Package 'SpatialDownscaling'

January 26, 2026

Type Package

Title Methods for Spatial Downscaling Using Deep Learning

Version 0.1.2

Date 2026-01-21

Imports stats, tensorflow, keras3, magrittr, Rdpack, raster, abind

Description The aim of the spatial downscaling is to increase the spatial resolution of the gridded geospatial input data. This package contains two deep learning based spatial downscaling methods, super-resolution deep residual network (SRDRN) (Wang et al., 2021 <doi:10.1029/2020WR029308>) and UNet (Ronneberger et al., 2015 <doi:10.1007/978-3-319-24574-4_28>), along with a statistical baseline method bias correction and spatial disaggregation (Wood et al., 2004 <doi:10.1023/B:CLIM.0000013685.99609.9e>). The SRDRN and UNet methods are implemented to optionally account for cyclical temporal patterns in case of spatio-temporal data. For more details of the methods, see Sipilä et al. (2025) <doi:10.48550/arXiv.2512.13753>.

License GPL-3

Encoding UTF-8

LazyData true

RdMacros Rdpack

RoxygenNote 7.3.2

SystemRequirements Python (>= 3.8), TensorFlow, Keras

Depends R (>= 4.4.0)

NeedsCompilation no

Author Mika Sipilä [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-5912-840X>), Claudia Cappello [aut] (ORCID: <https://orcid.org/0000-0002-7905-5068>), Sandra De Iaco [aut] (ORCID: <https://orcid.org/0000-0003-1820-2068>), Klaus Nordhausen [aut] (ORCID: <https://orcid.org/0000-0002-3758-8501>), Sara Taskinen [aut] (ORCID: <https://orcid.org/0000-0001-9470-7258>)

Maintainer Mika Sipilä <mika.e.sipila@jyu.fi>

Repository CRAN

Date/Publication 2026-01-26 16:20:38 UTC

Contents

`bcsd` *Bias Correction Spatial Disaggregation (BCSD) for statistical downscaling*

Description

Implements the BCSD method for statistical downscaling of climate data. The approach consists of two main steps: (1) bias correction using quantile mapping and (2) spatial disaggregation using interpolation.

Usage

```
bcsd(
  coarse_data,
  fine_data,
  method = "bilinear",
  n_quantiles = 100,
  reference_period = NULL,
  extrapolate = TRUE,
  normalize = TRUE
)
```

Arguments

`coarse_data`	A 3D array of coarse resolution input data. The two first dimensions are the spatial coordinates (e.g., latitude and longitude) in grid, and the third dimension refers to the training samples (e.g. time).
`fine_data`	A 3D array of fine resolution output data. The two first dimensions are the spatial coordinates (e.g., latitude and longitude) in grid, and the third dimension refers to the training samples (e.g. time).
`method`	Character. Interpolation method that is used by resample to perform predictions. The options are ('bilinear', 'ngb') that refer to bilinear and nearest neighbor interpolation, respectively. Default: 'bilinear'.
`n_quantiles`	Integer. Number of quantiles for bias correction. Default: 100.

`reference_period`
: Vector. Start and end indices for the reference period. Default: NULL (use all data).

`extrapolate`
: Logical. Indicating whether to extrapolate corrections outside the range of the training data. Default: TRUE.

`normalize`
: Logical. Indicating whether to normalize data before processing. Default: TRUE.

Details

The BCSD method is a statistical downscaling technique that combines bias correction and spatial disaggregation. It uses quantile mapping to correct biases in coarse resolution data and then applies spatial interpolation to disaggregate the data to a finer resolution.

The function allows for the interpolation methods "bilinear" and "ngb", that perform bilinear and nearest neighbor interpolation, respectively. For more information on these methods, please refer to the documentation for resample. The function provides the option to normalize data before processing, by using the `normalize` argument. The quantile mapping step involves calculating quantiles from the coarse data and mapping them to the fine data. The interpolation step uses the specified method to create a fine resolution grid from the coarse data.

Value

Object of class `BCSD` containing the trained model components:

`quantile_map`
: Quantile mapping function for bias correction.

`interpolation_params`
: Parameters for spatial interpolation.

`axis_names`
: Names of the axes in the fine data.

`scalers`
: List of scalers. If `normalize = TRUE`, the list contains scalers `coarse` and `fine` for the coarse and fine data, respectively. If `normalize = FALSE`, the list is empty.

`model_params`
: List of all model parameters.

Examples

```
# Simple example with random data
coarse <- array(rnorm(8 * 8 * 10), dim = c(8, 8, 10))  # e.g. lat x lon x time
fine <- array(rnorm(16 * 16 * 10), dim = c(16, 16, 10))    # e.g. lat x lon x time
model <- bcsd(coarse, fine, method = "bilinear", n_quantiles = 100)
coarse_new <- array(rnorm(8 * 8 * 3), dim = c(8, 8, 3))  # e.g. lat x lon x time
predictions <- predict(model, coarse_new)
```

predict.BCSD *Predict method for BCSD model*

Description

This function generates predictions using a trained BCSD model.

Usage

```
## S3 method for class 'BCSD'
predict(object, newdata, ...)
```

Arguments

object	A BCSD model object, an output of the bcsd function.
newdata	Array or raster that has the new coarse resolution data to be downscaled. The resolution should match the resolution of the training data. The first two dimensions are the spatial dimensions and the third refers to the training samples (e.g. time).
...	Additional arguments (not used).

Details

The predict method applies the trained BCSD model to new coarse resolution data. It performs bias correction using the quantile mapping function and then applies spatial interpolation, specified during model training with the method parameter, to generate fine resolution predictions.

Value

A matrix, array or raster of the downscaled predictions at fine resolution.

See Also

bcsd for training the model.

Examples

```
# Simple example with random data
coarse <- array(rnorm(10*20*30), dim = c(10, 20, 30))  # time x lat x lon
fine <- array(rnorm(10*40*60), dim = c(10, 40, 60))    # time x lat x lon
model <- bcsd(coarse, fine, method = "bilinear", n_quantiles = 100)
# New coarse data for prediction
new_coarse <- array(rnorm(5*20*30), dim = c(5, 20, 30))  # time x lat x lon
predictions <- predict(model, new_coarse)
# Check dimensions of predictions
dim(predictions)  # Should be (5, 40, 60) for time x lat x lon
```

`predict.srdrn` *Predict method for SRDRN*

Description

This function makes predictions using a trained SRDRN model. It takes a trained SRDRN object and new data as input, normalizes the new data, and uses the model to make predictions. The predictions are then rescaled back to the original range.

Usage

```
## S3 method for class 'srdrn'
predict(object, newdata, time_points = NULL, ...)
```

Arguments

`object`	A trained SRDRN object.
`newdata`	A 3D array of shape (N_1, N_1, n) representing the new coarse resolution data to be downscaled, where N_1 x N_1 is the coarse resolution of the training samples and n is the number of samples to be downscaled. The two first dimensions are the spatial coordinates in grid, and the third dimension refers to the samples (e.g. time).
`time_points`	An optional numeric vector of time points of the new data.
`...`	Additional parameters (not used).

Details

The predict method for the SRDRN class takes a trained SRDRN object and new data as input. The input resolution (N_1 x N_1) has to match the input dimension of the training samples. The method normalizes the new data using the same min-max scaling used during training. The new data is reshaped to match the input shape of the model, and the model is used to make predictions. The output data has the same fine resolution (N_2 x N_2) as the target training data. The predictions are rescaled back to the original range using the min-max scaling parameters of the training data. The output is a 3D array of the predicted data.

Value

A 3D array of shape (N_2, N_2, n) representing the predicted data, where N_2 x N_2 is the fine resolution of the training samples.

See Also

`srdrn` for fitting SRDRN model.

Examples

```
# Generate dummy low-resolution (16×16) and high-resolution (32×32) data
n <- 10
input  <- array(runif(8 * 8 * n),  dim = c(8, 8, n))
target <- array(runif(16 * 16 * n),  dim = c(16, 16, n))

time_vec <- 1:n
model <- srdrn(
  coarse_data  = input,
  fine_data = target,
  time_points = time_vec,
  cyclical_period = 365,
  temporal_layers = c(32, 64),
  epochs = 1,
  batch_size = 4
)

n_new <- 3
newdata <- array(runif(8 * 8 * n_new),
                     dim = c(8, 8, n_new))
predictions <- predict(model, newdata, 1:n_new)
```

---

predict.UNet *Predict function for UNet model*

Description

This function generates predictions using a trained UNet model.

Usage

```
## S3 method for class 'UNet'
predict(object, newdata, time_points = NULL, ...)
```

Arguments

object	A UNet model object.
newdata	Array or list of arrays. New data to predict on in format (x, y, time).
time_points	An optional numeric vector containing the time points of the new data.
...	Additional arguments (not used).

Details

The predict function applies the trained UNet model to new coarse data. It performs denormalization if the model was trained with normalization.

Value

Array of predictions in format (x, y, time).

See Also

unet for fitting UNet model.

Examples

```
# Create tiny dummy data:
# Coarse grid: 8x8 → Fine grid: 16x16
nx_c <- 8
ny_c <- 8
nx_f <- 16
ny_f <- 16
T <- 5  # number of time steps

# Coarse data:
coarse_data <- array(runif(nx_c * ny_c * T),
                     dim = c(nx_c, ny_c, T))

# Fine data:
fine_data <- array(runif(nx_f * ny_f * T),
                   dim = c(nx_f, ny_f, T))

# Optional time points
time_points <- 1:T

# Fit a tiny UNet (very small filters to keep the example fast)
model_obj <- unet(
  coarse_data,
  fine_data,
  time_points = time_points,
  filters = c(8, 16),
  initial_filters = c(4),
  epochs = 1,
  batch_size = 4,
  verbose = 0
)

T_new <- 3
newdata <- array(runif(nx_c * ny_c * T_new),
                     dim = c(nx_c, ny_c, T_new))
predictions <- predict(model_obj, newdata, 1:T_new)
```

srdrn *Super Resolution CNN for Spatial Downscaling*

Description

This function implements a Time-aware Super Resolution Deep Neural Network (SRDRN) for spatial downscaling of grid based data. The function allows an option for adding a temporal module for spatio-temporal applications.

Usage

```
srdrn(
  coarse_data,
  fine_data,
  time_points = NULL,
  val_coarse_data = NULL,
  val_fine_data = NULL,
  val_time_points = NULL,
  cyclical_period = NULL,
  temporal_basis = c(9, 17, 37),
  temporal_layers = c(32, 64, 128),
  temporal_cnn_filters = c(8, 16),
  temporal_cnn_kernel_sizes = list(c(3, 3), c(3, 3)),
  activation = "relu",
  cos_sin_time = FALSE,
  use_batch_norm = FALSE,
  output_channels = 1,
  num_residual_blocks = 3,
  num_res_block_filters = 64,
  upscaling_filters = c(64, 32, 16, 8, 4, 2),
  validation_split = 0,
  start_from_model = NULL,
  metrics = c(),
  epochs = 10,
  batch_size = 32,
  seed = NULL
)
```

Arguments

coarse_data	A 3D array of shape (N_1, N_1, n) representing the coarse resolution input data in grid, where N_1 x N_1 is the coarse resolution and n is the sample size. The two first dimensions are the spatial coordinates and the third dimension refers to the samples (e.g. time).
fine_data	A 3D array of shape (N_2, N_2, n) representing the fine resolution target data in grid, where N_2 x N_2 is the fine resolution and n is the sample size. The two

first dimensions are the spatial coordinates and the third dimension refers to the samples (e.g. time).

`time_points` An optional numeric vector of length n representing the time points associated with each sample.

`val_coarse_data`
An optional 3D array of shape (N_1, N_1, n) representing the input validation data.

`val_fine_data` An optional 3D array of shape (N_2, N_2, n) representing the target validation data.

`val_time_points`
An optional numeric vector of length n representing the time points of the validation samples.

`cyclical_period`
An optional numeric value representing the cyclical period for time encoding (e.g. 365 for yearly seasonality).

`temporal_basis` A numeric vector specifying the temporal basis functions to use for time encoding (default is c(9, 17, 37)).

`temporal_layers`
A numeric vector specifying the number of units in each dense layer for time encoding (default is c(32, 64, 128)).

`temporal_cnn_filters`
A numeric vector specifying the number of filters in each convolutional layer for temporal feature processing (default is c(8, 16)).

`temporal_cnn_kernel_sizes`
A list of integer vectors specifying the kernel sizes for each convolutional layer in the temporal feature processing (default is list(c(3, 3), c(3, 3))).

`activation` A character string specifying the activation function to use in the model to introduce nonlinearity. The options are listed in https://keras.io/api/layers/activations. Default is "relu".

`cos_sin_time` A logical value indicating whether to use cosine and sine transformations for time encoding (default is FALSE).

`use_batch_norm` A logical value indicating whether to use batch normalization in the residual blocks (default is FALSE).

`output_channels`
An integer specifying the number of output channels (default is 1).

`num_residual_blocks`
An integer specifying the number of residual blocks in the model (default is 3).

`num_res_block_filters`
A integer specifying the number of filters in each residual block (default is 64).

`upscaling_filters`
A numeric vector specifying the number of filters in each upsampling layer (by default, the first X values from vector c(64, 32, 16, 8, 4, 2) are selected, where X is the upscaling factor.).

`validation_split`
A numeric value between 0 and 1 specifying the fraction of the training data to use for validation (default is 0.2).

`start_from_model`
: An optional pre-trained Keras model to continue training from (default is NULL).

`metrics`
: A character vector specifying additional metrics to monitor during training (default is an empty vector).

`epochs`
: An integer specifying the number of training epochs (default is 10).

`batch_size`
: An integer specifying the batch size for training (default is 32).

`seed`
: An optional integer value to set the random seed for reproducibility (default is NULL).

Details

The Super Resolution Deep Residual Network (SRDRN) implements a deep-learning-based spatial downscaling approach inspired by Super-Resolution CNNs (SRCNN) (Dong et al. 2015) and extended for environmental applications following (Wang et al. 2021).

The objective of SRDRN is to learn a mapping from coarse-resolution gridded fields to finer-resolution targets by combining convolutional feature extraction, residual learning, and sub-pixel upsampling. The method is designed for both purely spatial and fully spatio-temporal downscaling when time information is provided. The method consists of the following main components:

- *Feature Extraction Block*: An initial convolutional layer extracts low-level spatial features from the coarse-resolution input.
- *Residual Blocks*: A sequence of residual blocks learn higher-order spatial dependencies. Residual connections stabilize training and allow deeper representations.
- *Upsampling Module*: Sub-pixel convolution (pixel shuffle) layers upscale feature maps to match the high-resolution target grid.

If `time_points` are provided, the model includes an auxiliary temporal branch. Time is encoded either via:

- Radial basis temporal encodings (`temporal_basis`), or
- Cosine–sine cyclical encodings (`cos_sin_time = TRUE`).

The encoded temporal features pass through a multilayer perceptron (`temporal_layers`) and are reshaped to spatial form before being concatenated with CNN features. This enables learning time-varying downscaling dynamics (e.g., seasonality, long-term trends). The function supports missing data via masking.

Value

An object of class SRDRN containing:

`model`
: The trained Keras model.

`input_mean`
: The mean value of the input data used for normalization.

`input_sd`
: The standard deviation of the input data used for normalization.

`target_mean`
: The mean value of the target data used for normalization.

`target_sd`
: The standard deviation of the target data used for normalization.

`input_mask`
: A logical array indicating the missing values in the input data.

`target_mask`	A logical array indicating the missing values in the target data.
`min_time_point`	The minimum time point in the input data.
`max_time_point`	The maximum time point in the input data.
`cyclical_period`	The cyclical period used for temporal encoding.
`axis_names`	A list containing the names of the axes (longitude, latitude, time).
`history`	The training history of the model.

References

Dong C, Loy CC, He K, Tang X (2015). "Image super-resolution using deep convolutional networks." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **38**(2), 295–307.

Wang F, Tian D, Lowe L, Kalin L, Lehrter J (2021). "Deep learning for daily precipitation and temperature downscaling." *Water Resources Research*, **57**(4), e2020WR029308. doi:10.1029/2020WR029308.

Examples

```
# Generate dummy low-resolution (16×16) and high-resolution (32×32) data
n <- 20
input  <- array(runif(16 * 16 * n),  dim = c(16, 16, n))
target <- array(runif(32 * 32 * n),  dim = c(32, 32, n))

model1 <- srdrn(
  coarse_data  = input,
  fine_data = target,
  epochs = 1,
  batch_size = 4
)
```

`unet` *UNet model for spatial downscaling using deep learning*

Description

Implements a time-aware UNet convolutional neural network for spatial downscaling of grid data. Time-aware UNet features an encoder-decoder architecture with skip connections and a temporal module. The function allows an option for adding a temporal module for spatio-temporal applications.

Usage

```
unet(
  coarse_data,
  fine_data,
  time_points = NULL,
  val_coarse_data = NULL,
  val_fine_data = NULL,
  val_time_points = NULL,
  cyclical_period = NULL,
  cycle_onehot = FALSE,
  cos_sin_transform = FALSE,
  temporal_basis = c(9, 17, 37),
  temporal_layers = c(32, 64, 128),
  temporal_cnn_filters = c(8, 16),
  temporal_cnn_kernel_sizes = list(c(3, 3), c(3, 3)),
  initial_filters = c(16),
  initial_kernel_sizes = list(c(3, 3)),
  filters = c(32, 64, 128),
  kernel_sizes = list(c(3, 3), c(3, 3), c(3, 3)),
  use_batch_norm = FALSE,
  dropout_rate = 0.2,
  activation = "relu",
  final_activation = "linear",
  optimizer = "adam",
  learning_rate = 0.001,
  loss = "mse",
  metrics = c(),
  batch_size = 32,
  epochs = 10,
  start_from_model = NULL,
  validation_split = 0,
  normalize = TRUE,
  callbacks = NULL,
  seed = NULL,
  verbose = 1
)
```

Arguments

coarse_data	3D array. The coarse resolution input data in format (x, y, time).
fine_data	3D array. The fine resolution target data in format (x, y, time).
time_points	Numeric vector. Optional time points corresponding to each time step in the data.
val_coarse_data	An optional 3D array of coarse resolution input data in format (x, y, time).
val_fine_data	An optional 3D array of fine resolution target data in format (x, y, time).

`val_time_points`
: An optional numeric vector of length n representing the time points of the validation samples.

`cyclical_period`
: Numeric. Optional period for cyclical time encoding (e.g., 365 for yearly seasonality).

`cycle_onehot`
: Boolean. If TRUE, a onehot encoded vector of temporal cycles is added as input to temporal module.

`cos_sin_transform`
: Logical. Whether to use cosine-sine transformation for time features. Default: FALSE.

`temporal_basis`
: A numeric vector specifying the temporal basis functions to use for time encoding (default is c(9, 17, 37)).

`temporal_layers`
: A numeric vector specifying the number of units in each dense layer for time encoding (default is c(32, 64, 128)).

`temporal_cnn_filters`
: A numeric vector specifying the number of filters in each convolutional layer for temporal feature processing (default is c(8, 16)).

`temporal_cnn_kernel_sizes`
: A list of integer vectors specifying the kernel sizes for each convolutional layer in the temporal feature processing (default is list(c(3, 3), c(3, 3))).

`initial_filters`
: Integer vector. Number of filters in the initial convolutional layers. Default: c(16).

`initial_kernel_sizes`
: List of integer vectors. Kernel sizes for the initial convolutional layers. Default: list(c(3, 3)).

`filters`
: Integer vector. Number of filters in each convolutional layer. Default: c(32, 64, 128).

`kernel_sizes`
: List of integer vectors. Kernel sizes for each convolutional layer. Default: list(c(3, 3), c(3, 3), c(3, 3)).

`use_batch_norm`
: Logical. Whether to use batch normalization after convolutional layers. Default: FALSE.

`dropout_rate`
: Numeric. Dropout rate for regularization. Default: 0.2.

`activation`
: Character. Activation function for hidden layers. The options are listed in https://keras.io/api/layers/activations. Default: "relu".

`final_activation`
: Character. Activation function for output layer. The options are listed in https://keras.io/api/layers/activations. Default: "linear".

`optimizer`
: Character or optimizer object used in `keras3::compile` (see e.g. optimizer_adam). Optimizer for training. The options are listed in https://keras.io/api/optimizers. Default: "adam".

`learning_rate`
: Numeric. Learning rate for optimizer. Default: 0.001.

loss	Character or loss function used in keras3::compile (see Loss). Loss function for training. The options are listed in https://keras.io/api/losses. Default: "mse".
metrics	Optional character vector used in keras3::compile. Metrics to track during training. The options are listed in https://keras.io/api/metrics. Default is an empty vector.
batch_size	Integer. Batch size for training. Default: 32.
epochs	Integer. Number of training epochs. Default: 100.
start_from_model	An optional pre-trained Keras model to continue training from (default is NULL).
validation_split	Numeric. Fraction of data to use for validation. Default: 0.
normalize	Logical. Whether to normalize data before training. Default: TRUE.
callbacks	List. Keras callbacks for training (see Callback). Default: NULL.
seed	Integer. Random seed for reproducibility. Default: NULL.
verbose	Integer. Verbosity mode (0, 1, or 2). Default: 1.

Details

The UNet architecture (Ronneberger et al. 2015) is widely used in image processing tasks and has recently been adopted for spatial downscaling applications (Sha et al. 2020). The method implemented here consists of:

1. **Initial Upscaling** – Coarse-resolution inputs are first upsampled using bilinear interpolation to match the spatial dimensions of the fine-resolution target.
2. **Initial Feature Extraction** – Multiple convolutional layers extract low-level features before entering the encoder path.
3. **Encoder Path** – A sequence of convolutional blocks with max-pooling reduces spatial dimensions while increasing feature depth.
4. **Decoder Path** – Spatial resolution is recovered via bilinear upsampling and convolutional layers. Skip connections from the encoder help preserve fine-scale information.
5. **Skip Connections** – These link encoder and decoder layers at matching resolutions, improving gradient flow and retaining fine spatial structure.
6. **Temporal Module (optional)** – Time information can be incorporated through cosine–sine encoding, one-hot seasonal encoding, or radial-basis temporal features. These are passed through dense layers and reshaped to merge with the UNet bottleneck.

The function supports missing data via masking, optional normalization, validation data, and configurable UNet depth and width.

Value

List containing the trained model and associated components:

model	Trained Keras model
input_mask	Mask for input data based on the missing values

target_mask	Mask for target data based on the missing values
min_time_point	Minimum time point in the training data
max_time_point	Maximum time point in the training data
cyclical_period	Cyclical period for time encoding
max_season	Maximum season for time encoding
axis_names	Names of the axes in the input data
history	Training history

References

Ronneberger O, Fischer P, Brox T (2015). "U-net: Convolutional networks for biomedical image segmentation." In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, 234–241. Springer. doi:10.1007/9783319245744_28.

Sha Y, Gagne II DJ, West G, Stull R (2020). "Deep-learning-based gridded downscaling of surface meteorological variables in complex terrain. Part II: Daily precipitation." *Journal of Applied Meteorology and Climatology*, **59**(12), 2075–2092.

Examples

```
# Create tiny dummy data:
# Coarse grid: 8x8 → Fine grid: 16x16
nx_c <- 8
ny_c <- 8
nx_f <- 16
ny_f <- 16
T <- 5  # number of time steps

# Coarse data:
coarse_data <- array(runif(nx_c * ny_c * T),
                     dim = c(nx_c, ny_c, T))

# Fine data:
fine_data <- array(runif(nx_f * ny_f * T),
                   dim = c(nx_f, ny_f, T))

# Optional time points
time_points <- 1:T

# Fit a tiny UNet (very small filters to keep the example fast)
model_obj <- unet(
  coarse_data,
  fine_data,
  time_points = time_points,
  filters = c(8, 16),
  initial_filters = c(4),
  epochs = 1,
```

```
  batch_size = 4,
  verbose = 0
)
```

`weather_italy` *Daily Weather Data for Italy*

Description

A dataset containing daily gridded weather variables for Italy that are obtained from the ERA5-Land dataset (Hersbach et al. 2023).

Usage

```
weather_italy
```

Format

A 4-dimensional array with dimensions:

longitude Longitude grid points (0.2 degree resolution)

latitude Latitude grid points (0.2 degree resolution)

variable Weather variables: relative humidity (%), temperature (°C), total precipitation (meters per day)

time Daily measurements from 2023-11-01 to 2023-12-31

Index