Package 'thangldKANr'

November 20, 2026

Type Package

Title Kolmogorov-Arnold Networks with Interchangeable Edge Bases Using 'torch'

Version 0.1.0

Description Implements Kolmogorov-Arnold Networks (KAN) in R on top of the 'torch' package (no 'Python' needed) with one network engine and interchangeable edge bases: Chebyshev, Legendre and Jacobi polynomials, Gaussian radial basis functions (FastKAN), Mexican hat wavelets (Bozorgasl and Chen (2024) <doi:10.48550/arXiv.2405.12832>), safe Pade rational functions (Molina, Schramowski and Kersting (2020) <doi:10.48550/arXiv.1907.06732>), a hybrid of SiLU, Chebyshev and radial basis parts, and squared rectified linear unit bumps (Qiu et al. (2024) <doi:10.48550/arXiv.2406.02075>). Each layer may use a different basis, and further bases can be registered by other packages. Besides tabular regression, the same layers power graph KANs for node regression (neighbourhood aggregation with the normalized adjacency matrix, as in Kiamari, Kiamari and Krishnamachari (2024) <doi:10.48550/arXiv.2406.06470>) and lag-window KANs for one-step time series forecasting. Provides multi-layer networks with Adam training, training-only standardization, validation-based early stopping, reproducible seeds, edge-function evaluation and plots, and model persistence. A base R reference implementation with analytic gradients serves as a numerical oracle and as a training backend that does not need 'libtorch'. The network follows Liu et al. (2024) <doi:10.48550/arXiv.2404.19756>.

License GPL-3

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports graphics, stats, torch (>= 0.13.0), utils

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/Thangpro9999/thangldKANr

BugReports https://github.com/Thangpro9999/thangldKANr/issues

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Thang Le-Dinh [aut, cre, cph]

Maintainer Thang Le-Dinh <thang87sg@gmail.com>

Repository CRAN

Date/Publication 2026-11-20 13:10:02 UTC

Contents

chebyshev_basis . . . 3
hybrid_basis . . . 3
hybrid_kan_components . . . 4
jacobi_basis . . . 5
kan_bases . . . 6
kan_layer_bases . . . 6
legendre_basis . . . 7
mexican_hat . . . 7
plot.thangld_kan . . . 8
predict.thangld_graph_kan . . . 9
predict.thangld_kan . . . 10
predict.thangld_temporal_kan . . . 11
rational . . . 11
rbf_basis . . . 12
register_kan_basis . . . 13
relu_basis . . . 14
thangld_graph_kan . . . 15
thangld_graph_kan_adjacency . . . 17
thangld_graph_kan_fit . . . 18
thangld_graph_kan_forward_r . . . 19
thangld_graph_kan_metrics . . . 20
thangld_graph_kan_save . . . 21
thangld_graph_kan_simulate . . . 21
thangld_graph_kan_validate . . . 22
thangld_kan . . . 23
thangld_kan_edge . . . 24
thangld_kan_fit . . . 25
thangld_kan_forward_r . . . 27
thangld_kan_metrics . . . 28
thangld_kan_save . . . 28
thangld_kan_validate . . . 29
thangld_temporal_kan . . . 30
thangld_temporal_kan_embed . . . 31
thangld_temporal_kan_fit . . . 32

thangld_temporal_kan_rolling 33
thangld_temporal_kan_save 34
thangld_temporal_kan_validate 35

Index **36**

`chebyshev_basis` *Chebyshev polynomials and their derivatives*

Description

`chebyshev_basis()` evaluates $T_0(z), \ldots, T_K(z)$ by the recurrence $T_0 = 1$, $T_1 = z$, $T_{k+1}(z) = 2zT_k(z) - T_{k-1}(z)$. `chebyshev_basis_deriv()` returns $T_k'(z) = kU_{k-1}(z)$ with the Chebyshev polynomials of the second kind U, which has no singularity at $z = \pm 1$ (unlike the $\arccos$ form).

Usage

```
chebyshev_basis(z, degree)

chebyshev_basis_deriv(z, degree)
```

Arguments

`z`	Numeric vector.
`degree`	Maximum degree $K \geq 0$.

Value

A `length(z)` by `degree + 1` matrix (column `k + 1` = degree `k`).

Examples

```
chebyshev_basis(c(-1, 0, 0.5, 1), 3)
chebyshev_basis_deriv(c(-1, 0, 1), 3)
```

`hybrid_basis` *Basis functions of a hybrid KAN edge*

Description

`hybrid_basis()` returns the features one edge combines: $\mathrm{silu}(x) = x/(1 + e^{-x})$, the Chebyshev polynomials $T_0, \ldots, T_K$ of $\tanh(x)$ (as in ChebyKAN) and the Gaussian radial basis functions $\exp(-((x - \mu_m)/d)^2)$ on a fixed grid (as in FastKAN). `hybrid_basis_deriv()` returns their derivatives with respect to x, using $T_{k+1}' = 2T_k + 2zT_k' - T_{k-1}'$ (no singularity at $z = \pm 1$) and the chain rule through $\tanh$.

Usage

```
hybrid_basis(
  x,
  degree = 3L,
  num_grids = 8L,
  grid_min = -2,
  grid_max = 2,
  denominator = NULL
)

hybrid_basis_deriv(
  x,
  degree = 3L,
  num_grids = 8L,
  grid_min = -2,
  grid_max = 2,
  denominator = NULL
)
```

Arguments

`x`	Numeric vector.
`degree`	Chebyshev degree $K \geq 0$.
`num_grids`	Number of RBF centers $G \geq 2$.
`grid_min`, `grid_max`	Range of the RBF centers.
`denominator`	RBF width d; `NULL` gives the center spacing.

Value

A matrix with columns `silu`, `T0`..`TK` and `rbf1`..`rbfG`.

Examples

```
hybrid_basis(c(-1, 0, 1), degree = 2, num_grids = 3)
```

---

hybrid_kan_components *Size of each basis part on data*

Description

For every edge of one layer, evaluates the SiLU, Chebyshev and RBF parts at the layer inputs produced by `x` and reports the standard deviation of each part and its share of the summed standard deviations. This is a description of the fitted coefficients (the parts are not orthogonal; see `thangld_kan()`), not a variable-importance or causal measure.

Usage

```
hybrid_kan_components(model, x, layer = 1L)
```

Arguments

model	A (fitted) thangld_kan whose layer layer uses basis = "hybrid".
x	Predictor matrix or data frame on the original scale (columns as in the fit).
layer	Layer index.

Value

A data frame with one row per edge and part: input, output, part, sd, share.

Examples

```
set.seed(1)
x <- matrix(runif(200, -2, 2), 100, 2, dimnames = list(NULL, c("a", "b")))
fit <- thangld_kan_fit(thangld_kan(basis = "hybrid", c(2, 1), seed = 1), x, sin(x[, 1]) + x[, 2],
                       epochs = 20, backend = "r", seed = 1)
hybrid_kan_components(fit, x)
```

---

jacobi_basis *Jacobi polynomials and their derivatives*

Description

`jacobi_basis()` evaluates $P_0^{(\alpha,\beta)}(z), \ldots, P_K^{(\alpha,\beta)}(z)$ by the three-term recurrence

$$2n(n+\alpha+\beta)(2n+\alpha+\beta-2)P_n = (2n+\alpha+\beta-1)\{(2n+\alpha+\beta)(2n+\alpha+\beta-2)z+\alpha^2-\beta^2\}P_{n-1}-2(n+\alpha-1)(n+\beta-1)(2n+$$

with $P_0 = 1$ and $P_1 = \{(\alpha-\beta)+(\alpha+\beta+2)z\}/2$ (standard normalization, $P_n(1) = \binom{n+\alpha}{n}$). `jacobi_basis_deriv()` uses $\frac{d}{dz}P_n^{(\alpha,\beta)} = \frac{n+\alpha+\beta+1}{2}P_{n-1}^{(\alpha+1,\beta+1)}$. Special cases: $\alpha = \beta = 0$ gives Legendre polynomials; $\alpha = \beta = -1/2$ gives Chebyshev polynomials up to a constant factor per degree.

Usage

```
jacobi_basis(z, degree, alpha = 1, beta = 1)

jacobi_basis_deriv(z, degree, alpha = 1, beta = 1)
```

Arguments

z	Numeric vector.
degree	Maximum degree $K \geq 0$.
alpha, beta	Jacobi parameters, both > -1.

Value

A `length(z)` by `degree + 1` matrix.

Examples

```
jacobi_basis(c(-1, 0, 1), 3, alpha = 1, beta = 1)
```

---

kan_bases *List the available edge bases*

Description

List the available edge bases

Usage

```
kan_bases()
```

Value

A data frame with one row per basis: `name` (the value for the `basis` argument of `thangld_kan()`), `label`, `builtin` and `arguments` (the basis-specific arguments accepted by `thangld_kan()`).

Examples

```
kan_bases()
```

---

kan_layer_bases *Bases used by the layers of a model*

Description

Bases used by the layers of a model

Usage

```
kan_layer_bases(model)
```

Arguments

`model`	A `thangld_kan` object.

Value

Character vector with one basis name per layer.

Examples

```
kan_layer_bases(thangld_kan(c(2, 3, 1), basis = c("rbf", "jacobi"), seed = 1))
```

legendre_basis *Legendre polynomials and their derivatives*

Description

`legendre_basis()` evaluates $P_0(z), \ldots, P_K(z)$ by Bonnet's recurrence $P_0 = 1$, $P_1 = z$, $(k+1)P_{k+1}(z) = (2k+1)zP_k(z) - kP_{k-1}(z)$. `legendre_basis_deriv()` returns $P_k'(z)$ from $P_{k+1}' = P_{k-1}' + (2k+1)P_k$, $P_0' = 0$, $P_1' = 1$, which has no singularity at $z = \pm 1$.

Usage

```
legendre_basis(z, degree)

legendre_basis_deriv(z, degree)
```

Arguments

`z`	Numeric vector.
`degree`	Maximum degree $K \geq 0$.

Details

Base R only; this is the numerical reference for the torch layer. The polynomials are orthogonal on $[-1, 1]$ with unit weight; that does not make the features $P_k(\tanh(x))$ orthogonal on a sample.

Value

A `length(z)` by `degree + 1` matrix (column `k + 1` = degree `k`).

Examples

```
legendre_basis(c(-1, 0, 0.5, 1), 3)
legendre_basis_deriv(c(-1, 0, 1), 3)
```

mexican_hat *Mexican hat (Ricker) wavelet*

Description

`mexican_hat()` evaluates $\psi(u) = C(1-u^2)e^{-u^2/2}$ with $C = 2/(\sqrt{3}\pi^{1/4})$, so that $\int \psi^2 = 1$ and $\int \psi = 0$; `mexican_hat_deriv()` returns $\psi'(u) = Ce^{-u^2/2}(u^3 - 3u)$.

Usage

```
mexican_hat(u)

mexican_hat_deriv(u)
```

Arguments

u	Numeric vector.

Details

On an edge the wavelet is used as $w\,\psi((x-t)/s)$ without the $1/\sqrt{s}$ factor of the continuous wavelet transform (the same convention as Wav-KAN); the learned weight w absorbs the amplitude. Wav-KAN codes the wavelet with the opposite sign, $C(u^2-1)e^{-u^2/2}$, which corresponds to $-w$.

Value

Numeric vector.

Examples

```
mexican_hat(c(-1, 0, 1))
stats::integrate(function(u) mexican_hat(u)^2, -Inf, Inf)$value
```

---

plot.thangld_kan *Plot a Kolmogorov-Arnold network*

Description

`type = "edge"` draws one learned edge function (see `thangld_kan_edge()`); `type = "loss"` draws the loss history of a fitted model.

Usage

```
## S3 method for class 'thangld_kan'
plot(
  x,
  type = c("edge", "loss"),
  layer = 1L,
  input = 1L,
  output = 1L,
  scale = c("model", "original"),
  ...
)
```

Arguments

x	A model.
type	`"edge"` or `"loss"`.
layer, input, output, scale	Passed to `thangld_kan_edge()`.
...	Graphical parameters passed to `graphics::plot()`.

Value

x, invisibly.

Examples

```
plot(thangld_kan(c(1, 1), seed = 1))
```

`predict.thangld_graph_kan` *Predict node targets*

Description

Predict node targets

Usage

```
## S3 method for class 'thangld_graph_kan'
predict(object, newdata = NULL, graph = NULL, backend = c("r", "torch"), ...)
```

Arguments

`object`	A fitted `thangld_graph_kan_fit()` model.
`newdata`	`NULL` (the fitted graph and features: predictions for all nodes, transductive) or a new node feature matrix, used with `graph`.
`graph`	Graph for `newdata` (adjacency matrix, edge list or `NULL` for no graph); ignored when `newdata` is `NULL`.
`backend`	`"r"` or `"torch"`.
`...`	Unused.

Value

Numeric vector, one prediction per node, on the original scale.

`predict.thangld_kan` *Predict with a fitted Kolmogorov-Arnold network*

Description

Applies the stored training preprocessing, evaluates the network and returns predictions on the original scale of the response. Columns of `newdata` are matched by name when the model was trained with names (order does not matter, extra columns are ignored, missing ones are an error).

Usage

```
## S3 method for class 'thangld_kan'
predict(
  object,
  newdata,
  backend = c("r", "torch"),
  dtype = c("float64", "float32"),
  ...
)
```

Arguments

`object`	A fitted model.
`newdata`	Numeric matrix, data frame or vector of predictors.
`backend`	`"r"` (default, base R, no libtorch needed) or `"torch"`.
`dtype`	Floating point type for `backend = "torch"`.
`...`	Unused.

Value

Numeric vector of predictions.

Examples

```
set.seed(2)
x <- matrix(runif(100, -1, 1), ncol = 1, dimnames = list(NULL, "x1"))
fit <- thangld_kan_fit(thangld_kan(c(1, 1), seed = 1), x, x[, 1]^3, epochs = 20,
             lr = 0.02, backend = "r")
predict(fit, data.frame(x1 = c(-0.5, 0, 0.5)))
```

predict.thangld_temporal_kan *One-step-ahead forecast*

Description

Forecasts the value following the end of `newdata` (default: the series used in fitting) with the fitted network and the stored training standardization. Only one-step forecasts are supported.

Usage

```
## S3 method for class 'thangld_temporal_kan'
predict(object, newdata = NULL, h = 1L, backend = c("r", "torch"), ...)
```

Arguments

`object`	A fitted `thangld_temporal_kan`.
`newdata`	Optional series (history up to the forecast origin); at least `max(lags)` (+1 with `difference`) values.
`h`	Forecast horizon; must be 1.
`backend`	`"r"` (default) or `"torch"`.
`...`	Unused.

Value

A single forecast on the original scale of the series.

Examples

```
y <- as.numeric(datasets::lynx)
fit <- thangld_temporal_kan_fit(thangld_temporal_kan(1:3, seed = 1), log(y[1:100]),
                          epochs = 30, lr = 0.02, backend = "r", seed = 1)
c(forecast = predict(fit, log(y[1:100])), actual = log(y[101]))
```

rational *Safe Pade rational function*

Description

Evaluates the "safe" Pade form used by rational activation functions (Molina, Schramowski and Kersting, 2020, version B) and by the group rational KAN of Yang and Wang (2025):

$$R(x) = \frac{a_0 + a_1 x + \cdots + a_m x^m}{1 + |b_1 x + \cdots + b_n x^n|}.$$

The denominator is at least 1, so R has no poles. For $m = n + 1$ the function grows linearly in $|x|$. `rational_deriv()` returns $R'(x)$ (with $d|u|/du = \mathrm{sign}(u)$, 0 at $u = 0$, as in torch).

Usage

```
rational(x, num, den)

rational_deriv(x, num, den)
```

Arguments

x	Numeric vector.
num	Numerator coefficients $a_0, \ldots, a_m$.
den	Denominator coefficients $b_1, \ldots, b_n$ (n <= m).

Value

Numeric vector of length(x).

Examples

```
rational(c(-2, 0, 2), num = c(0, 1, 0, 0.5), den = c(0, 0.5))
rational_deriv(c(-2, 0, 2), num = c(0, 1, 0, 0.5), den = c(0, 0.5))
```

---

rbf_basis *Gaussian radial basis functions*

Description

rbf_basis() evaluates $b_m(u) = \exp(-((u - c_m)/d)^2)$ for centers centers and width (denominator) d; rbf_basis_deriv() returns $db_m/du = -2(u - c_m)/d^2\, b_m(u)$. rbf_grid() returns the FastKAN default grid: num_grids equally spaced centers on [grid_min, grid_max] and d = (grid_max - grid_min) / (num_grids - 1).

Usage

```
rbf_basis(u, centers, d)

rbf_basis_deriv(u, centers, d)

rbf_grid(grid_min = -2, grid_max = 2, num_grids = 8L)
```

Arguments

u	Numeric vector.
centers	Numeric vector of centers.
d	Positive width.
grid_min, grid_max, num_grids	Grid definition.

Value

`rbf_basis()`/`rbf_basis_deriv()`: a `length(u)` by `length(centers)` matrix. `rbf_grid()`: a list with `centers` and `d`.

Examples

```
g <- rbf_grid(-2, 2, 8)
g
round(rbf_basis(c(-2, 0, 2), g$centers, g$d), 3)
```

`register_kan_basis` *Register an additional edge basis*

Description

Adds an edge type to the registry so that `thangld_kan(basis = name)` builds networks with it. Every other function of the package (fitting with both backends, prediction, edge plots, save/load and `thangld_kan_validate()`) then works with the new basis unchanged. This is how other packages plug in their basis (for example a B-spline basis as `"bspline"`), typically from their `.onLoad()`.

Usage

```
register_kan_basis(name, impl, overwrite = FALSE)

unregister_kan_basis(name)
```

Arguments

`name`	Basis name: lower-case letters, digits and _, starting with a letter. Built-in names cannot be replaced.
`impl`	Named list of functions described in Details.
`overwrite`	Replace an earlier registration of `name`.

Details

`impl` is a named list of functions. Required:

`layer_specs(widths, ...)` Checks the basis arguments and returns a list with one specification (a list) per layer. Its formal arguments other than `widths` are the arguments `thangld_kan()` accepts for this basis.

`describe(spec)` One-line description of a layer.

`param_dims(p, q, spec)` Named list of parameter array dimensions for a layer with `p` inputs and `q` outputs.

`init(p, q, spec)` Named list of initial parameter arrays (uses the R random number generator; the seed is set by `thangld_kan()`).

`check_params(par, spec)` Error if the parameters left their domain; otherwise invisible `TRUE`.

`forward_r(h, par, spec)` Base R forward pass of an n x p matrix: `list(out = n x q matrix, cache = anything)`.

`backward_r(cache, g, par, spec)` Analytic backward pass given the n x q gradient of the loss with respect to the layer output: `list(grads = named list like par, dh = n x p matrix)`.

`forward_torch(h, par, spec)` The same forward pass with 'torch' tensors (par is a named list of tensors); must be differentiable by autograd. Write `tensor * scalar`, not `scalar * tensor`, so that R doubles are not rounded to float32.

`edge_r(x, par, spec, i, j)` Data frame describing edge $i \to j$ at inputs x, with the edge output in column `phi`.

Optional, used by `thangld_kan_validate()` (a missing one is reported as a skipped check, never as passed): `naive_forward(h, par, spec)` (an independent loop implementation), `basis_checks()` (list of `list(check, error, tol)`), `validation_fixture()` (returns `list(model, x, y)`), and `label`.

Models store only the basis name, so a model with a registered basis can be reloaded only after the basis is registered again.

Value

`register_kan_basis()` returns `name` invisibly; `unregister_kan_basis()` returns `TRUE` invisibly if a basis was removed.

Examples

```
# a toy linear basis: phi(x) = w * x + b on every edge
lin <- list(
  label = "linear (toy)",
  layer_specs = function(widths) rep(list(list()), length(widths) - 1L),
  describe = function(spec) "linear",
  param_dims = function(p, q, spec) list(w = c(p, q)),
  init = function(p, q, spec) list(w = matrix(rnorm(p * q, sd = 0.3), p, q)),
  check_params = function(par, spec) invisible(TRUE),
  forward_r = function(h, par, spec) list(out = h %*% par$w, cache = h),
  backward_r = function(cache, g, par, spec)
    list(grads = list(w = crossprod(cache, g)), dh = g %*% t(par$w)),
  forward_torch = function(h, par, spec) torch::torch_matmul(h, par$w),
  edge_r = function(x, par, spec, i, j) data.frame(phi = par$w[i, j] * x))
register_kan_basis("toy_linear", lin)
m <- thangld_kan(c(2, 3, 1), basis = c("toy_linear", "chebyshev"), seed = 1)
m
unregister_kan_basis("toy_linear")
```

`relu_basis` *ReLU-KAN basis functions and their derivatives*

Description

`relu_basis()` evaluates the $G = g + k$ ReLU-KAN basis functions

$$R_m(x) = \{r\,\mathrm{ReLU}(x - s_m)\,\mathrm{ReLU}(e_m - x)\}^2, \quad m = 1, \ldots, G,$$

on an equally spaced grid over `grid_range = c(a, b)` with $L = b - a$: $s_m = a + L(m - 1 - k)/g$, $e_m = s_m + L(k + 1)/g$ and $r = 4g^2/\{(k + 1)^2 L^2\}$, so each function is a smooth bump with support $[s_m, e_m]$ and peak value 1 at the center. With `grid_range = c(0, 1)` the grid and r are those of the original ReLU-KAN layer (Qiu et al. 2024). `relu_basis_deriv()` returns $dR_m/dx = 2r^2uv(v - u)$ with $u = \mathrm{ReLU}(x - s_m)$, $v = \mathrm{ReLU}(e_m - x)$.

Usage

```
relu_basis(x, grid_size = 5L, k = 3L, grid_range = c(-2, 2))

relu_basis_deriv(x, grid_size = 5L, k = 3L, grid_range = c(-2, 2))
```

Arguments

`x`	Numeric vector.
`grid_size`	Number of grid intervals $g \geq 1$.
`k`	Support width in grid intervals minus one, $k \geq 0$.
`grid_range`	Numeric `c(min, max)`, `min` < `max`.

Value

A `length(x)` by `grid_size + k` matrix.

Examples

```
relu_basis(c(-1, 0, 1), grid_size = 5, k = 3)
```

---

thangld_graph_kan *Create a graph KAN architecture*

Description

A graph KAN layer first averages node states over neighborhoods with the normalized adjacency matrix $\hat{A}$ (`thangld_graph_kan_adjacency()`) and then applies a Kolmogorov-Arnold layer to every node,

$$H^{(l)} = \mathrm{KAN}_l(\hat{A}H^{(l-1)}), \qquad \mathrm{KAN}_l(U)_{v,j} = \sum_i \sum_{k=0}^{K} c_{i,j,k} T_k(\tanh U_{v,i}),$$

which is the "aggregate, then KAN" architecture of graph KANs (Kiamari, Kiamari and Krishnamachari, 2024; the GraphKAN prototype of Zhang, 2024, uses the same order with Fourier edges). By default edges are Chebyshev expansions (layout `c(in, out, K + 1)`, no bias, degree 3); any basis of `thangld_kan()` can be used instead (`basis`, `...`, `options`). With `L` layers a node's prediction depends on its `L`-hop neighborhood. With no graph (identity $\hat{A}$) the model reduces to a Chebyshev KAN on the node's own features, which is the natural baseline.

Usage

```
thangld_graph_kan(
  widths,
  basis = "chebyshev",
  ...,
  self_branch = FALSE,
  options = NULL,
  seed = NULL
)
```

Arguments

widths	Integer layer widths; the first is the number of node features, the last must be 1.
basis	Edge basis, one value or one per layer (see kan_bases()).
...	Named basis-specific arguments, as in thangld_kan(). For "chebyshev" the degree defaults to 3 here.
self_branch	Logical; feed [H, A_hat H] instead of A_hat H to every layer (see Details).
options	NULL or a named list of basis-specific arguments per basis, as in thangld_kan().
seed	Optional seed for the initialization.

Details

With self_branch = TRUE every layer receives the node's own state next to its neighborhood mean,

$$H^{(l)} = \mathrm{KAN}_l([H^{(l-1)}, \hat{A}H^{(l-1)}]),$$

so a layer has $2\times$ the inputs (the "concatenate self and neighbors" update of GraphSAGE, Hamilton, Ying and Leskovec, 2017). The network can then use the node's own features, its one-hop mean or both in any layer, so a two-layer model can still fit a target that depends on one hop only, or on the node's own features only. The default (FALSE) is the plain "aggregate, then KAN" architecture above.

Task: transductive node regression (one numeric target per node, one graph, full-batch training).

Value

An object of class thangld_graph_kan.

References

Kiamari, M., Kiamari, M. and Krishnamachari, B. (2024). GKAN: Graph Kolmogorov-Arnold Networks. doi:10.48550/arXiv.2406.06470

Hamilton, W. L., Ying, R. and Leskovec, J. (2017). Inductive representation learning on large graphs. Advances in Neural Information Processing Systems 30. doi:10.48550/arXiv.1706.02216

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. International Conference on Learning Representations. doi:10.48550/arXiv.1609.02907

Examples

```
thangld_graph_kan(c(2, 4, 1), seed = 1)
thangld_graph_kan(c(2, 4, 1), self_branch = TRUE, seed = 1)
thangld_graph_kan(c(2, 4, 1), basis = "rbf", seed = 1)
```

`thangld_graph_kan_adjacency`

Normalized adjacency matrix of an undirected graph

Description

Returns $\hat{A} = D^{-1/2}(A + I)D^{-1/2}$ (Kipf and Welling, 2017), where D is the degree matrix of $A + I$; with `self_loops = FALSE` the identity is not added. This is the propagation operator of every graph KAN layer.

Usage

```
thangld_graph_kan_adjacency(graph, n = NULL, self_loops = TRUE)
```

Arguments

`graph`	Either an $n \times n$ non-negative symmetric adjacency matrix (weights allowed, diagonal ignored) or a two-column matrix or data frame of node indices (one undirected edge per row; duplicates and self-edges are dropped). A 2×2 matrix is read as an adjacency matrix unless `n` is given and differs from 2.
`n`	Number of nodes (needed for an edge list when the highest-index nodes are isolated; default: the largest index).
`self_loops`	Add I before normalizing.

Value

A dense $n \times n$ matrix.

Examples

```
edges <- cbind(c(1, 2, 3), c(2, 3, 4))
thangld_graph_kan_adjacency(edges)
```

thangld_graph_kan_fit *Fit a graph KAN for node regression*

Description

Transductive node regression on one graph: all nodes and their features are propagated, the loss is the mean squared error over the `train` nodes, and the `val` nodes are used only for early stopping (the epoch with the lowest validation loss is restored). Labels of other nodes are never read; they may be `NA`. Feature and target standardization constants are estimated on the training nodes only. Training is full batch: one Adam step per epoch, after which the training and validation losses are recorded.

Usage

```
thangld_graph_kan_fit(
  model,
  x,
  graph,
  y,
  train,
  val = NULL,
  epochs = 300L,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  patience = NULL,
  min_delta = 0,
  self_loops = TRUE,
  backend = c("torch", "r"),
  dtype = c("float64", "float32"),
  verbose = FALSE
)
```

Arguments

`model`	A `thangld_graph_kan()` object.
`x`	Node feature matrix or data frame (n rows, finite values).
`graph`	Adjacency matrix or edge list (see `thangld_graph_kan_adjacency()`), or `NULL` for no graph (identity propagation; the node-wise KAN baseline).
`y`	Numeric target, one value per node (`NA` allowed outside `train` and `val`).
`train`, `val`	Node indices (or logical vectors) of training and validation nodes; they must not overlap.
`epochs`	Maximum number of epochs (= Adam steps).
`lr`, `betas`, `eps`	Adam settings (as `torch::optim_adam`).
`patience`	Early-stopping patience in epochs (`NULL` for none).

min_delta	Minimum validation improvement.
self_loops	Passed to thangld_graph_kan_adjacency().
backend	"torch" (autograd) or "r" (analytic gradients in base R; no libtorch needed).
dtype	"float64" or "float32" (torch backend).
verbose	Print progress.

Value

The fitted thangld_graph_kan with history, preprocess, a_hat, x and train information.

Examples

```
g <- thangld_graph_kan_simulate(60, seed = 1)
y <- sin(g$x[, 1]) + drop(thangld_graph_kan_adjacency(g$adjacency) %*% g$x[, 2])
fit <- thangld_graph_kan_fit(thangld_graph_kan(c(2, 3, 1), seed = 1), g$x, g$adjacency, y,
                     train = 1:40, val = 41:50, epochs = 50, lr = 0.02,
                     backend = "r")
fit
```

---

thangld_graph_kan_forward_r

Forward pass and gradients in base R

Description

thangld_graph_kan_forward_r() evaluates the network on standardized features x with propagation matrix a_hat. thangld_graph_kan_gradient_r() returns the masked mean squared error over nodes and its analytic gradients with respect to every coefficient and to x; thangld_graph_kan_fd_gradient_r() the same by central differences (order 2 or 4), for checking.

Usage

```
thangld_graph_kan_forward_r(model, x, a_hat, layers = FALSE)

thangld_graph_kan_gradient_r(model, x, a_hat, y, nodes)

thangld_graph_kan_fd_gradient_r(
  model,
  x,
  a_hat,
  y,
  nodes,
  step = 1e-05,
  order = 2L
)
```

Arguments

`model`	A `thangld_graph_kan`.
`x`	Standardized node features.
`a_hat`	Propagation matrix (`thangld_graph_kan_adjacency()`).
`layers`	Return every layer's output.
`y`	Target (model scale).
`nodes`	Indices of the nodes in the loss.
`step`, `order`	Finite-difference step and order.

Value

See Description.

Examples

```
g <- thangld_graph_kan_simulate(8, seed = 2)
m <- thangld_graph_kan(c(2, 2, 1), seed = 1)
a <- thangld_graph_kan_adjacency(g$adjacency)
thangld_graph_kan_forward_r(m, g$x, a)
thangld_graph_kan_gradient_r(m, g$x, a, rnorm(8), 1:5)$loss
```

`thangld_graph_kan_metrics` *Regression metrics*

Description

Regression metrics

Usage

```
thangld_graph_kan_metrics(truth, estimate)
```

Arguments

`truth`, `estimate`	Numeric vectors of equal length.

Value

Named vector `rmse`, `mae`, `r2`.

Examples

```
thangld_graph_kan_metrics(c(1, 2, 3), c(1.1, 1.9, 3.2))
```

thangld_graph_kan_save *Save and load a graph KAN*

Description

Models are plain R objects (no torch pointers), stored with `saveRDS()` together with a format tag and the package and R versions.

Usage

```
thangld_graph_kan_save(model, file)

thangld_graph_kan_load(file)
```

Arguments

`model`	A `thangld_graph_kan`.
`file`	Path.

Value

`thangld_graph_kan_save()`: `file`, invisibly. `thangld_graph_kan_load()`: the model.

Examples

```
f <- tempfile(fileext = ".rds")
thangld_graph_kan_save(thangld_graph_kan(c(2, 1), seed = 1), f)
thangld_graph_kan_load(f)
```

thangld_graph_kan_simulate *Simulate a stochastic block model graph with node features*

Description

Nodes are split into `blocks` equal communities; an edge appears with probability `p_in` inside a community and `p_out` between communities. Features are independent $N(0, 1)$ plus a community mean shift of `shift`. Used in examples, tests and benchmarks (no external data).

Usage

```
thangld_graph_kan_simulate(
  n = 150L,
  p = 2L,
  blocks = 3L,
  p_in = 0.1,
  p_out = 0.01,
  shift = 1,
  seed = NULL
)
```

Arguments

n	Number of nodes.
p	Number of features.
blocks	Number of communities.
p_in, p_out	Edge probabilities.
shift	Size of the community mean shift of the features.
seed	Optional seed (the caller's random number stream is restored).

Value

A list with x ($n \times p$), adjacency (0/1 matrix), edges (two-column matrix) and block (community labels).

Examples

```
g <- thangld_graph_kan_simulate(30, seed = 1)
dim(g$adjacency)
```

---

thangld_graph_kan_validate

Run the numerical validation suite

Description

Checks the normalized adjacency against an explicit loop, the Chebyshev forward pass against an independent loop implementation (explicit neighbor sums and $\cos(k \arccos z)$), analytic gradients of the masked loss against fourth-order central differences ($h = 10^{-4}, 10^{-5}, 10^{-6}$), for both the plain and the self-branch architecture, permutation equivariance, the identity-graph reduction, save/load, and, with the torch runtime, torch forward, autograd and full-batch Adam training against the base R oracle. These checks use Chebyshev edges; for every other built-in basis the gradients (and, with torch, the torch forward pass and autograd) of a self-branch graph model are also checked. Torch checks that cannot run are reported with pass = NA and counted in n_skipped; they are never counted as passed.

Usage

```
thangld_graph_kan_validate(backend = c("both", "r", "torch"))
```

Arguments

backend	"both", "r" or "torch".

Value

A data frame (group, check, error, tol, pass) with attributes all_pass and n_skipped.

Examples

```
v <- thangld_graph_kan_validate("r")
v[, c("check", "pass")]
```

---

thangld_kan *Create a Kolmogorov-Arnold network with a chosen edge basis*

Description

Each layer computes $h_j^{(l+1)} = \sum_i \phi_{j,i}(h_i^{(l)})$, where the edge functions $\phi_{j,i}$ are expansions in the chosen basis. basis may differ between layers. The built-in bases follow published KAN variants (see kan_bases()); other bases are added with register_kan_basis().

Usage

```
thangld_kan(widths, basis = "chebyshev", ..., options = NULL, seed = NULL)
```

Arguments

widths	Integer layer widths; the last must be 1.
basis	Basis name, one value or one per layer (see kan_bases()).
...	Named basis-specific arguments (see Details).
options	NULL or a named list (names are bases) of named lists of basis-specific arguments.
seed	Optional seed for the initialization.

Details

Basis-specific arguments are passed by name in .... A per-layer argument (marked "per layer") may be one value or one value per layer of the network; in a mixed network the values for layers of another basis are ignored. An argument that no selected basis uses is an error. options gives basis-specific values that override ..., for example options = list(jacobi = list(degree = 3)).

"chebyshev" $\sum_k c_{ijk} T_k(\tanh x)$. degree (per layer, default 4), init_sd (NULL: $1/(\mathrm{in}(K+1))$). Same layout and initialization as ChebyKAN.

`"rbf"` FastKAN: optional layer normalization, Gaussian radial basis functions on a fixed grid and a SiLU base branch. `num_grids` (8), `grid_min` (-2), `grid_max` (2), `denominator` (`NULL`: grid spacing), `layernorm` (`TRUE`), `base_update` (`TRUE`), `ln_eps` (1e-5), `spline_init_sd` (0.1).

`"legendre"` $\sum_k c_{ijk} P_k(\tanh x)$. `degree` (per layer, 4), `init_sd` (`NULL`).

`"wavelet"` Wav-KAN with the Mexican-hat wavelet, trainable translation and scale per edge. `scale_param` (`"log"` or `"raw"`), `bias` (`TRUE`).

`"rational"` Safe Pade form $P(x)/(1+|Q(x)|)$ (no poles). `num_degree` (per layer, 3), `den_degree` (per layer, 2, not above `num_degree`), `init_sd` (`NULL`), `den_init_sd` (0.1).

`"hybrid"` Sum of SiLU, Chebyshev and RBF parts. `bases` (subset of `"silu"`, `"chebyshev"`, `"rbf"`), `degree` (per layer, 3), `num_grids` (per layer, 8), `grid_min` (-2), `grid_max` (2), `denominator` (`NULL`), `rbf_init_sd` (0.1). See `hybrid_kan_components()`.

`"jacobi"` $\sum_k c_{ijk} P_k^{(\alpha,\beta)}(\tanh x)$. `degree` (per layer, 4), `alpha` (1), `beta` (1), `init_sd` (`NULL`).

`"relu"` ReLU-KAN squared ReLU bumps with trainable phases. `grid_size` (per layer, 5), `k` (per layer, 3), `grid_range` (`c(-2, 2)`).

Value

An object of class `thangld_kan`.

Examples

```
thangld_kan(c(2, 3, 1), basis = "chebyshev", degree = 4, seed = 1)
thangld_kan(c(2, 3, 1), basis = "relu", grid_size = 4, seed = 1)
# one basis per layer
thangld_kan(c(2, 3, 1), basis = c("rbf", "legendre"), degree = 3, seed = 1)
```

---

thangld_kan_edge *Evaluate a learned edge function*

Description

Returns the edge function of layer `layer` from input node `input` to output node `output` on a grid. An edge function is one component of the fitted network: in a multi-layer network it is not the partial effect of a predictor, and it is never a causal effect.

Usage

```
thangld_kan_edge(
  model,
  layer = 1L,
  input = 1L,
  output = 1L,
  x = NULL,
  n = 101L,
  scale = c("model", "original")
)
```

Arguments

`model`	A model (fitted or not).
`layer`, `input`, `output`	Edge indices (1-based).
`x`	Optional grid of edge inputs; by default `n` points over the range of this edge's input in the training data (or `c(-3, 3)`).
`n`	Number of grid points.
`scale`	`"model"` (layer-input scale) or `"original"` (layer 1 of a fitted model: original units of the predictor).

Value

A data frame with `x`, `h` (model-scale layer input), the basis-specific components and `phi` (edge output on the model scale).

Examples

```
m <- thangld_kan(c(2, 3, 1), seed = 1)
head(thangld_kan_edge(m, layer = 1, input = 2, output = 3, n = 5))
```

`thangld_kan_fit` *Fit a Kolmogorov-Arnold network for regression*

Description

Trains all parameters of a `thangld_kan()` network by mini-batch Adam on the mean squared error. Inputs (and optionally the response) are standardized with constants estimated on the training rows only; predictions are returned on the original scale of `y`.

Usage

```
thangld_kan_fit(
  model,
  x,
  y,
  validation_data = NULL,
  validation_split = 0,
  epochs = 200L,
  batch_size = NULL,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  patience = NULL,
  min_delta = 0,
  standardize_x = TRUE,
  standardize_y = TRUE,
```

```
  seed = NULL,
  backend = c("torch", "r"),
  dtype = c("float64", "float32"),
  verbose = FALSE
)
```

Arguments

model	A thangld_kan() object whose first width equals ncol(x). If it was already fitted, training continues and keeps its preprocessing.
x	Numeric matrix or data frame of predictors.
y	Numeric response vector.
validation_data	Optional list(x = , y =) on the original scale.
validation_split	Fraction in [0, 1) of rows held out for validation when validation_data is NULL.
epochs	Maximum number of epochs.
batch_size	Mini-batch size; NULL means full batch.
lr, betas, eps	Adam settings.
patience	NULL (no early stopping) or number of epochs.
min_delta	Minimum improvement of the validation loss.
standardize_x, standardize_y	Standardize with training mean and standard deviation.
seed	Optional seed for the batch order and the validation split; the caller's random number stream is restored afterwards.
backend	"torch" or "r".
dtype	"float64" (default) or "float32" (torch backend only).
verbose	Print progress.

Details

Backends. backend = "torch" (default) uses R 'torch' autograd and torch::optim_adam(). backend = "r" uses the base R analytic gradient (thangld_kan_gradient_r()) and an Adam update that follows torch.optim.Adam; it needs no libtorch. Both backends receive the same mini-batch schedule (drawn in R from seed), so in float64 they follow the same trajectory up to rounding.

Validation and early stopping. Supply validation_data (a list with x and y) or validation_split. The validation loss is recorded every epoch; with patience, training stops after patience epochs without an improvement larger than min_delta. With validation data the parameters of the epoch with the lowest validation loss are restored. Never use test data here.

Constant columns in the training rows are fixed at their training value on the model scale during fitting and prediction (with a warning).

Value

The fitted model with `preprocess`, `history` (`epoch`, `train_loss`, `val_loss` on the standardized scale) and `train`.

Examples

```
set.seed(1)
x <- matrix(runif(200, -2, 2), ncol = 2, dimnames = list(NULL, c("a", "b")))
y <- sin(x[, "a"]) + 0.5 * x[, "b"]^2 + rnorm(100, sd = 0.05)
fit <- thangld_kan_fit(thangld_kan(c(2, 1), seed = 1), x, y, epochs = 50, lr = 0.02,
              backend = "r", validation_split = 0.2, seed = 1)
fit
head(predict(fit, x))
```

---

thangld_kan_forward_r *Forward pass and gradients in base R (numerical oracle)*

Description

`thangld_kan_forward_r()` evaluates the network on model-scale inputs (already standardized). `thangld_kan_gradient_r()` returns the mean squared error $L = n^{-1} \sum_b (\hat{y}_b - y_b)^2$ and its analytic gradient with respect to every parameter array and to the inputs. `thangld_kan_fd_gradient_r()` approximates the same gradients by central differences of order 2 (truncation error $O(h^2)$) or 4 ($O(h^4)$); validation only. These functions use base R only and are the reference for the torch backend.

Usage

```
thangld_kan_forward_r(model, x, layers = FALSE)

thangld_kan_gradient_r(model, x, y)

thangld_kan_fd_gradient_r(model, x, y, step = 1e-05, order = 2L)
```

Arguments

`model`	A `thangld_kan()` object.
`x`	Numeric matrix of model-scale inputs (`n` by `widths[1]`).
`layers`	If `TRUE`, return the list of layer outputs.
`y`	Numeric vector of length `n` (model scale).
`step`	Central-difference step.
`order`	Order of the central-difference stencil, 2 or 4.

Value

`thangld_kan_forward_r()`: numeric vector (or list of layer outputs). The gradient functions: a list with `loss`, `params` (list per layer of named gradient arrays shaped like `model$params`) and `x`.

Examples

```
m <- thangld_kan(c(2, 3, 1), seed = 1)
x <- matrix(c(-1, 0.5, 2, -0.2), 2)
thangld_kan_forward_r(m, x)
g <- thangld_kan_gradient_r(m, x, c(0, 1))
fd <- thangld_kan_fd_gradient_r(m, x, c(0, 1))
max(abs(unlist(g$params) - unlist(fd$params)))
```

---

thangld_kan_metrics *Regression metrics*

Description

Regression metrics

Usage

```
thangld_kan_metrics(truth, estimate)
```

Arguments

`truth`, `estimate` Numeric vectors of equal length.

Value

Named vector `rmse`, `mae`, `r2`; `r2` is `NA` when `truth` has zero variance.

Examples

```
thangld_kan_metrics(c(1, 2, 3), c(1.1, 1.9, 3.2))
```

---

thangld_kan_save *Save and load a Kolmogorov-Arnold network*

Description

A model is plain R data, saved with `saveRDS()` together with metadata (format version, package version, R version, time); it can be reloaded with or without libtorch.

Usage

```
thangld_kan_save(model, file)

thangld_kan_load(file)
```

Arguments

model	A model.
file	Path of the .rds file.

Value

thangld_kan_save() returns file invisibly; thangld_kan_load() returns the model, with saving metadata in attr(, "saved").

Examples

```
m <- thangld_kan(c(2, 1), seed = 1)
f <- tempfile(fileext = ".rds")
thangld_kan_save(m, f)
identical(m$params, thangld_kan_load(f)$params)
```

---

thangld_kan_validate *Run the numerical validation suite*

Description

Runs the acceptance checks of thangldKANr on small fixtures in float64, for each requested basis, and returns one row per check: basis-specific checks, the forward pass against an independent loop implementation, analytic gradients against fourth-order central differences (steps 1e-4, 1e-5, 1e-6) for a multi-layer network, save/load, and (when the torch runtime is installed) the torch forward pass, autograd gradients and a short Adam run against the base R backend. With mixed = TRUE the same forward, gradient and torch checks also run on a network whose layers use different bases. Missing checks (no torch runtime, or a registered basis without a naive implementation) are reported as skipped (pass = NA), never as passed. Errors are scaled errors max(abs(a - b) / pmax(1, abs(a), abs(b))) unless stated otherwise.

Usage

```
thangld_kan_validate(
  backend = c("both", "r", "torch"),
  basis = kan_bases()$name,
  mixed = TRUE
)
```

Arguments

backend	"both", "r" or "torch".
basis	Bases to validate (default: all available, see kan_bases()).
mixed	Also validate a network that mixes bases across layers.

Value

A data frame with `basis`, `group`, `check`, `error`, `tol`, `pass`; attributes `all_pass` and `n_skipped`.

Examples

```
res <- thangld_kan_validate("r", basis = "chebyshev", mixed = FALSE)
res[, c("check", "pass")]
```

`thangld_temporal_kan` *Create a lag-window Temporal KAN*

Description

A Temporal KAN here is a Kolmogorov-Arnold network that receives a window of lagged values of a univariate series and predicts the next value (one-step-ahead forecasting). The KAN part is a stack of edge layers of any basis of `thangld_kan()` (default Chebyshev, $\phi_{j,i}(x) = \sum_k c_{i,j,k} T_k(\tanh x)$); with `hidden = integer()` the forecast is an additive function of the lags (a nonlinear additive autoregression), and hidden layers allow interactions between lags. It is not recurrent or attention-based.

Usage

```
thangld_temporal_kan(
  lags = 1:12,
  hidden = integer(),
  basis = "chebyshev",
  ...,
  difference = FALSE,
  log = FALSE,
  options = NULL,
  seed = NULL
)
```

Arguments

`lags`	Positive integer lags used as inputs, e.g. `1:12` or `c(1, 2, 12)`.
`hidden`	Integer widths of hidden layers (`integer()` for none).
`basis`	Edge basis, one value or one per layer (see `kan_bases()`).
`...`	Named basis-specific arguments, as in `thangld_kan()` (for example `degree` for `"chebyshev"`, default 4).
`difference`	If `TRUE`, the network models first differences $y_t - y_{t-1}$ and forecasts are integrated back.
`log`	If `TRUE`, the series is log-transformed first (must be positive) and forecasts are exponentiated back (median forecast).
`options`	`NULL` or a named list of basis-specific arguments per basis, as in `thangld_kan()`.
`seed`	Optional seed for the initialization.

Value

An object of class thangld_temporal_kan.

Examples

```
thangld_temporal_kan(lags = 1:12, hidden = 4, seed = 1)
thangld_temporal_kan(lags = 1:6, basis = "rbf", seed = 1)
```

---

thangld_temporal_kan_embed

Lag-window design matrix

Description

Builds, for every time index t with all lags available, the row $(z_{t-l_1}, \ldots, z_{t-l_p})$ and the target z_t, where z is the (optionally log-transformed and differenced) series. Only values before t enter row t.

Usage

```
thangld_temporal_kan_embed(y, lags, difference = FALSE, log = FALSE)
```

Arguments

y	Numeric vector or ts.
lags	Positive integer lags.
difference, log	Transformations as in thangld_temporal_kan().

Value

A list with x (matrix, columns lag1, ...), target, index (time index of each target in y) and z (transformed series aligned with y, NA where undefined).

Examples

```
w <- thangld_temporal_kan_embed(1:10, lags = c(1, 3))
cbind(w$x, target = w$target, t = w$index)
```

thangld_temporal_kan_fit *Fit a Temporal KAN on a series*

Description

Builds lag windows from y, estimates the standardization (mean and standard deviation of the transformed series) from the training targets only, holds out the last n_val windows (in time order) for validation and early stopping, and trains by mini-batch Adam on the mean squared one-step error. Mini-batches are drawn from training windows; every window contains only values before its target, so shuffling windows does not use future information. The validation windows come after all training windows.

Usage

```
thangld_temporal_kan_fit(
  model,
  y,
  n_val = 0L,
  epochs = 200L,
  batch_size = 32L,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  patience = NULL,
  min_delta = 0,
  seed = NULL,
  backend = c("torch", "r"),
  dtype = c("float64", "float32"),
  verbose = FALSE
)
```

Arguments

model	A thangld_temporal_kan() object.
y	Numeric vector or ts (the full history available at the forecast origin).
n_val	Number of final windows used for validation (0 for none).
epochs, batch_size, lr, betas, eps, patience, min_delta	Training settings (see thangld_kan_fit() for their meaning).
seed	Optional seed for the batch order.
backend	"torch" (default) or "r" (base R, no libtorch).
dtype	"float64" or "float32" (torch backend).
verbose	Print progress.

Value

The fitted thangld_temporal_kan with scaling, history, train and the stored series y (used by predict.thangld_temporal_kan()).

Examples

```
y <- as.numeric(datasets::lynx)
m <- thangld_temporal_kan(lags = 1:3, seed = 1)
fit <- thangld_temporal_kan_fit(m, log(y), n_val = 10, epochs = 50, lr = 0.02,
                        backend = "r", seed = 1)
predict(fit)
```

---

thangld_temporal_kan_rolling

Rolling-origin one-step evaluation

Description

For every forecast origin t_0 from initial to length(y) - 1, the value y_{t_0+1} is forecast using only $y_1, \ldots, y_{t_0}$. The Temporal KAN is (re)fitted on $y_{1:t_0}$ at the first origin and then every refit_every origins (expanding window); between refits the last fitted network and its standardization (estimated before that refit's origin) are applied to the current history, so no value after an origin is ever used for that origin's forecast. Validation windows for early stopping are the last n_val windows before each refit origin.

Usage

```
thangld_temporal_kan_rolling(
  model,
  y,
  initial,
  refit_every = 1L,
  period = NULL,
  ...
)
```

Arguments

model	An unfitted thangld_temporal_kan() (its architecture and seed are reused at every refit).
y	Numeric vector or ts.
initial	First forecast origin (number of observations available).
refit_every	Refit frequency in origins.
period	Seasonal period for the seasonal naive baseline and MASE (default frequency(y) for a ts, else 1).
...	Passed to thangld_temporal_kan_fit() (e.g. n_val, epochs, lr, seed, backend).

Details

Baselines computed at the same origins: naive ($\hat{y}_{t_0+1} = y_{t_0}$) and seasonal naive ($\hat{y}_{t_0+1} = y_{t_0+1-m}$) when `period` = `m` > 1. MASE scales by the in-sample mean absolute one-step seasonal-naive (or naive) error of the series up to the first origin.

Value

A list with `forecasts` (data frame: `origin`, `index`, `actual`, `kan`, `naive`, `snaive`, `refit`) and `metrics` (RMSE, MAE, MASE per method).

Examples

```
y <- log(as.numeric(datasets::lynx))
r <- thangld_temporal_kan_rolling(thangld_temporal_kan(1:3, seed = 1), y, initial = 100,
                          refit_every = 7, n_val = 10, epochs = 30,
                          lr = 0.02, backend = "r", seed = 1)
r$metrics
```

`thangld_temporal_kan_save`

Save and load a Temporal KAN

Description

Save and load a Temporal KAN

Usage

```
thangld_temporal_kan_save(model, file)

thangld_temporal_kan_load(file)
```

Arguments

`model`	A `thangld_temporal_kan`.
`file`	Path of the `.rds` file.

Value

`thangld_temporal_kan_save()` returns `file` invisibly; `thangld_temporal_kan_load()` returns the model.

Examples

```
m <- thangld_temporal_kan(1:3, seed = 1)
f <- tempfile(fileext = ".rds")
thangld_temporal_kan_save(m, f)
identical(thangld_temporal_kan_load(f)$params, m$params)
```

thangld_temporal_kan_validate *Run the numerical validation suite*

Description

Checks the lag-window model with Chebyshev edges (forward pass against an independent `cos(k acos z)` loop implementation, analytic gradients against fourth-order central differences for steps `1e-4`, `1e-5`, `1e-6`), the lag-window construction against a direct loop, and (with the torch runtime) the torch forward pass, autograd and a short Adam run against the base R backend, and the gradients (and torch pass) of the model with every other built-in basis. Skipped torch checks are reported with `pass = NA`, never as passed.

Usage

```
thangld_temporal_kan_validate(backend = c("both", "r", "torch"))
```

Arguments

backend	`"both"`, `"r"` or `"torch"`.

Value

A data frame with `group`, `check`, `error`, `tol`, `pass`; attributes `all_pass` and `n_skipped`.

Examples

```
thangld_temporal_kan_validate("r")[, c("check", "pass")]
```

Index

chebyshev_basis, 3
chebyshev_basis_deriv *(*chebyshev_basis*)*, 3

graphics::plot(), *8*

hybrid_basis, 3
hybrid_basis_deriv *(*hybrid_basis*)*, 3
hybrid_kan_components, 4
hybrid_kan_components(), *24*

jacobi_basis, 5
jacobi_basis_deriv *(*jacobi_basis*)*, 5

kan_bases, 6
kan_bases(), *16*, *23*, *29*, *30*
kan_layer_bases, 6

legendre_basis, 7
legendre_basis_deriv *(*legendre_basis*)*, 7

mexican_hat, 7
mexican_hat_deriv *(*mexican_hat*)*, 7

plot.thangld_kan, 8
predict.thangld_graph_kan, 9
predict.thangld_kan, 10
predict.thangld_temporal_kan, 11
predict.thangld_temporal_kan(), *33*

rational, 11
rational_deriv *(*rational*)*, 11
rbf_basis, 12
rbf_basis_deriv *(*rbf_basis*)*, 12
rbf_grid *(*rbf_basis*)*, 12
register_kan_basis, 13
register_kan_basis(), *23*
relu_basis, 14
relu_basis_deriv *(*relu_basis*)*, 14

saveRDS(), *28*

thangld_graph_kan, 15
thangld_graph_kan(), *18*
thangld_graph_kan_adjacency, 17
thangld_graph_kan_adjacency(), *15*, *18–20*
thangld_graph_kan_fd_gradient_r *(*thangld_graph_kan_forward_r*)*, 19
thangld_graph_kan_fit, 18
thangld_graph_kan_fit(), *9*
thangld_graph_kan_forward_r, 19
thangld_graph_kan_gradient_r *(*thangld_graph_kan_forward_r*)*, 19
thangld_graph_kan_load *(*thangld_graph_kan_save*)*, 21
thangld_graph_kan_metrics, 20
thangld_graph_kan_save, 21
thangld_graph_kan_simulate, 21
thangld_graph_kan_validate, 22
thangld_kan, 23
thangld_kan(), *4*, *6*, *13*, *15*, *16*, *25–27*, *30*
thangld_kan_edge, 24
thangld_kan_edge(), *8*
thangld_kan_fd_gradient_r *(*thangld_kan_forward_r*)*, 27
thangld_kan_fit, 25
thangld_kan_fit(), *32*
thangld_kan_forward_r, 27
thangld_kan_gradient_r *(*thangld_kan_forward_r*)*, 27
thangld_kan_gradient_r(), *26*
thangld_kan_load *(*thangld_kan_save*)*, 28
thangld_kan_metrics, 28
thangld_kan_save, 28
thangld_kan_validate, 29
thangld_kan_validate(), *13*, *14*
thangld_temporal_kan, 30
thangld_temporal_kan(), *31–33*

thangld_temporal_kan_embed, 31
thangld_temporal_kan_fit, 32
thangld_temporal_kan_fit(), *33*
thangld_temporal_kan_load
(thangld_temporal_kan_save), 34
thangld_temporal_kan_rolling, 33
thangld_temporal_kan_save, 34
thangld_temporal_kan_validate, 35

unregister_kan_basis
(register_kan_basis), 13