Package 'mlr3hyperband'

September 13, 2021

Title Hyperband for 'mlr3'

Version 0.2.0

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>.

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 (>= 0.6.0), R (>= 3.1.0)

Imports bbotk (>= 0.3.0), checkmate (>= 1.9.4), data.table, lgr, mlr3 (>= 0.7.0), mlr3misc (>= 0.7.0), paradox (>= 0.7.0), R6

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

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.1.1

Collate 'TunerHyperband.R' 'TunerSuccessiveHalving.R' 'OptimizerHyperband.R' 'OptimizerSuccessiveHalving.R' 'nds_selection.R' 'bibentries.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 2021-09-13 13:30:02 UTC

R topics documented:

`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>.

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

`mlr_optimizers_hyperband` *Optimizer using the Hyperband algorithm*

Description

`OptimizerHyperband` class that implements hyperband optimization. Hyperband is a budget oriented-procedure, weeding out suboptimal performing configurations early in a sequential training process, increasing optimization efficiency as a consequence.

For this, several brackets are constructed with an associated set of configurations for each bracket. Each bracket as several stages. Different brackets are initialized with different amounts of configurations and different budget sizes. To get an idea of how the bracket layout looks like for a given argument set, please have a look in the `details`.

To identify the budget for evaluating hyperband, the user has to specify explicitly which parameter of the objective function influences the budget by tagging a single parameter in the paradox::ParamSet with `"budget"`.

Naturally, hyperband terminates once all of its brackets are evaluated, so a bbotk::Terminator in the OptimInstanceSingleCrit | OptimInstanceMultiCrit acts as an upper bound and should be only set to a low value if one is unsure of how long hyperband will take to finish under the given settings.

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)`
: Fraction parameter of the successive halving algorithm: With every step the configuration budget is increased by a factor of `eta` and only the best `1/eta` configurations are used for 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 during the initialization of each bracket. The default is uniform sampling.

Archive

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

- `bracket (integer(1))`
  The console logs about the bracket index are actually not matching with the original hyperband algorithm, which counts down the brackets and stops after evaluating bracket 0. The true bracket indices are given in this column.

- `bracket_stage` (integer(1))
  The bracket stage of each bracket. Hyperband starts counting at 0.
- `budget_scaled` (`numeric(1)`)
  The intermediate budget in each bracket stage calculated by hyperband. Because hyperband is originally only considered for budgets starting at 1, some rescaling is done to allow budgets starting at different values. For this, budgets are internally divided by the lower budget bound to get a lower budget of 1. Before the objective function receives its budgets for evaluation, the budget is transformed back to match the original scale again.
- `budget_real` (`numeric(1)`)
  The real budget values the objective function uses for evaluation after hyperband calculated its scaled budget.
- `n_configs` (`integer(1)`)
  The amount of evaluated configurations in each stage. These correspond to the `r_i` in the original paper.

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))
))
```

Runtime

The calculation of each bracket currently assumes a linear runtime in the chosen budget parameter is always given. Hyperband is designed so each bracket requires approximately the same runtime as the sum of the budget over all configurations in each bracket is roughly the same. This will not hold true once the scaling in the budget parameter is not linear anymore, even though the sum of the budgets in each bracket remains the same. A possible adaption would be to introduce a trafo, like it is shown in the *examples* section.

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

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. The points of one stage in a bracket are evaluated in one batch. Parallelization is supported via package **future** (see `mlr3::benchmark()`'s section on parallelization for more details).

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)

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

# modified branin function
objective = ObjectiveRFunDt$new(
  fun = function(xdt) {
    a = 1
    b = 5.1 / (4 * (pi ^ 2))
    c = 5 / pi
    r = 6
    s = 10
```

```
    t = 1 / (8 * pi)
    data.table(y =
      (a * ((xdt[["x2"]] -
      b * (xdt[["x1"]] ^ 2L) +
      c * xdt[["x1"]] - r) ^ 2) +
      ((s * (1 - t)) * cos(xdt[["x1"]])) +
      s - (5 * xdt[["fidelity"]] * xdt[["x1"]])))
  },
  domain = domain,
  codomain = ps(y = p_dbl(tags = "minimize"))
)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("none")
)

optimizer = opt("hyperband")

# modifies the instance by reference
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. The algorithm samples `n` points and evaluates them with the smallest budget (lower bound of the `budget` parameter). 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. The optimization terminates when the maximum budget is reached (upper bound of the `budget` parameter).

To identify the budget, the user has to specify explicitly which parameter of the objective function influences the budget by tagging a single parameter in the search_space (paradox::ParamSet) with `"budget"`.

Parameters

`n integer(1)`
: Number of points in first stage.

`eta` `numeric(1)`
: With every stage, the point 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.

`sampler` paradox::Sampler
: Object defining how the samples of the parameter space should be drawn during the initialization of each bracket. The default is uniform sampling.

Archive

The bbotk::Archive holds the following additional column that is specific to the successive halving algorithm:

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

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))
))
```

Runtime

The calculation of each bracket currently assumes a linear runtime in the chosen budget parameter is always given. Hyperband is designed so each bracket requires approximately the same runtime as the sum of the budget over all configurations in each bracket is roughly the same. This will not hold true once the scaling in the budget parameter is not linear anymore, even though the sum of the budgets in each bracket remains the same. A possible adaption would be to introduce a trafo, like it is shown in the *examples* section.

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

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. The points of one stage in a bracket are evaluated in one batch. Parallelization is supported via package **future** (see `mlr3::benchmark()`'s section on parallelization for more details).

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)

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

# modified branin function
objective = ObjectiveRFunDt$new(
  fun = function(xdt) {
    a = 1
    b = 5.1 / (4 * (pi ^ 2))
    c = 5 / pi
    r = 6
```

```
    s = 10
    t = 1 / (8 * pi)
    data.table(y =
      (a * ((xdt[["x2"]] -
      b * (xdt[["x1"]] ^ 2L) +
      c * xdt[["x1"]] - r) ^ 2) +
      ((s * (1 - t)) * cos(xdt[["x1"]])) +
      s - (5 * xdt[["fidelity"]] * xdt[["x1"]])))
  },
  domain = domain,
  codomain = ps(y = p_dbl(tags = "minimize"))
)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("none")
)

optimizer = opt("successive_halving")

# modifies the instance by reference
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. Hyperband is a budget oriented-procedure, weeding out suboptimal performing configurations early in a sequential training process, increasing tuning efficiency as a consequence.

For this, several brackets are constructed with an associated set of configurations for each bracket. Each bracket has several stages. Different brackets are initialized with different amounts of configurations and different budget sizes.

Within the context of hyperband each evaluation of a learner consumes a certain budget. This budget is not fixed but controlled by a certain hyperparameter, e.g. the number of boosting iterations or the number of trees in a random forest. The user has to specify explicitly which hyperparameter of the learner controls the consumption of the budget by tagging a single hyperparameter in the paradox::ParamSet with `"budget"`. An alternative approach using subsampling and pipelines is described below.

Naturally, hyperband terminates once all of its brackets are evaluated, so a bbotk::Terminator in the tuning instance acts as an upper bound and should be only set to a low value if one is unsure of how long hyperband will take to finish under the given settings.

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)`
: Fraction parameter of the successive halving algorithm: With every step the configuration budget is increased by a factor of `eta` and only the best `1/eta` configurations are used for 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 during the initialization of each bracket. The default is uniform sampling.

Archive

The mlr3tuning::ArchiveTuning holds the following additional columns that are specific to the hyperband tuner:

- `bracket` (`integer(1)`)
  The console logs about the bracket index are actually not matching with the original hyperband algorithm, which counts down the brackets and stops after evaluating bracket 0. The true bracket indices are given in this column.
- `bracket_stage` (integer(1))
  The bracket stage of each bracket. Hyperband starts counting at 0.
- `budget_scaled` (`numeric(1)`)
  The intermediate budget in each bracket stage calculated by hyperband. Because hyperband is originally only considered for budgets starting at 1, some rescaling is done to allow budgets starting at different values. For this, budgets are internally divided by the lower budget bound to get a lower budget of 1. Before the learner receives its budgets for evaluation, the budget is transformed back to match the original scale again.
- `budget_real` (`numeric(1)`)
  The real budget values the learner uses for evaluation after hyperband calculated its scaled budget.
- `n_configs` (`integer(1)`)
  The amount of evaluated configurations in each stage. These correspond to the `r_i` in the original paper.

Hyperband without learner budget

Thanks to **mlr3pipelines**, it is possible to use hyperband in combination with learners lacking a natural budget parameter. For example, any mlr3::Learner can be augmented with a mlr3pipelines::PipeOp operator such as mlr3pipelines::PipeOpSubsample. With the subsampling rate as budget parameter, the resulting mlr3pipelines::GraphLearner is fitted on small proportions of the mlr3::Task in the first brackets, and on the complete Task in last brackets. See examples for some code.

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))
))
```

Runtime

The calculation of each bracket currently assumes a linear runtime in the chosen budget parameter is always given. Hyperband is designed so each bracket requires approximately the same runtime as the sum of the budget over all configurations in each bracket is roughly the same. This will not hold true once the scaling in the budget parameter is not linear anymore, even though the sum of the budgets in each bracket remains the same. A possible adaption would be to introduce a trafo, like it is shown in the *examples* section.

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

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. The points of one stage in a bracket are evaluated in one batch. Parallelization is supported via package **future** (see `mlr3::benchmark()`'s section on parallelization for more details).

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),
  measure = msr("classif.ce"),
  search_space = search_space
)

# best performing hyperparameter configuration
instance$result

}
```

`mlr_tuners_successive_halving` *Hyperparameter Tuning with Successive Halving*

Description

`TunerSuccessiveHalving` class that implements the successive halving algorithm. The algorithm samples `n` configurations and evaluates them with the smallest budget (lower bound of the `budget` parameter). With every stage the budget is increased by a factor of `eta` and only the best `1/eta` configurations are promoted to the next stage. The optimization terminates when the maximum budget is reached (upper bound of the `budget` parameter).

To identify the budget, the user has to specify explicitly which parameter of the objective function influences the budget by tagging a single parameter in the search_space (paradox::ParamSet) with `"budget"`.

Parameters

`n integer(1)`
: Number of configurations in first stage.

`eta numeric(1)`
: With every step, the configuration budget is increased by a factor of `eta` and only the best `1/eta` configurations are used for 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 during the initialization of each bracket. The default is uniform sampling.

Archive

The mlr3tuning::ArchiveTuning holds the following additional column that is specific to the successive halving algorithm:

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

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 = rsmp("cv", folds = 3),
  measure = msr("classif.ce"),
  search_space = search_space
)

# best performing hyperparameter configuration
instance$result

}
```

nds_selection *Best points w.r.t. non dominated sorting with hypervolume contrib.*

Description

Select best subset of points by non dominated sorting with hypervolume contribution for tie breaking. Works on an arbitrary dimension of size two or higher. Returns a vector of indices of selected points.

Value

`integer()`

Parameters

`points matrix()`
: Numeric matrix with each column corresponding to a point.

`n_select integer(1)`
: Amount of points to select.

`ref_point integer()`
: Reference point for hypervolume.

`minimize logical()`
: Should the ranking be based on minimization? (Single bool for all dimensions, or vector of bools with each entry corresponding to each dimension).

Index