metadata': {}, 'keys
↪': []}, 'metric': 0.02788888054657708, 'params': {'a': -0.13745463941867586, 'b': -0.
↪09484251498594332}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 0.
↪02788888054657708, 'log_time': datetime.datetime(2021, 10, 6, 23, 35, 19, 961138)}
0.010490219126635992 {'trial': {'trial_id': '45179c01-7358-5546-8f41-d7c6f120523f',
↪'params': {'a': 0.06699961867542388, 'b': -0.07746786575079878}, 'metadata': {}, 'keys
↪': []}, 'metric': 0.010490219126635992, 'params': {'a': 0.06699961867542388, 'b': -0.
↪07746786575079878}, 'metadata': {}, 'cost': 1.0, 'rung': 0, 'sort_metric': 0.
↪010490219126635992, 'log_time': datetime.datetime(2021, 10, 6, 23, 35, 21, 593974)}
```

[ ]:
```
```

4.5 Tune Dataset

`TuneDataset` contains searching space and all related dataframes with metadata for a tuning task.

`TuneDataset` should not to be constructed by users directly. Instead, you should use `TuneDatasetBuilder` or the factory method to construct `TuneDataset`.

[1]:
```
from fugue_notebook import setup

setup(is_lab=True)

import pandas as pd
from tune import TUNE_OBJECT_FACTORY, TuneDatasetBuilder, Space, Grid
from fugue import FugueWorkflow
```

`TUNE_OBJECT_FACTORY.make_dataset` is a wrapper of `TuneDatasetBuilder`, making the dataset construction even easier. But `TuneDatasetBuilder` still has the most flexibility. For example, it can add multiple dataframes with different join types while `TUNE_OBJECT_FACTORY.make_dataset` can add at most two dataframes (nomrally train and validations dataframes).

[2]:
```
with FugueWorkflow() as dag:
    builder = TuneDatasetBuilder(Space(a=1, b=2))
    dataset = builder.build(dag)
```

(continues on next page)

(continued from previous page)

```
    dataset.data.show();

with FugueWorkflow() as dag:
    dataset = TUNE_OBJECT_FACTORY.make_dataset(dag, Space(a=1, b=2))
    dataset.data.show();
```

```
                                    __tune_trials__
0  gASVXwEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

```
<IPython.core.display.HTML object>
```

```
                                    __tune_trials__
0  gASVXwEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

```
<IPython.core.display.HTML object>
```

Here are the equivalent ways to construct `TuneDataset` with space and two dataframes.

In `TuneDataset`, every dataframe will be partition by certain keys, and each partition will be saved into a temp parquet file. The temp path must be specified. Using the factory, you can call `set_temp_path` once so you no longer need to provide the temp path explicitly, if you still provide a path, it will be used.

[3]:
```
pdf1 = pd.DataFrame([[0,1],[1,1],[0,2]], columns = ["a", "b"])
pdf2 = pd.DataFrame([[0,0.5],[2,0.1],[0,0.1],[1,0.3]], columns = ["a", "c"])
space = Space(a=1, b=Grid(1,2,3))

with FugueWorkflow() as dag:
    builder = TuneDatasetBuilder(space, path="/tmp")
    # here we must make pdf1 pdf2 the FugueWorkflowDataFrame, and they
    # both need to be partitioned by the same keys so each partition
    # will be saved to a temp parquet file, and the chunks of data are
    # replaced by file paths before join.
    builder.add_df("df1", dag.df(pdf1).partition_by("a"))
    builder.add_df("df2", dag.df(pdf2).partition_by("a"), how="inner")
    dataset = builder.build(dag)
    dataset.data.show();


TUNE_OBJECT_FACTORY.set_temp_path("/tmp")

with FugueWorkflow() as dag:
    # this method is significantly simpler, as long as you don't have more
    # than 2 dataframes for a tuning task, use this.
    dataset = TUNE_OBJECT_FACTORY.make_dataset(
        dag, space,
        df_name="df1", df=pdf1,
        test_df_name="df2", test_df=pdf2,
        partition_keys=["a"],
    )
    dataset.data.show();
```

```
   a                                    __tune_df__df1  \
0  0  /tmp/01b823d6-2d65-43be-898d-ed4d5b1ab582.parquet
1  0  /tmp/01b823d6-2d65-43be-898d-ed4d5b1ab582.parquet
2  0  /tmp/01b823d6-2d65-43be-898d-ed4d5b1ab582.parquet
```

(continues on next page)

(continued from previous page)

```
3  1  /tmp/15f2ec83-3494-4ba8-80a5-fa7c558c273c.parquet
4  1  /tmp/15f2ec83-3494-4ba8-80a5-fa7c558c273c.parquet
5  1  /tmp/15f2ec83-3494-4ba8-80a5-fa7c558c273c.parquet

                                      __tune_df__df2  \
0  /tmp/5c35d480-6fa8-4776-a0f9-770974b73bb4.parquet
1  /tmp/5c35d480-6fa8-4776-a0f9-770974b73bb4.parquet
2  /tmp/5c35d480-6fa8-4776-a0f9-770974b73bb4.parquet
3  /tmp/2fe00d9c-b690-49c6-87a5-d365d59066c6.parquet
4  /tmp/2fe00d9c-b690-49c6-87a5-d365d59066c6.parquet
5  /tmp/2fe00d9c-b690-49c6-87a5-d365d59066c6.parquet

                                    __tune_trials__
0  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
1  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
2  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
3  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
4  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
5  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

```
<IPython.core.display.HTML object>
```

```
   a                                     __tune_df__df1  \
0  0  /tmp/943302c8-2704-4b29-a2ac-64946352a90d.parquet
1  0  /tmp/943302c8-2704-4b29-a2ac-64946352a90d.parquet
2  0  /tmp/943302c8-2704-4b29-a2ac-64946352a90d.parquet
3  1  /tmp/74fa6215-116d-4828-a49c-f58358a9b4e7.parquet
4  1  /tmp/74fa6215-116d-4828-a49c-f58358a9b4e7.parquet
5  1  /tmp/74fa6215-116d-4828-a49c-f58358a9b4e7.parquet

                                      __tune_df__df2  \
0  /tmp/9084e1ad-2156-4f3a-be36-52cf55d5c2fb.parquet
1  /tmp/9084e1ad-2156-4f3a-be36-52cf55d5c2fb.parquet
2  /tmp/9084e1ad-2156-4f3a-be36-52cf55d5c2fb.parquet
3  /tmp/0aa2aae2-3ab7-46e7-82e2-34a14ded2f0f.parquet
4  /tmp/0aa2aae2-3ab7-46e7-82e2-34a14ded2f0f.parquet
5  /tmp/0aa2aae2-3ab7-46e7-82e2-34a14ded2f0f.parquet

                                    __tune_trials__
0  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
1  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
2  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
3  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
4  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
5  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

```
<IPython.core.display.HTML object>
```

We got 6 rows, because the space will contain 3 configurations. And since for the dataframes, we partitioned by `a` and inner joined, there will be 2 rows. So in total there are 6 rows in the `TuneDataset`.

Notice, the number of rows of TuneDataset determines max parallelism. For this case, if you assign 10 workers, 4 will always be idle.

Actually, a more common case is that for each of the dataframe, we don't partition at all. For `TUNE_OBJECT_FACTORY.`

`make_dataset` we just need to remove the `partition_keys`.

[4]:
```
with FugueWorkflow() as dag:
    dataset = TUNE_OBJECT_FACTORY.make_dataset(
        dag, space,
        df_name="df1", df=pdf1,
        test_df_name="df2", test_df=pdf2,
    )
    dataset.data.show();
```

```
                                      __tune_df__df1  \
0  /tmp/a774965e-d0df-417c-84d0-bb693ac337d1.parquet
1  /tmp/a774965e-d0df-417c-84d0-bb693ac337d1.parquet
2  /tmp/a774965e-d0df-417c-84d0-bb693ac337d1.parquet

                                      __tune_df__df2  \
0  /tmp/2f9a93cd-121b-4697-8fe9-0513aa6bcd82.parquet
1  /tmp/2f9a93cd-121b-4697-8fe9-0513aa6bcd82.parquet
2  /tmp/2f9a93cd-121b-4697-8fe9-0513aa6bcd82.parquet

                                     __tune_trials__
0  gASVXwEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
1  gASVXwEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
2  gASVXwEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

```
<IPython.core.display.HTML object>
```

But what if we want to partition on `df1` but not on `df2`? Then again, you can use `TuneDatasetBuilder`

[5]:
```
with FugueWorkflow() as dag:
    builder = TuneDatasetBuilder(space, path="/tmp")
    builder.add_df("df1", dag.df(pdf1).partition_by("a"))
    # use cross join because there no common key
    builder.add_df("df2", dag.df(pdf2), how="cross")
    dataset = builder.build(dag)
    dataset.data.show();
```

```
   a                                     __tune_df__df1  \
0  0  /tmp/4e16f5d7-1dc2-438c-86c7-504502c3e1ad.parquet
1  0  /tmp/4e16f5d7-1dc2-438c-86c7-504502c3e1ad.parquet
2  0  /tmp/4e16f5d7-1dc2-438c-86c7-504502c3e1ad.parquet
3  1  /tmp/058862d5-4c24-437e-ae38-c4810d071a11.parquet
4  1  /tmp/058862d5-4c24-437e-ae38-c4810d071a11.parquet
5  1  /tmp/058862d5-4c24-437e-ae38-c4810d071a11.parquet

                                      __tune_df__df2  \
0  /tmp/3b92a6f2-31aa-485e-a608-58dcdc925a3c.parquet
1  /tmp/3b92a6f2-31aa-485e-a608-58dcdc925a3c.parquet
2  /tmp/3b92a6f2-31aa-485e-a608-58dcdc925a3c.parquet
3  /tmp/3b92a6f2-31aa-485e-a608-58dcdc925a3c.parquet
4  /tmp/3b92a6f2-31aa-485e-a608-58dcdc925a3c.parquet
5  /tmp/3b92a6f2-31aa-485e-a608-58dcdc925a3c.parquet

                                     __tune_trials__
0  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

(continues on next page)

(continued from previous page)

```
1  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
2  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
3  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
4  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
5  gASVYgEAAAAAAABdlIwYdHVuZS5jb25jZXB0cy5mbG93Ln...
```

```
<IPython.core.display.HTML object>
```

[ ]:

4.6 Checkpoint

Checkpoint is normally constructed and provided to you, but if you are interested, this can give you some details.

[4]:
```
from tune import Checkpoint
from triad import FileSystem

root = FileSystem()
fs = root.makedirs("/tmp/test", recreate=True)
checkpoint = Checkpoint(fs)
print(len(checkpoint))
```

```
0
```

[5]:
```
!ls /tmp/test
```

[6]:
```
with checkpoint.create() as folder:
    folder.writetext("a.txt", "test")
```

[7]:
```
!ls /tmp/test
```

```
STATE  d9ed2530-20f1-42b3-8818-7fbf1b8eedf3
```

Here is how to create a new checkpoint under `/tmp/test`

[8]:
```
with checkpoint.create() as folder:
    folder.writetext("a.txt", "test2")
```

[9]:
```
!ls /tmp/test/*/
```

```
/tmp/test/8d4e7fed-2a4c-4789-a732-0cb46294e704/:
a.txt

/tmp/test/d9ed2530-20f1-42b3-8818-7fbf1b8eedf3/:
a.txt
```

Here is how to get the latest checkpoint folder

[10]:
```
print(len(checkpoint))
print(checkpoint.latest.readtext("a.txt"))
```

```
2
test2
```

[ ]:

PYTHON MODULE INDEX

t

tune.api.factory, 19
tune.api.optimize, 20
tune.api.suggest, 21
tune.concepts.checkpoint, 36
tune.concepts.dataset, 37
tune.concepts.flow.judge, 23
tune.concepts.flow.report, 26
tune.concepts.flow.trial, 29
tune.concepts.space.parameters, 30
tune.concepts.space.spaces, 35
tune.constants, 47
tune.exceptions, 47
tune.iterative.asha, 40
tune.iterative.objective, 41
tune.iterative.sha, 42
tune.iterative.study, 42
tune.noniterative.convert, 43
tune.noniterative.objective, 43
tune.noniterative.stopper, 44
tune.noniterative.study, 47
tune_hyperopt.optimizer, 47
tune_notebook.monitors, 55
tune_optuna.optimizer, 48
tune_sklearn.objective, 48
tune_sklearn.suggest, 49
tune_sklearn.utils, 51
tune_tensorflow.objective, 51
tune_tensorflow.spec, 52
tune_tensorflow.suggest, 53
tune_tensorflow.utils, 55
tune_test.local_optmizer, 56

INDEX

A

add_df() (*TuneDatasetBuilder method*), 38
add_dfs() (*TuneDatasetBuilder method*), 39
always_checkpoint (*ASHAJudge property*), 40
ASHAJudge (*class in tune.iterative.asha*), 40

B

best (*RungHeap property*), 40
best (*TrialReportLogger property*), 28
bests (*RungHeap property*), 40
budget (*TrialDecision property*), 25
build() (*TuneDatasetBuilder method*), 39

C

can_accept() (*ASHAJudge method*), 40
can_accept() (*NonIterativeStopper method*), 44
can_accept() (*NoOpTrailJudge method*), 24
can_accept() (*RemoteTrialJudge method*), 24
can_accept() (*TrialJudge method*), 25
capacity (*RungHeap property*), 40
Checkpoint (*class in tune.concepts.checkpoint*), 36
Choice (*class in tune.concepts.space.parameters*), 30
compile_model() (*KerasTrainingSpec method*), 52
compute_sort_metric() (*KerasTrainingSpec method*), 52
concat() (*TuningParametersTemplate method*), 33
copy() (*IterativeObjectiveFunc method*), 41
copy() (*KerasObjective method*), 51
copy() (*Trial method*), 29
copy() (*TrialReport method*), 27
cost (*TrialReport property*), 27
create() (*Checkpoint method*), 36
current_trial (*IterativeObjectiveFunc property*), 41

D

data (*TuneDataset property*), 38
decode() (*TuningParametersTemplate static method*), 33
dfs (*KerasTrainingSpec property*), 52
dfs (*Trial property*), 29
dfs (*TuneDataset property*), 38
distributable (*NonIterativeObjectiveLocalOptimizer property*), 43

E

empty (*TuningParametersTemplate property*), 34
encode() (*TuningParametersTemplate method*), 34
entrypoint() (*TrialCallback method*), 25
extract_keras_spec() (*in module tune_tensorflow.utils*), 55

F

fill() (*TuningParametersTemplate method*), 34
fill_dict() (*TrialReport method*), 27
fill_dict() (*TuningParametersTemplate method*), 34
finalize() (*IterativeObjectiveFunc method*), 41
finalize() (*KerasObjective method*), 51
finalize() (*KerasTrainingSpec method*), 52
finalize() (*Monitor method*), 23
finalize() (*NotebookSimpleChart method*), 55
fit() (*KerasTrainingSpec method*), 52
full (*RungHeap property*), 40
FuncParam (*class in tune.concepts.space.parameters*), 30

G

generate() (*Choice method*), 30
generate() (*NormalRand method*), 31
generate() (*NormalRandInt method*), 31
generate() (*Rand method*), 31
generate() (*RandInt method*), 32
generate() (*StochasticExpression method*), 32
generate_many() (*StochasticExpression method*), 32
generate_sort_metric() (*IterativeObjectiveFunc method*), 41
generate_sort_metric() (*KerasObjective method*), 51
generate_sort_metric() (*KerasTrainingSpec method*), 52
generate_sort_metric() (*NonIterativeObjectiveFunc method*), 43
generate_sort_metric() (*SKObjective method*), 49
generate_sort_metric() (*TrialReport method*), 27
get_budget() (*ASHAJudge method*), 40
get_budget() (*NoOpTrailJudge method*), 24
get_budget() (*RemoteTrialJudge method*), 25
get_budget() (*TrialJudge method*), 26

get_compile_params() (*KerasTrainingSpec method*), 53
get_fit_metric() (*KerasTrainingSpec method*), 53
get_fit_params() (*KerasTrainingSpec method*), 53
get_model() (*KerasTrainingSpec method*), 53
get_path_or_temp() (*TuneObjectFactory method*), 19
get_reports() (*NonIterativeStopper method*), 44
get_reports() (*NonIterativeStopperCombiner method*), 45
Grid (*class in tune.concepts.space.parameters*), 30

H

has_grid (*TuningParametersTemplate property*), 34
has_stochastic (*Space property*), 36
has_stochastic (*TuningParametersTemplate property*), 34
HyperoptLocalOptimizer (*class in tune_hyperopt.optimizer*), 47

I

initialize() (*IterativeObjectiveFunc method*), 41
initialize() (*KerasObjective method*), 51
initialize() (*Monitor method*), 23
IterativeObjectiveFunc (*class in tune.iterative.objective*), 41
IterativeStudy (*class in tune.iterative.study*), 42

J

jsondict (*Choice property*), 30
jsondict (*NormalRand property*), 31
jsondict (*NormalRandInt property*), 31
jsondict (*Rand property*), 31
jsondict (*RandInt property*), 32
jsondict (*StochasticExpression property*), 32
jsondict (*TransitionChoice property*), 33
judge() (*ASHAJudge method*), 40
judge() (*NonIterativeStopper method*), 44
judge() (*NoOpTrailJudge method*), 24
judge() (*RemoteTrialJudge method*), 25
judge() (*TrialJudge method*), 26

K

keras_space() (*in module tune_tensorflow.utils*), 55
KerasObjective (*class in tune_tensorflow.objective*), 51
KerasTrainingSpec (*class in tune_tensorflow.spec*), 52
keys (*Trial property*), 29
keys (*TuneDataset property*), 38

L

latest (*Checkpoint property*), 37
load_checkpoint() (*IterativeObjectiveFunc method*), 41
load_checkpoint() (*KerasObjective method*), 51
load_checkpoint() (*KerasTrainingSpec method*), 53
log() (*TrialReportCollection method*), 46
log() (*TrialReportLogger method*), 28
log_time (*TrialReport property*), 27

M

make_dataset() (*TuneObjectFactory method*), 19
make_optimizer() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
metadata (*Trial property*), 29
metadata (*TrialDecision property*), 25
metadata (*TrialReport property*), 27
metric (*TrialReport property*), 27
model (*KerasObjective property*), 52
module
    tune.api.factory, 19
    tune.api.optimize, 20
    tune.api.suggest, 21
    tune.concepts.checkpoint, 36
    tune.concepts.dataset, 37
    tune.concepts.flow.judge, 23
    tune.concepts.flow.report, 26
    tune.concepts.flow.trial, 29
    tune.concepts.space.parameters, 30
    tune.concepts.space.spaces, 35
    tune.constants, 47
    tune.exceptions, 47
    tune.iterative.asha, 40
    tune.iterative.objective, 41
    tune.iterative.sha, 42
    tune.iterative.study, 42
    tune.noniterative.convert, 43
    tune.noniterative.objective, 43
    tune.noniterative.stopper, 44
    tune.noniterative.study, 47
    tune_hyperopt.optimizer, 47
    tune_notebook.monitors, 55
    tune_optuna.optimizer, 48
    tune_sklearn.objective, 48
    tune_sklearn.suggest, 49
    tune_sklearn.utils, 51
    tune_tensorflow.objective, 51
    tune_tensorflow.spec, 52
    tune_tensorflow.suggest, 53
    tune_tensorflow.utils, 55
    tune_test.local_optmizer, 56
Monitor (*class in tune.concepts.flow.judge*), 23
monitor (*TrialJudge property*), 26

N

n_samples() (*in module tune.noniterative.stopper*), 46
n_updates() (*in module tune.noniterative.stopper*), 46
NewCheckpoint (*class in tune.concepts.checkpoint*), 37
next_tune_dataset() (*StudyResult method*), 37

no_update_period() (*in module tune.noniterative.stopper*), 46
noniterative_objective() (*in module tune.noniterative.convert*), 43
NonIterativeObjectiveFunc (*class in tune.noniterative.objective*), 43
NonIterativeObjectiveLocalOptimizer (*class in tune.noniterative.objective*), 43
NonIterativeObjectiveLocalOptimizerTests (*class in tune_test.local_optmizer*), 56
NonIterativeObjectiveLocalOptimizerTests.Tests (*class in tune_test.local_optmizer*), 56
NonIterativeStopper (*class in tune.noniterative.stopper*), 44
NonIterativeStopperCombiner (*class in tune.noniterative.stopper*), 45
NonIterativeStudy (*class in tune.noniterative.study*), 47
NoOpTrailJudge (*class in tune.concepts.flow.judge*), 24
NormalRand (*class in tune.concepts.space.parameters*), 30
NormalRandInt (*class in tune.concepts.space.parameters*), 31
NotebookSimpleChart (*class in tune_notebook.monitors*), 55
NotebookSimpleHist (*class in tune_notebook.monitors*), 56
NotebookSimpleRungs (*class in tune_notebook.monitors*), 56
NotebookSimpleTimeSeries (*class in tune_notebook.monitors*), 56

O

on_get_budget() (*Monitor method*), 24
on_judge() (*Monitor method*), 24
on_report() (*Monitor method*), 24
on_report() (*NonIterativeStopper method*), 45
on_report() (*NonIterativeStopperCombiner method*), 45
on_report() (*NotebookSimpleChart method*), 55
on_report() (*PrintBest method*), 56
on_report() (*SimpleNonIterativeStopper method*), 45
on_report() (*TrialReportLogger method*), 28
optimize() (*IterativeStudy method*), 42
optimize() (*NonIterativeStudy method*), 47
optimize_by_continuous_asha() (*in module tune.api.optimize*), 20
optimize_by_hyperband() (*in module tune.api.optimize*), 20
optimize_by_sha() (*in module tune.api.optimize*), 20
optimize_noniterative() (*in module tune.api.optimize*), 20
OptunaLocalOptimizer (*class in tune_optuna.optimizer*), 48

P

params (*KerasTrainingSpec property*), 53
params (*Trial property*), 29
params (*TrialReport property*), 27
params (*TuningParametersTemplate property*), 34
params_dict (*TuningParametersTemplate property*), 34
plot() (*NotebookSimpleChart method*), 56
plot() (*NotebookSimpleHist method*), 56
plot() (*NotebookSimpleRungs method*), 56
plot() (*NotebookSimpleTimeSeries method*), 56
pop() (*TrialReportHeap method*), 28
PrintBest (*class in tune_notebook.monitors*), 56
product_grid() (*TuningParametersTemplate method*), 34
push() (*RungHeap method*), 40
push() (*TrialReportHeap method*), 28

R

Rand (*class in tune.concepts.space.parameters*), 31
RandBase (*class in tune.concepts.space.parameters*), 32
RandInt (*class in tune.concepts.space.parameters*), 32
reason (*TrialDecision property*), 25
RemoteTrialJudge (*class in tune.concepts.flow.judge*), 24
report (*RemoteTrialJudge property*), 25
report (*TrialDecision property*), 25
reports (*TrialReportCollection property*), 46
reset_log_time() (*TrialReport method*), 27
reset_monitor() (*TrialJudge method*), 26
result() (*StudyResult method*), 37
run() (*HyperoptLocalOptimizer method*), 48
run() (*IterativeObjectiveFunc method*), 41
run() (*NonIterativeObjectiveFunc method*), 43
run() (*NonIterativeObjectiveLocalOptimizer method*), 43
run() (*OptunaLocalOptimizer method*), 48
run() (*SKCVObjective method*), 49
run() (*SKObjective method*), 49
run_monitored_process() (*NonIterativeObjectiveLocalOptimizer method*), 44
run_single_iteration() (*IterativeObjectiveFunc method*), 41
run_single_rung() (*IterativeObjectiveFunc method*), 41
run_single_rung() (*KerasObjective method*), 52
rung (*IterativeObjectiveFunc property*), 42
rung (*TrialReport property*), 27
RungHeap (*class in tune.iterative.asha*), 40

S

safe_run() (*NonIterativeObjectiveFunc method*), 43
sample() (*Space method*), 36
sample() (*TuningParametersTemplate method*), 34

save_checkpoint() (*IterativeObjectiveFunc method*), 42
save_checkpoint() (*KerasObjective method*), 52
save_checkpoint() (*KerasTrainingSpec method*), 53
schedule (*ASHAJudge property*), 40
set_temp_path() (*TuneObjectFactory method*), 19
should_checkpoint (*TrialDecision property*), 25
should_stop (*TrialDecision property*), 25
should_stop() (*NonIterativeStopper method*), 45
should_stop() (*NonIterativeStopperCombiner method*), 45
should_stop() (*SimpleNonIterativeStopper method*), 46
simple_value (*TuningParametersTemplate property*), 35
SimpleNonIterativeStopper (*class in tune.noniterative.stopper*), 45
sk_space() (*in module tune_sklearn.utils*), 51
SKCVObjective (*class in tune_sklearn.objective*), 48
SKObjective (*class in tune_sklearn.objective*), 49
small_improvement() (*in module tune.noniterative.stopper*), 46
sort_metric (*TrialReport property*), 27
Space (*class in tune.concepts.space.spaces*), 35
spec (*KerasObjective property*), 52
split() (*TuneDataset method*), 38
StochasticExpression (*class in tune.concepts.space.parameters*), 32
StudyResult (*class in tune.concepts.dataset*), 37
suggest_by_continuous_asha() (*in module tune.api.suggest*), 21
suggest_by_hyperband() (*in module tune.api.suggest*), 21
suggest_by_sha() (*in module tune.api.suggest*), 22
suggest_for_noniterative_objective() (*in module tune.api.suggest*), 22
suggest_keras_models_by_continuous_asha() (*in module tune_tensorflow.suggest*), 53
suggest_keras_models_by_hyperband() (*in module tune_tensorflow.suggest*), 54
suggest_keras_models_by_sha() (*in module tune_tensorflow.suggest*), 54
suggest_sk_models() (*in module tune_sklearn.suggest*), 49
suggest_sk_models_by_cv() (*in module tune_sklearn.suggest*), 50

T

template (*TuningParametersTemplate property*), 35
test_choice() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
test_optimization() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
test_optimization_dummy() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
test_optimization_nested_param() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
test_rand() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
test_randint() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
test_transition_choice() (*NonIterativeObjectiveLocalOptimizerTests.Tests method*), 57
to_keras_spec() (*in module tune_tensorflow.utils*), 55
to_keras_spec_expr() (*in module tune_tensorflow.utils*), 55
to_noniterative_objective() (*in module tune.noniterative.convert*), 43
to_sk_model() (*in module tune_sklearn.utils*), 51
to_sk_model_expr() (*in module tune_sklearn.utils*), 51
to_template() (*in module tune.concepts.space.parameters*), 35
TransitionChoice (*class in tune.concepts.space.parameters*), 33
Trial (*class in tune.concepts.flow.trial*), 29
trial (*TrialDecision property*), 25
trial (*TrialReport property*), 27
trial_id (*Trial property*), 29
trial_id (*TrialDecision property*), 25
trial_id (*TrialReport property*), 27
TrialCallback (*class in tune.concepts.flow.judge*), 25
TrialDecision (*class in tune.concepts.flow.judge*), 25
TrialJudge (*class in tune.concepts.flow.judge*), 25
TrialReport (*class in tune.concepts.flow.report*), 26
TrialReportCollection (*class in tune.noniterative.stopper*), 46
TrialReportHeap (*class in tune.concepts.flow.report*), 28
TrialReportLogger (*class in tune.concepts.flow.report*), 28
tune.api.factory
    module, 19
tune.api.optimize
    module, 20
tune.api.suggest
    module, 21
tune.concepts.checkpoint
    module, 36
tune.concepts.dataset
    module, 37
tune.concepts.flow.judge
    module, 23
tune.concepts.flow.report
    module, 26
tune.concepts.flow.trial
    module, 29

tune.concepts.space.parameters
    module, 30
tune.concepts.space.spaces
    module, 35
tune.constants
    module, 47
tune.exceptions
    module, 47
tune.iterative.asha
    module, 40
tune.iterative.objective
    module, 41
tune.iterative.sha
    module, 42
tune.iterative.study
    module, 42
tune.noniterative.convert
    module, 43
tune.noniterative.objective
    module, 43
tune.noniterative.stopper
    module, 44
tune.noniterative.study
    module, 47
tune_hyperopt.optimizer
    module, 47
tune_notebook.monitors
    module, 55
tune_optuna.optimizer
    module, 48
tune_sklearn.objective
    module, 48
tune_sklearn.suggest
    module, 49
tune_sklearn.utils
    module, 51
tune_tensorflow.objective
    module, 51
tune_tensorflow.spec
    module, 52
tune_tensorflow.suggest
    module, 53
tune_tensorflow.utils
    module, 55
tune_test.local_optmizer
    module, 56
TuneCompileError, 47
TuneDataset (*class in tune.concepts.dataset*), 38
TuneDatasetBuilder (*class in tune.concepts.dataset*), 38
TuneInterrupted, 47
TuneObjectFactory (*class in tune.api.factory*), 19
TuneRuntimeError, 47
TuningParameterExpression (*class in tune.concepts.space.parameters*), 33
TuningParametersTemplate (*class in tune.concepts.space.parameters*), 33

U

union_with() (*StudyResult method*), 37
updated (*NonIterativeStopper property*), 45

V

validate_iterative_objective() (*in module tune.iterative.objective*), 42
validate_noniterative_objective() (*in module tune.noniterative.objective*), 44
values (*Choice property*), 30
values() (*RungHeap method*), 41
values() (*TrialReportHeap method*), 28

W

with_cost() (*TrialReport method*), 28
with_dfs() (*Trial method*), 29
with_params() (*Trial method*), 29
with_rung() (*TrialReport method*), 28
with_sort_metric() (*TrialReport method*), 28