

# Package ‘mlr3hyperband’

November 27, 2022

**Title** Hyperband for 'mlr3'

**Version** 0.4.4

**Description** Implements hyperband method for hyperparameter tuning. Various termination criteria can be set and combined. The class 'AutoTuner' provides a convenient way to perform nested resampling in combination with 'mlr3'. The hyperband algorithm was proposed by Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh and Ameet Talwalkar (2018) <[arXiv:1603.06560](https://arxiv.org/abs/1603.06560)>.

**License** LGPL-3

**URL** <https://mlr3hyperband.mlr-org.com>,  
<https://github.com/mlr-org/mlr3hyperband>

**BugReports** <https://github.com/mlr-org/mlr3hyperband/issues>

**Depends** mlr3tuning (>= 0.13.0), R (>= 3.1.0)

**Imports** bbotk (>= 0.7.0), checkmate (>= 1.9.4), data.table, lgr, mlr3 (>= 0.13.1), mlr3misc (>= 0.10.0), paradox (>= 0.9.0), R6

**Suggests** emoa, mlr3learners (>= 0.5.2), mlr3pipelines, rpart, testthat (>= 3.0.0), xgboost

**Config/testthat/edition** 3

**Config/testthat/parallel** true

**Encoding** UTF-8

**NeedsCompilation** no

**RoxygenNote** 7.2.2

**Collate** 'TunerHyperband.R' 'TunerSuccessiveHalving.R'  
'OptimizerHyperband.R' 'OptimizerSuccessiveHalving.R'  
'bibentries.R' 'helper.R' 'zzz.R'

**Author** Marc Becker [aut, cre] (<<https://orcid.org/0000-0002-8115-0400>>),  
Sebastian Gruber [aut] (<<https://orcid.org/0000-0002-8544-3470>>),  
Jakob Richter [aut] (<<https://orcid.org/0000-0003-4481-5554>>),  
Julia Moosbauer [aut] (<<https://orcid.org/0000-0002-0000-9297>>),  
Bernd Bischl [aut] (<<https://orcid.org/0000-0001-6002-6980>>)

**Maintainer** Marc Becker <marcbecker@posteo.de>

**Repository** CRAN

**Date/Publication** 2022-11-27 12:00:02 UTC

## R topics documented:

|   |    |
|---|----|
| mlr3hyperband-package . . . . .             | 2  |
| hyperband_budget . . . . .                  | 3  |
| hyperband_n_configs . . . . .               | 3  |
| hyperband_schedule . . . . .                | 4  |
| mlr_optimizers_hyperband . . . . .          | 5  |
| mlr_optimizers_successive_halving . . . . . | 8  |
| mlr_tuners_hyperband . . . . .              | 11 |
| mlr_tuners_successive_halving . . . . .     | 14 |

**Index** **18**

---

mlr3hyperband-package *mlr3hyperband: Hyperband for 'mlr3'*

---

## Description

Implements hyperband method for hyperparameter tuning. Various termination criteria can be set and combined. The class 'AutoTuner' provides a convenient way to perform nested resampling in combination with 'mlr3'. The hyperband algorithm was proposed by Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh and Ameet Talwalkar (2018) [arXiv:1603.06560](https://arxiv.org/abs/1603.06560).

## Author(s)

**Maintainer:** Marc Becker <marcbecker@posteo.de> ([ORCID](#))

Authors:

- Sebastian Gruber <gruber\_sebastian@t-online.de> ([ORCID](#))
- Jakob Richter <jakob1richter@gmail.com> ([ORCID](#))
- Julia Moosbauer <ju.moosbauer@googlemail.com> ([ORCID](#))
- Bernd Bischl <bernd\_bischl@gmx.net> ([ORCID](#))

## See Also

Useful links:

- <https://mlr3hyperband.mlr-org.com>
- <https://github.com/mlr-org/mlr3hyperband>
- Report bugs at <https://github.com/mlr-org/mlr3hyperband/issues>

---

|                  |                         |
|------------------|-------------------------|
| hyperband_budget | <i>Hyperband Budget</i> |
|------------------|-------------------------|

---

**Description**

Calculates the total budget used by hyperband.

**Usage**

```
hyperband_budget(r_min, r_max, eta, integer_budget = FALSE)
```

**Arguments**

|                |   |
|----------------|---|
| r_min          | (numeric(1))<br>Lower bound of budget parameter.  |
| r_max          | (numeric(1))<br>Upper bound of budget parameter.  |
| eta            | (numeric(1))<br>Fraction parameter of the successive halving algorithm: With every stage the configuration budget is increased by a factor of eta and only the best 1/eta points are used for the next stage. Non-integer values are supported, but eta is not allowed to be less or equal 1. |
| integer_budget | (logical(1))<br>Determines if budget is an integer.   |

**Value**

integer(1)

---

|                     |                          |
|---------------------|--------------------------|
| hyperband_n_configs | <i>Hyperband Configs</i> |
|---------------------|--------------------------|

---

**Description**

Calculates how many different configurations are sampled.

**Usage**

```
hyperband_n_configs(r_min, r_max, eta)
```

**Arguments**

|       |   |
|-------|---|
| r_min | (numeric(1))<br>Lower bound of budget parameter.  |
| r_max | (numeric(1))<br>Upper bound of budget parameter.  |
| eta   | (numeric(1))<br>Fraction parameter of the successive halving algorithm: With every stage the configuration budget is increased by a factor of eta and only the best 1/eta points are used for the next stage. Non-integer values are supported, but eta is not allowed to be less or equal 1. |

**Value**

integer(1)

---

hyperband\_schedule     *Hyperband Schedule*

---

**Description**

Returns hyperband schedule.

**Usage**

```
hyperband_schedule(r_min, r_max, eta, integer_budget = FALSE)
```

**Arguments**

|                |   |
|----------------|---|
| r_min          | (numeric(1))<br>Lower bound of budget parameter.  |
| r_max          | (numeric(1))<br>Upper bound of budget parameter.  |
| eta            | (numeric(1))<br>Fraction parameter of the successive halving algorithm: With every stage the configuration budget is increased by a factor of eta and only the best 1/eta points are used for the next stage. Non-integer values are supported, but eta is not allowed to be less or equal 1. |
| integer_budget | (logical(1))<br>Determines if budget is an integer.   |

**Value**

`data.table::data.table()`

---

mlr\_optimizers\_hyperband

*Optimizer Using the Hyperband Algorithm*


---

## Description

OptimizerHyperband class that implements hyperband optimization (HB). HB repeatedly calls SHA ([OptimizerSuccessiveHalving](#)) with different numbers of starting points. A larger number of starting points corresponds to a smaller budget allocated in the base stage. Each run of SHA within HB is called a bracket. HB considers  $s_{\max} + 1$  brackets with  $s_{\max} = \text{floor}(\log(r_{\max} / r_{\min}, \text{eta}))$ . The most explorative bracket  $s = s_{\max}$  constructs  $s_{\max} + 1$  stages and allocates the minimum budget ( $r_{\min}$ ) in the base stage. The minimum budget is increased in each bracket by a factor of  $\text{eta}$  and the number of starting points is computed so that each bracket approximately spends the same budget. Use [hyperband\\_schedule\(\)](#) to get a preview of the bracket layout.

| s | 3   |     | 2   |     | 1   |     | 0   |     |
|---|-----|-----|-----|-----|-----|-----|-----|-----|
| i | n_i | r_i | n_i | r_i | n_i | r_i | n_i | r_i |
| 0 | 8   | 1   | 6   | 2   | 4   | 4   | 8   | 4   |
| 1 | 4   | 2   | 3   | 4   | 2   | 8   |     |     |
| 2 | 2   | 4   | 1   | 8   |     |     |     |     |
| 3 | 1   | 8   |     |     |     |     |     |     |

$s$  is the bracket number,  $i$  is stage number,  $n_i$  is the number of configurations and  $r_i$  is the budget allocated to a single configuration.

The budget hyperparameter must be tagged with "budget" in the search space. The minimum budget ( $r_{\min}$ ) which is allocated in the base stage of the most explorative bracket, is set by the lower bound of the budget parameter. The upper bound defines the maximum budget ( $r_{\max}$ ) which is allocated to the candidates in the last stages.

## Dictionary

This [Optimizer](#) can be instantiated via the [dictionary mlr\\_optimizers](#) or with the associated sugar function [opt\(\)](#):

```
mlr_optimizers$get("hyperband")
opt("hyperband")
```

## Parameters

`eta` numeric(1)

With every stage, the budget is increased by a factor of  $\text{eta}$  and only the best  $1 / \text{eta}$  points are promoted to the next stage. Non-integer values are supported, but  $\text{eta}$  is not allowed to be less or equal 1.

`sampler` [paradox::Sampler](#)

Object defining how the samples of the parameter space should be drawn in the base stage of each bracket. The default is uniform sampling.

repetitions integer(1)

If 1 (default), optimization is stopped once all brackets are evaluated. Otherwise, optimization is stopped after repetitions runs of hyperband. The `bbotk::Terminator` might stop the optimization before all repetitions are executed.

### Archive

The `bbotk::Archive` holds the following additional columns that are specific to the hyperband algorithm:

- `bracket` (integer(1))  
The bracket index. Counts down to 0.
- `stage` (integer(1))  
The stages of each bracket. Starts counting at 0.
- `repetition` (integer(1))  
Repetition index. Start counting at 1.

### Custom Sampler

Hyperband supports custom `paradox::Sampler` object for initial configurations in each bracket. A custom sampler may look like this (the full example is given in the *examples* section):

```
# - beta distribution with alpha = 2 and beta = 5
# - categorical distribution with custom probabilities
sampler = SamplerJointIndep$new(list(
  Sampler1DRfun$new(params[[2]], function(n) rbeta(n, 2, 5)),
  Sampler1DCateg$new(params[[3]], prob = c(0.2, 0.3, 0.5))
))
```

### Progress Bars

`$optimize()` supports progress bars via the package `progressr` combined with a `Terminator`. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

### Logging

Hyperband uses a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

### Super class

```
bbotk::Optimizer -> OptimizerHyperband
```

### Methods

#### Public methods:

- `OptimizerHyperband$new()`
- `OptimizerHyperband$clone()`

**Method** `new()`: Creates a new instance of this R6 class.

*Usage:*

```
OptimizerHyperband$new()
```

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
OptimizerHyperband$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Source

Li L, Jamieson K, DeSalvo G, Rostamizadeh A, Talwalkar A (2018). “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization.” *Journal of Machine Learning Research*, **18**(185), 1-52. <https://jmlr.org/papers/v18/16-558.html>.

## Examples

```
library(bbotk)
library(data.table)

# set search space
search_space = domain = ps(
  x1 = p_dbl(-5, 10),
  x2 = p_dbl(0, 15),
  fidelity = p_dbl(1e-2, 1, tags = "budget")
)

# Branin function with fidelity, see `bbotk::branin()`
fun = function(xs) branin_wu(xs[["x1"]], xs[["x2"]], xs[["fidelity"]])

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = ps(y = p_dbl(tags = "minimize"))
)

# initialize instance and optimizer
instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 50)
)

optimizer = opt("hyperband")

# optimize branin function
optimizer$optimize(instance)
```

```
# best scoring evaluation
instance$result

# all evaluations
as.data.table(instance$archive)
```

---

```
mlr_optimizers_successive_halving
```

*Hyperparameter Optimization with Successive Halving*

---

## Description

OptimizerSuccessiveHalving class that implements the successive halving algorithm (SHA). SHA randomly samples  $n$  candidate points and allocates a minimum budget ( $r_{\min}$ ) to all candidates. The candidates are raced down in stages to a single best candidate by repeatedly increasing the budget by a factor of  $\eta$  and promoting only the best  $1 / \eta$  fraction to the next stage. This means promising points are allocated a higher budget overall and lower performing ones are discarded early on.

# The budget hyperparameter must be tagged with "budget" in the search space. The minimum budget ( $r_{\min}$ ) which is allocated in the base stage, is set by the lower bound of the budget parameter. The upper bound defines the maximum budget ( $r_{\max}$ ) which is allocated to the candidates in the last stage. The number of stages is computed so that each candidate in base stage is allocated the minimum budget and the candidates in the last stage are not evaluated on more than the maximum budget. The following table is the stage layout for  $\eta = 2$ ,  $r_{\min} = 1$  and  $r_{\max} = 8$ .

| $i$ | $n_i$ | $r_i$ |
|-----|-------|-------|
| 0   | 8     | 1     |
| 1   | 4     | 2     |
| 2   | 2     | 4     |
| 3   | 1     | 8     |

$i$  is stage number,  $n_i$  is the number of configurations and  $r_i$  is the budget allocated to a single configuration.

## Parameters

$n$  integer(1)

Number of points in base stage.

$\eta$  numeric(1)

With every stage, the budget is increased by a factor of  $\eta$  and only the best  $1 / \eta$  points are promoted to the next stage.

sampler [paradox::Sampler](#)

Object defining how the samples of the parameter space should be drawn. The default is uniform sampling.



repetitions integer(1)  
 If 1 (default), optimization is stopped once all stages are evaluated. Otherwise, optimization is stopped after repetitions runs of SHA. The `bbotk::Terminator` might stop the optimization before all repetitions are executed.

adjust\_minimum\_budget logical(1)  
 If TRUE, minimum budget is increased so that the last stage uses the maximum budget defined in the search space.

## Archive

The `bbotk::Archive` holds the following additional columns that are specific to the successive halving algorithm:

- stage (integer(1))  
 Stage index. Starts counting at 0.
- repetition (integer(1))  
 Repetition index. Start counting at 1.

## Custom Sampler

Hyperband supports custom `paradox::Sampler` object for initial configurations in each bracket. A custom sampler may look like this (the full example is given in the *examples* section):

```
# - beta distribution with alpha = 2 and beta = 5
# - categorical distribution with custom probabilities
sampler = SamplerJointIndep$new(list(
  Sampler1DRfun$new(params[[2]], function(n) rbeta(n, 2, 5)),
  Sampler1DCateg$new(params[[3]], prob = c(0.2, 0.3, 0.5))
))
```

## Progress Bars

`$optimize()` supports progress bars via the package `progressr` combined with a `Terminator`. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

## Logging

Hyperband uses a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

## Super class

```
bbotk::Optimizer -> OptimizerSuccessiveHalving
```

## Methods

### Public methods:

- `OptimizerSuccessiveHalving$new()`

- [OptimizerSuccessiveHalving\\$clone\(\)](#)

**Method** `new()`: Creates a new instance of this R6 class.

*Usage:*

```
OptimizerSuccessiveHalving$new()
```

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
OptimizerSuccessiveHalving$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Source

Jamieson K, Talwalkar A (2016). “Non-stochastic Best Arm Identification and Hyperparameter Optimization.” In Gretton A, Robert CC (eds.), *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 series Proceedings of Machine Learning Research, 240-248. <http://proceedings.mlr.press/v51/jamieson16.html>.

## Examples

```
library(bbotk)
library(data.table)

# set search space
search_space = domain = ps(
  x1 = p_dbl(-5, 10),
  x2 = p_dbl(0, 15),
  fidelity = p_dbl(1e-2, 1, tags = "budget")
)

# Branin function with fidelity, see `bbotk::branin()`
fun = function(xs) branin_wu(xs[["x1"]], xs[["x2"]], xs[["fidelity"]])

# create objective
objective = ObjectiveRfun$new(
  fun = fun,
  domain = domain,
  codomain = ps(y = p_dbl(tags = "minimize"))
)

# initialize instance and optimizer
instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 50)
)

optimizer = opt("successive_halving")
```

```

# optimize branin function
optimizer$optimize(instance)

# best scoring evaluation
instance$result

# all evaluations
as.data.table(instance$archive)

```

---

mlr\_tuners\_hyperband *Tuner Using the Hyperband Algorithm*


---

## Description

TunerHyperband class that implements hyperband tuning (HB). HB repeatedly calls SHA ([Tuner-SuccessiveHalving](#)) with different numbers of starting configurations. A larger number of starting configurations corresponds to a smaller budget allocated in the base stage. Each run of SHA within HB is called a bracket. HB considers  $s_{\max} + 1$  brackets with  $s_{\max} = \text{floor}(\log(r_{\max} / r_{\min}, \text{eta}))$ . The most explorative bracket  $s = s_{\max}$  constructs  $s_{\max} + 1$  stages and allocates the minimum budget ( $r_{\min}$ ) in the base stage. The minimum budget is increased in each bracket by a factor of  $\text{eta}$  and the number of starting configurations is computed so that each bracket approximately spends the same budget. Use [hyperband\\_schedule\(\)](#) to get a preview of the bracket layout.

| s | 3   |     | 2   |     | 1   |     | 0   |     |
|---|-----|-----|-----|-----|-----|-----|-----|-----|
| i | n_i | r_i | n_i | r_i | n_i | r_i | n_i | r_i |
| 0 | 8   | 1   | 6   | 2   | 4   | 4   | 8   | 4   |
| 1 | 4   | 2   | 3   | 4   | 2   | 8   |     |     |
| 2 | 2   | 4   | 1   | 8   |     |     |     |     |
| 3 | 1   | 8   |     |     |     |     |     |     |

$s$  is the bracket number,  $i$  is stage number,  $n_i$  is the number of configurations and  $r_i$  is the budget allocated to a single configuration.

The budget hyperparameter must be tagged with "budget" in the search space. The minimum budget ( $r_{\min}$ ) which is allocated in the base stage of the most explorative bracket, is set by the lower bound of the budget parameter. The upper bound defines the maximum budget ( $r_{\max}$ ) which is allocated to the candidates in the last stages.

## Subsample Budget

If the learner lacks a natural budget parameter, [mlr3pipelines::PipeOpSubsample](#) can be applied to use the subsampling rate as budget parameter. The resulting [mlr3pipelines::GraphLearner](#) is fitted on small proportions of the [mlr3::Task](#) in the first stage, and on the complete task in last stage.

## Dictionary

This [Optimizer](#) can be instantiated via the [dictionary mlr\\_optimizers](#) or with the associated sugar function `opt()`:

```
mlr_optimizers$get("hyperband")
opt("hyperband")
```

## Parameters

`eta` `numeric(1)`

With every stage, the budget is increased by a factor of `eta` and only the best  $1 / \text{eta}$  configurations are promoted to the next stage. Non-integer values are supported, but `eta` is not allowed to be less or equal 1.

`sampler` [paradox::Sampler](#)

Object defining how the samples of the parameter space should be drawn in the base stage of each bracket. The default is uniform sampling.

`repetitions` `integer(1)`

If 1 (default), optimization is stopped once all brackets are evaluated. Otherwise, optimization is stopped after `repetitions` runs of hyperband. The [bbotk::Terminator](#) might stop the optimization before all repetitions are executed.

## Archive

The [mlr3tuning::ArchiveTuning](#) holds the following additional columns that are specific to the hyperband algorithm:

- `bracket` (`integer(1)`)  
The bracket index. Counts down to 0.
- `stage` (`integer(1)`)  
The stages of each bracket. Starts counting at 0.
- `repetition` (`integer(1)`)  
Repetition index. Start counting at 1.

## Custom Sampler

Hyperband supports custom [paradox::Sampler](#) object for initial configurations in each bracket. A custom sampler may look like this (the full example is given in the *examples* section):

```
# - beta distribution with alpha = 2 and beta = 5
# - categorical distribution with custom probabilities
sampler = SamplerJointIndep$new(list(
  Sampler1DRfun$new(params[[2]], function(n) rbeta(n, 2, 5)),
  Sampler1DCateg$new(params[[3]], prob = c(0.2, 0.3, 0.5))
))
```

## Progress Bars

`$optimize()` supports progress bars via the package [progressr](#) combined with a [Terminator](#). Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package [progress](#) as backend; enable with `progressr::handlers("progress")`.

## Parallelization

This hyperband implementation evaluates hyperparameter configurations of equal budget across brackets in one batch. For example, all configurations in stage 1 of bracket 3 and stage 0 of bracket 2 in one batch. To select a parallel backend, use the `plan()` function of the **future** package.

## Logging

Hyperband uses a logger (as implemented in **lgr**) from package **bbotk**. Use `lgr::get_logger("bbotk")` to access and control the logger.

## Super classes

```
mlr3tuning::Tuner -> mlr3tuning::TunerFromOptimizer -> TunerHyperband
```

## Methods

### Public methods:

- `TunerHyperband$new()`
- `TunerHyperband$clone()`

**Method** `new()`: Creates a new instance of this R6 class.

*Usage:*

```
TunerHyperband$new()
```

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
TunerHyperband$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Source

Li L, Jamieson K, DeSalvo G, Rostamizadeh A, Talwalkar A (2018). "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." *Journal of Machine Learning Research*, **18**(185), 1-52. <https://jmlr.org/papers/v18/16-558.html>.

## Examples

```
if(requireNamespace("xgboost")) {
  library(mlr3learners)

  # define hyperparameter and budget parameter
  search_space = ps(
    nrounds = p_int(lower = 1, upper = 16, tags = "budget"),
    eta = p_dbl(lower = 0, upper = 1),
    booster = p_fct(levels = c("gbtree", "gblinear", "dart"))
  )
}
```

```

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
  method = "hyperband",
  task = tsk("pima"),
  learner = lrn("classif.xgboost", eval_metric = "logloss"),
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  search_space = search_space,
  term_evals = 100
)

# best performing hyperparameter configuration
instance$result
}

```

---

mlr\_tuners\_successive\_halving

*Hyperparameter Tuning with Successive Halving*

---

## Description

TunerSuccessiveHalving class that implements the successive halving algorithm (SHA). SHA randomly samples  $n$  candidate hyperparameter configurations and allocates a minimum budget ( $r_{\min}$ ) to all candidates. The candidates are raced down in stages to a single best candidate by repeatedly increasing the budget by a factor of  $\eta$  and promoting only the best  $1 / \eta$  fraction to the next stage. This means promising hyperparameter configurations are allocated a higher budget overall and lower performing ones are discarded early on.

The budget hyperparameter must be tagged with "budget" in the search space. The minimum budget ( $r_{\min}$ ) which is allocated in the base stage, is set by the lower bound of the budget parameter. The upper bound defines the maximum budget ( $r_{\max}$ ) which is allocated to the candidates in the last stage. The number of stages is computed so that each candidate in base stage is allocated the minimum budget and the candidates in the last stage are not evaluated on more than the maximum budget. The following table is the stage layout for  $\eta = 2$ ,  $r_{\min} = 1$  and  $r_{\max} = 8$ .

| $i$ | $n_i$ | $r_i$ |
|-----|-------|-------|
| 0   | 8     | 1     |
| 1   | 4     | 2     |
| 2   | 2     | 4     |
| 3   | 1     | 8     |

$i$  is stage number,  $n_i$  is the number of configurations and  $r_i$  is the budget allocated to a single configuration.

### Subsample Budget

If the learner lacks a natural budget parameter, `mlr3pipelines::PipeOpSubsample` can be applied to use the subsampling rate as budget parameter. The resulting `mlr3pipelines::GraphLearner` is fitted on small proportions of the `mlr3::Task` in the first stage, and on the complete task in last stage.

### Parameters

- `n` `integer(1)`  
Number of candidates in base stage.
- `eta` `numeric(1)`  
With every stage, the budget is increased by a factor of `eta` and only the best  $1 / \eta$  candidates are promoted to the next stage. Non-integer values are supported, but `eta` is not allowed to be less or equal 1.
- `sampler` `paradox::Sampler`  
Object defining how the samples of the parameter space should be drawn. The default is uniform sampling.
- `repeats` `logical(1)`  
If `FALSE` (default), SHA terminates once all stages are evaluated. Otherwise, SHA starts over again once the last stage is evaluated.
- `adjust_minimum_budget` `logical(1)`  
If `TRUE`, minimum budget is increased so that the last stage uses the maximum budget defined in the search space.

### Archive

The `mlr3tuning::ArchiveTuning` holds the following additional columns that are specific to the successive halving algorithm:

- `stage` (`integer(1)`)  
Stage index. Starts counting at 0.
- `repetition` (`integer(1)`)  
Repetition index. Start counting at 1.

### Custom Sampler

Hyperband supports custom `paradox::Sampler` object for initial configurations in each bracket. A custom sampler may look like this (the full example is given in the *examples* section):

```
# - beta distribution with alpha = 2 and beta = 5
# - categorical distribution with custom probabilities
sampler = SamplerJointIndep$new(list(
  Sampler1DRfun$new(params[[2]], function(n) rbeta(n, 2, 5)),
  Sampler1DCateg$new(params[[3]], prob = c(0.2, 0.3, 0.5))
))
```

### Progress Bars

`$optimize()` supports progress bars via the package **progressr** combined with a **Terminator**. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package **progress** as backend; enable with `progressr::handlers("progress")`.

### Parallelization

The hyperparameter configurations of one stage are evaluated in parallel with the **future** package. To select a parallel backend, use the `plan()` function of the **future** package.

### Logging

Hyperband uses a logger (as implemented in **lgr**) from package **bbotk**. Use `lgr::get_logger("bbotk")` to access and control the logger.

### Super classes

```
mlr3tuning::Tuner -> mlr3tuning::TunerFromOptimizer -> TunerSuccessiveHalving
```

### Methods

#### Public methods:

- `TunerSuccessiveHalving$new()`
- `TunerSuccessiveHalving$clone()`

**Method** `new()`: Creates a new instance of this R6 class.

*Usage:*

```
TunerSuccessiveHalving$new()
```

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
TunerSuccessiveHalving$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

### Source

Jamieson K, Talwalkar A (2016). “Non-stochastic Best Arm Identification and Hyperparameter Optimization.” In Gretton A, Robert CC (eds.), *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 series Proceedings of Machine Learning Research, 240-248. <http://proceedings.mlr.press/v51/jamieson16.html>.



**Examples**

```
if(requireNamespace("xgboost")) {  
  library(mlr3learners)  
  
  # define hyperparameter and budget parameter  
  search_space = ps(  
    nrounds = p_int(lower = 1, upper = 16, tags = "budget"),  
    eta = p_dbl(lower = 0, upper = 1),  
    booster = p_fct(levels = c("gbtree", "gblinear", "dart"))  
  )  
  
  # hyperparameter tuning on the pima indians diabetes data set  
  instance = tune(  
    method = "successive_halving",  
    task = tsk("pima"),  
    learner = lrn("classif.xgboost", eval_metric = "logloss"),  
    resampling = rsmpl("cv", folds = 3),  
    measures = msr("classif.ce"),  
    search_space = search_space,  
    term_evals = 100  
  )  
  
  # best performing hyperparameter configuration  
  instance$result  
}
```

# Index

bbotk::Archive, [6](#), [9](#)  
bbotk::Optimizer, [6](#), [9](#)  
bbotk::Terminator, [6](#), [9](#), [12](#)

data.table::data.table(), [4](#)  
dictionary, [5](#), [12](#)

hyperband\_budget, [3](#)  
hyperband\_n\_configs, [3](#)  
hyperband\_schedule, [4](#)  
hyperband\_schedule(), [5](#), [11](#)

mlr3::Task, [11](#), [15](#)  
mlr3hyperband (mlr3hyperband-package), [2](#)  
mlr3hyperband-package, [2](#)  
mlr3pipelines::GraphLearner, [11](#), [15](#)  
mlr3pipelines::PipeOpSubsample, [11](#), [15](#)  
mlr3tuning::ArchiveTuning, [12](#), [15](#)  
mlr3tuning::Tuner, [13](#), [16](#)  
mlr3tuning::TunerFromOptimizer, [13](#), [16](#)  
mlr\_optimizers, [5](#), [12](#)  
mlr\_optimizers\_hyperband, [5](#)  
mlr\_optimizers\_successive\_halving, [8](#)  
mlr\_tuners\_hyperband, [11](#)  
mlr\_tuners\_successive\_halving, [14](#)

opt(), [5](#), [12](#)  
Optimizer, [5](#), [12](#)  
OptimizerHyperband  
    (mlr\_optimizers\_hyperband), [5](#)  
OptimizerSuccessiveHalving, [5](#)  
OptimizerSuccessiveHalving  
    (mlr\_optimizers\_successive\_halving),  
    [8](#)

paradox::Sampler, [5](#), [6](#), [8](#), [9](#), [12](#), [15](#)

R6, [7](#), [10](#), [13](#), [16](#)

Terminator, [6](#), [9](#), [12](#), [16](#)

TunerHyperband (mlr\_tuners\_hyperband),  
    [11](#)  
TunerSuccessiveHalving, [11](#)  
TunerSuccessiveHalving  
    (mlr\_tuners\_successive\_halving),  
    [14](#)