

Package ‘neuralnetwork’

October 10, 2026

Title Fast Compact Multilayer Perceptrons

Version 0.1.4

Description A small multilayer perceptron implementation for 'R'. It supports regression and classification, multiple hidden layers, mini-batch training, adaptive moment estimation, stochastic gradient descent, momentum, Nesterov acceleration, resilient backpropagation and limited-memory quasi-Newton optimization, dropout, squared-weight regularization, early stopping, convergence thresholds, gradient clipping, sample and class weights, callback hooks, target scaling and robust Huber loss for regression, 'Rcpp' forward-pass kernels, formula interfaces, model evaluation with balanced classification metrics, cross-validation, compact tuning, permutation importance, model persistence helpers, and standard prediction methods. Methods follow Rumelhart, Hinton and Williams (1986) <[doi:10.1038/323533a0](#)>, with optimizers including Riedmiller and Braun (1993) <[doi:10.1109/ICNN.1993.298623](#)>, Nocedal (1980) <[doi:10.1090/S0025-5718-1980-0572855-7](#)>, and Kingma and Ba (2014) <[doi:10.48550/arXiv.1412.6980](#)>.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports Rcpp

LinkingTo Rcpp

Suggests knitr, rmarkdown

VignetteBuilder knitr

NeedsCompilation yes

Author Feng Ji [aut, cre]

Maintainer Feng Ji <f.ji@utoronto.ca>

Repository CRAN

Date/Publication 2026-10-10 15:00:02 UTC

Contents

neuralnetwork-package	2
neuralnetwork-callbacks	3
neuralnetwork-compat	4
neuralnetwork-metrics	6
neuralnetwork-objects	8
nn_cv	10
nn_evaluate	11
nn_fit	13
nn_permutation_importance	17
nn_save	18
nn_tune	19
plot.neuralnetwork	21
predict.neuralnetwork	22
print.neuralnetwork	23
summary.neuralnetwork	23
Index	25

neuralnetwork-package *Compact neural networks for tabular R data*

Description

neuralnetwork fits small multilayer perceptrons for tabular regression, binary classification, and multiclass classification. Inputs can be formulas, data frames, matrices, or vectors. Fitted objects support S3 prediction methods, metrics, tuning, cross-validation, feature importance, save/load helpers, and compatibility helpers for common **nnet** and **neuralnet** tasks.

Details

The main entry point is `nn_fit`. A usual analysis has these steps:

1. Fit a model with `nn_fit`.
2. Predict with `predict.neuralnetwork`.
3. Evaluate with `nn_evaluate`.
4. Tune or cross-validate with `nn_tune` and `nn_cv` when the problem needs it.
5. Inspect feature importance with `nn_permutation_importance`.

The automatic defaults keep models small. `hidden = "auto"` chooses a hidden-layer layout from the task and input width, `optimizer = "auto"` uses L-BFGS for small deterministic tabular models and Adam for stochastic training features, and `backend = "auto"` uses the Rcpp forward-pass backend when available.

For users coming from **nnet** or **neuralnet**, the [compatibility reference](#) covers class indicators, layer activations, sensitivities and the limits of parameter inference.

Main functions

- [nn_fit](#): fit a neural network.
- [nn_evaluate](#): compute task-aware metrics.
- [nn_tune](#): run a compact validation-set grid search.
- [nn_cv](#): run repeated k-fold cross-validation.
- [nn_permutation_importance](#): estimate feature importance.
- [nn_save](#) and [nn_load](#): save and load models.

References

- [neuralnetwork-metrics](#): metric definitions and metric selection.
- [neuralnetwork-callbacks](#): callback state and return values.
- [neuralnetwork-objects](#): fitted object structure.

See Also

[nn_fit](#), [nn_evaluate](#), [nn_tune](#), [nn_cv](#), [nn_permutation_importance](#)

Examples

```
fit <- nn_fit(Species ~ ., iris, hidden = "auto", epochs = 5,
             validation_split = 0.2, seed = 1, verbose = FALSE)
fit
nn_evaluate(fit, iris)
```

neuralnetwork-callbacks

Callbacks in neuralnetwork training

Description

Reference for callback functions passed to [nn_fit](#) through the `callbacks` argument.

Details

Callbacks are called once at the end of each epoch for non-L-BFGS optimizers. Pass either a single function or a list of functions. Each callback receives a state list with:

`epoch` Current epoch.

`train_loss` Training loss after the epoch.

`validation_loss` Validation loss, or NA when no validation split is used.

`train_metric` Training task metric, accuracy for classification or RMSE for regression.

`validation_metric` Validation task metric, or NA.

`best_loss` Best monitored loss so far.

`best_epoch` Epoch with the best monitored loss.

`gradient_norm` Largest batch gradient norm observed in the epoch.

`learning_rate` Learning rate for the next epoch unless changed by a callback.

`task` Resolved task name.

A callback may return `FALSE` or `list(stop = TRUE)` to stop training. Return `NULL` to leave training unchanged.

To change the next epoch's learning rate, return `list(learning_rate = value)`. The value must be positive and finite. If `stop` is returned in a list, it must be a single `TRUE` or `FALSE`.

When multiple callbacks are supplied, they are run in order. If one callback requests stopping, later callbacks are skipped for that epoch.

See Also

[nn_fit](#)

Examples

```
halve_after_two <- function(state) {
  if (state$epoch == 2) {
    return(list(learning_rate = state$learning_rate * 0.5))
  }
  NULL
}

fit <- nn_fit(mpg ~ wt + hp, mtcars, hidden = 4, epochs = 4,
             callbacks = halve_after_two, seed = 1, verbose = FALSE)
fit
```

neuralnetwork-compat *Compatibility helpers*

Description

Small helpers that cover common nnet and neuralnet tasks.

Usage

```
nn_class_ind(classes)

nn_which_is_max(x)

nn_multinom(formula, data, weights = NULL, ...)

nn_compute(model, newdata)

nn_generalized_weights(model, newdata, response = 1L,
```

```

    epsilon = 1e-4)

nn_gwplot(model, newdata, selected.covariate = 1L,
  response = 1L, ...)

nn_hessian(model, newdata, y = NULL, epsilon = 1e-4,
  max_params = 120)

nn_confint(model, newdata, y = NULL, level = 0.95,
  epsilon = 1e-4, max_params = 120)

```

Arguments

<code>classes</code>	A non-empty factor or vector of classes without missing values.
<code>x</code>	A vector, matrix, or data frame.
<code>formula, data, weights, ...</code>	Arguments passed to <code>nn_fit</code> .
<code>model</code>	A fitted neuralnetwork model.
<code>newdata</code>	New predictor data.
<code>response</code>	Response name or index.
<code>epsilon</code>	Positive finite-difference step for generalized weights and Hessian calculations. Accepted but not used by the analytical interval calculation.
<code>selected.covariate</code>	Predictor name or index for plotting generalized weights.
<code>y</code>	Optional truth vector or matrix, with one row or value per prediction row.
<code>level</code>	Confidence level between 0 and 1.
<code>max_params</code>	Positive integer maximum parameter count allowed for finite-difference Hessian.

Details

These helpers are intentionally small wrappers around the fitted neuralnetwork object.

- `nn_class_ind()` creates a one-hot class indicator matrix.
- `nn_which_is_max()` returns the first maximum index for a vector, or row-wise maxima for matrices and data frames.
- `nn_multinom()` fits a no-hidden-layer multiclass model, similar in spirit to a multinomial log-linear model.
- `nn_compute()` returns activations and final outputs in a compute-style list.
- `nn_generalized_weights()`: input sensitivities for a selected response, estimated by central finite differences.
- `nn_hessian()` computes finite-difference curvature of the mean unweighted data loss, excluding regularization. It is a diagnostic, not a covariance estimate or a substitute for total sample information.

- `nn_confint()` provides parameter intervals only for converged, unweighted, unregularized L-BFGS fits without hidden layers: single-output squared-error regression or binary logistic classification. Older models without training metadata must be refitted. It uses analytical total sample information, estimated residual variance and Student t limits for regression, and unit dispersion with normal limits for logistic models. Supply the original training observations. Singular or ill-conditioned information is an error; no ridge or negative-variance truncation is used.

Sensitivities are central finite differences per unit of an encoded input before predictor scaling. Regression output is back-transformed from target scaling. Classification sensitivities refer to probabilities, not log odds. For `poly()`, interactions and factor contrasts, columns are encoded features rather than original variables. These are not identical to **neuralnet**'s generalized weights. The sensitivity plot pads an almost constant vertical range to avoid magnifying finite-difference noise; an explicit `ylim` overrides it.

Parameter intervals refer to internal weights and biases, including any predictor or target scaling. Use `scale = FALSE`, `y_scale = FALSE` to compare linear coefficients directly with `lm()` or `glm()`. The regression intervals assume independent Gaussian errors with constant variance; logistic intervals are asymptotic and can be unreliable with small samples or separation. One-class outcomes or fitted probabilities within 10^{-6} of zero or one are refused for intervals. Hidden-layer weights are not generally identifiable; multiclass softmax also has a redundant parameterization. Neither receives parameter intervals from this helper.

Value

`nn_compute()` returns a list with `neurons` and `net.result`.

`nn_generalized_weights()` returns a matrix of input sensitivities. `nn_hessian()` returns a curvature matrix.

`nn_confint()` returns a table of estimates, standard errors and limits.

Examples

```
nn_class_ind(iris$Species[1:5])
fit <- nn_multinom(Species ~ ., iris, epochs = 5,
                  verbose = FALSE, seed = 1)
nn_compute(fit, iris[1:3, ])$net.result

linear <- nn_fit(mpg ~ wt, mtcars, hidden = 0, optimizer = "lbfgs",
               scale = FALSE, y_scale = FALSE, epochs = 300,
               verbose = FALSE, seed = 2)
nn_confint(linear, mtcars)
confint(lm(mpg ~ wt, mtcars)) # Same limits, in intercept-first order
```

Description

Metric definitions for model evaluation and selection.

The same conventions apply in `nn_evaluate`, `nn_tune` and `nn_cv`.

[Permutation importance](#) uses them too.

Details

Classification metrics:

`accuracy` Fraction of correct predictions.

`balanced_accuracy` Mean recall across classes present in the truth data. Useful when class frequencies are uneven.

`macro_precision` Mean class-wise precision over all fitted classes.

`macro_recall` Mean class-wise recall over all fitted classes.

`macro_f1` Mean class-wise F1 score over all fitted classes.

`log_loss` Negative log likelihood of the true class under the predicted probabilities. Lower is better.

Binary classification also reports:

`sensitivity` Recall for the second fitted class, treated as the positive class.

`specificity` Recall for the first fitted class.

`precision` Positive predictive value for the second fitted class.

`recall` Alias for sensitivity.

`f1` Positive-class F1, computed as $2TP/(2TP + FP + FN)$.

For a class with no predicted observations, precision is set to zero. For a class absent from the truth, recall is set to zero. F1 is computed directly as $2TP/(2TP + FP + FN)$, with zero returned when the denominator is zero. These rules apply to binary metrics too. They are reporting conventions; a class absent from the truth has no observed sensitivity to estimate.

Macro averages include every class learned during fitting, even when a class is absent from the evaluation data or predictions. A class present in the truth but never predicted therefore contributes zero to macro F1 rather than being dropped. Balanced accuracy averages only over classes present in the truth. It equals macro recall when every fitted class appears in the truth. Macro F1 averages the individual class F1 scores; it is not the harmonic mean of macro precision and macro recall.

Regression metrics:

`rmse` Root mean squared error. Lower is better.

`mae` Mean absolute error. Lower is better.

`rsq` Pooled coefficient of determination. Higher is better. Returns NA for constant truth, including a single observation.

For model selection helpers, `metric = "auto"` uses accuracy for classification and RMSE for regression. `metric = "f1"` uses positive-class F1 for binary classification and macro F1 for multiclass classification. `metric = "loss"` uses the unpenalized fitted data loss in every helper, including Huber loss when selected. Regression loss is computed on the training target scale and sums over outputs; RMSE and MAE are averages over both rows and outputs on the response scale. They are different quantities, not aliases.

For q regression outputs and row weights w_i , RMSE is $\sqrt{\sum_i w_i \sum_j (\hat{y}_{ij} - y_{ij})^2 / (q \sum_i w_i)}$. MAE replaces squared errors with absolute errors. R-squared is one minus the ratio of weighted total squared error to weighted total variation, centering each outcome column at its own weighted mean. This pools output variances; it does not equally average the individual outputs' R-squared values. Different units or scales can make one output dominate pooled R-squared.

Sample-weighted classification uses weighted confusion counts. Classes with zero true weight are excluded from balanced accuracy but not macro averages. Reported log loss clips probabilities below 10^{-15} ; training and `metric = "loss"` use stable logits without that cap. For soft targets, cross-entropy sums target probability times negative log predicted probability.

See Also

[nn_evaluate](#), [nn_tune](#), [nn_cv](#), [nn_permutation_importance](#)

Examples

```
class_names <- c("Excellent", "Poor", "Typical")
confusion <- matrix(
  c(20, 0, 45, 0, 0, 19, 18, 0, 378),
  nrow = 3, byrow = TRUE,
  dimnames = list(truth = class_names, estimate = class_names)
)
confusion
class_counts <- rowSums(confusion) + colSums(confusion)
class_f1 <- 2 * diag(confusion) / class_counts
class_f1
mean(class_f1) # 0.4301658; Poor contributes zero, so divide by three
```

neuralnetwork-objects *neuralnetwork model objects*

Description

Structure of fitted model objects returned by [nn_fit](#).

Details

A fitted model is an S3 object of class `neuralnetwork`. Users usually interact with it through `print()`, `predict()`, `plot()`, `summary()`, `coef()`, [nn_evaluate](#), [nn_save](#), and [nn_load](#).

Important fields include:

params List of weight matrices and bias vectors.
task One of these resolved tasks:
 "regression",
 "binary_classification", or
 "classification".
hidden Hidden-layer sizes.
activation Hidden-layer activation.
optimizer Resolved optimizer.
loss Resolved loss. Classification uses cross-entropy internally.
backend Resolved forward-pass backend.
scaler Predictor centering and scaling information.
target_scaler Regression target scaling information.
blueprint Information used to prepare new data consistently.
classes Classification levels, or NULL for regression.
history Data frame with per-epoch training diagnostics.
training_data_loss Selected model's unpenalized training data loss, using combined training weights.
training_units Stored iteration units:
 L-BFGS: "function_evaluations".
 Other optimizers: "epochs".
convergence List with optimizer convergence metadata. For L-BFGS fits, code and message come from [optim](#).
best_epoch Epoch with the best monitored loss.
validation Prepared held-out predictors, encoded and raw outcomes, sample weights and original row indices; NULL without a holdout. Retaining these data increases model size.
training_options Regularization, dropout, weighting and training-row-count metadata used to check inference eligibility.
 The field `n` counts positive-weight training rows.
 excluded_zero_weight_rows counts omitted training rows.
stopped_early Whether training stopped before the requested number of epochs.

The history data frame records each epoch, with:

- Losses: `train_loss` and `validation_loss`.
- Scores: `train_metric` and `validation_metric`.
- Controls: `gradient_norm`, `learning_rate` and `backtracked`.

Validation columns are NA without a holdout. Printed metrics and summaries describe the selected checkpoint, not the last attempted epoch. The complete history still records later attempted epochs. The object structure is documented for inspection. Code that only needs model predictions should prefer the exported S3 methods and helper functions.

See Also

[nn_fit](#), [predict.neuralnetwork](#), [summary.neuralnetwork](#), [nn_evaluate](#)

nn_cv

*Cross-validate neuralnetwork models***Description**

Run repeated k-fold cross-validation for [nn_fit](#) models and return fold-level scores plus summary statistics.

Usage

```
nn_cv(
  x,
  y = NULL,
  data = NULL,
  k = 5,
  repeats = 1,
  stratify = TRUE,
  metric = c("auto", "loss", "accuracy", "balanced_accuracy",
            "log_loss", "f1", "rmse", "mae", "rsq"),
  seed = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

x, y, data	Model inputs passed to nn_fit .
k	Integer number of folds, between 2 and the number of rows.
repeats	Positive integer number of repeated fold assignments.
stratify	Whether to stratify classification folds, including indicator matrices when task = "classification". Soft targets use their first maximal class for stratification.
metric	Metric to report. "auto" uses accuracy for classification and RMSE for regression. "f1" uses the positive-class F1 for binary classification and macro F1 for multiclass classification.
seed	Optional non-negative integer random seed. Fold models receive deterministic consecutive seeds derived from this value.
verbose	Whether individual fits should print progress.
...	Additional arguments passed to nn_fit .

Details

For classification outcomes, `stratify = TRUE` attempts to keep class proportions similar across folds. Each class must have at least k rows; otherwise a stratified split would create folds whose training sets do not contain all classes. Unused outcome levels do not count as observed classes.

For regression outcomes, folds are sampled without stratification. Repeated cross-validation creates a new fold assignment for each repeat.

The metric is computed on each held-out fold using [nn_evaluate](#). The returned summary data frame reports the mean and standard deviation of finite fold scores for each metric name, with `n_scored` and `n_folds`. Undefined scores remain in results; an entirely undefined metric returns NA mean and standard deviation, not an empty summary. Fold-score standard deviation is not a confidence interval. `sample_weight` in ... is split into training and held-out weights; held-out metrics are weighted too. `metric = "loss"` uses the fitted data loss without a regularization penalty, not RMSE.

Value

A `neuralnetwork_cv` object, a list with:

`results` Fold-level scores.

`summary` Mean and standard deviation by metric.

`models` Fitted model for each fold.

`folds` Repeat identifier, fold identifier and held-out row indices for each model.

`k` Number of folds.

`repeats` Number of repeats.

`call` Matched call.

See Also

[neuralnetwork-metrics](#), [nn_fit](#), [nn_tune](#)

Examples

```
cv <- nn_cv(Species ~ ., iris, k = 3, epochs = 3,
            seed = 1, verbose = FALSE)
cv
```

nn_evaluate

Evaluate a neuralnetwork model

Description

Compute regression or classification metrics for a fitted model. Classification metrics include accuracy, balanced accuracy, log loss, macro precision, macro recall, and macro F1. Binary classification also reports sensitivity, specificity, precision, recall, and F1.

Usage

```
nn_evaluate(model, newdata, y = NULL, sample_weight = NULL)
```

Arguments

<code>model</code>	A fitted neuralnetwork object.
<code>newdata</code>	Data containing predictors, and optionally the response. Evaluation requires at least one observation.
<code>y</code>	Optional truth vector or matrix, with one row or value per prediction row. Required when <code>newdata</code> does not contain the original formula response.
<code>sample_weight</code>	Optional finite non-negative row weights with positive total weight. These affect metrics, confusion counts and data loss.

Details

For classification, truth values are compared against predicted classes and probabilities. Macro precision, recall, and F1 average over all fitted classes. Class-wise precision, recall, and F1 use zero for a zero denominator; a class that is never predicted still contributes to the macro averages. F1 is computed directly from true-positive, false-positive, and false-negative counts. Balanced accuracy averages recall over classes present in the truth. See [neuralnetwork-metrics](#) for formulas and a worked example.

For binary classification, the second fitted class is the positive class. Sensitivity and recall refer to that class; specificity refers to the first fitted class. Precision and F1 refer to the positive class. The positive class is predicted when its probability is at least 0.5, including exactly 0.5; model-selection helpers use the same threshold.

For regression, predictions are returned on the original response scale before metrics are computed. Regression metrics are RMSE, MAE, and R-squared. For a transformed formula response such as $\log(y)$, that response transformation is applied to truth as well: scores are on the log scale, not the raw y scale. Named outcome matrices are aligned to fitted outputs. Classification indicator or probability matrices are also accepted. Class metrics use their first-maximal class; log loss retains soft probabilities.

Value

An object of class `neuralnetwork_evaluation`, a list with:

<code>metrics</code>	Named numeric vector of task-aware metrics.
<code>loss</code>	Unpenalized fitted data loss, using the training target scaler. Classification uses stable logits; unlike reported <code>log_loss</code> , this value is not capped by probability clipping.
<code>predictions</code>	Predicted classes for classification, or numeric predictions for regression.
<code>confusion</code>	Classification only: a confusion table.
<code>probabilities</code>	Classification only: class probabilities.

See Also

[neuralnetwork-metrics](#), [nn_tune](#), [nn_cv](#), [nn_permutation_importance](#)

Examples

```
fit <- nn_fit(Species ~ ., iris, hidden = 6, epochs = 5,
              verbose = FALSE, seed = 1)
nn_evaluate(fit, iris)
```

nn_fit

Fit a small multilayer perceptron

Description

Fit a compact vectorized neural network for regression or classification.

Usage

```
nn_fit(
  x,
  y = NULL,
  data = NULL,
  hidden = "auto",
  task = c("auto", "classification", "regression"),
  activation = c("auto", "relu", "tanh", "sigmoid",
                 "leaky_relu"),
  optimizer = c("auto", "adam", "sgd", "momentum", "nesterov",
                 "rprop", "grprop", "lbfgs"),
  backend = c("auto", "rcpp", "r"),
  loss = c("auto", "squared_error", "huber"),
  huber_delta = 1,
  epochs = 100,
  stepmax = NULL,
  batch_size = 32,
  learning_rate = 0.001,
  momentum = 0.9,
  beta1 = 0.9,
  beta2 = 0.999,
  epsilon = 1e-8,
  l2 = 0,
  dropout = 0,
  sample_weight = NULL,
  class_weight = NULL,
  gradient_clip = Inf,
  learning_rate_decay = 0,
  threshold = 0,
  validation_split = 0,
  patience = Inf,
  min_delta = 1e-4,
  scale = TRUE,
  y_scale = TRUE,
```

```

    shuffle = TRUE,
    seed = NULL,
    callbacks = NULL,
    verbose = TRUE,
    validation_rows = NULL
)

```

Arguments

x	A formula, data frame, matrix, or numeric vector of predictors.
y	Outcome vector or matrix. Required when x is not a formula. For multiclass classification matrices, rows must be one-hot indicators or class probabilities that sum to 1.
data	Data frame used when x is a formula.
hidden	Integer vector giving hidden layer sizes, 0 for no hidden layer, or "auto" for a small default chosen from the task and input width.
task	One of "auto", "classification", or "regression".
activation	Hidden-layer activation. "auto" chooses a default from the task and hidden-layer layout.
optimizer	Optimizer, one of "auto", "adam", "sgd", "momentum", "nesterov", "rprop", "grprop", or "lbfgs". "auto" uses L-BFGS for small deterministic tabular models and Adam when stochastic training features such as dropout or callbacks are used.
backend	Forward-pass backend. "auto" uses Rcpp when available.
loss	Regression loss. "auto" uses squared error; "huber" uses a robust Huber loss. Classification uses cross-entropy internally.
huber_delta	Positive finite Huber transition point, in the training target scale when y_scale = TRUE.
epochs	Positive integer number of training epochs.
stepmax	Optional neuralnet -style alias for epochs.
batch_size	Positive integer mini-batch size.
learning_rate	Positive finite optimizer learning rate.
momentum	Momentum coefficient in $[0, 1)$ for "momentum" and "nesterov" optimizers.
beta1, beta2	Adam exponential-decay coefficients in $[0, 1)$.
epsilon	Positive finite Adam denominator stabilizer.
l2	Finite non-negative L2 regularization strength.
dropout	Dropout probability. Either one value or one value per hidden layer.
sample_weight	Optional finite non-negative row weights. Zero-weight training rows are excluded from training and fitted preprocessing.
class_weight	Optional classification weights. Use "balanced" or a numeric vector with one value per class.
gradient_clip	Global gradient-norm clip value. Use Inf to disable.

learning_rate_decay	Finite non-negative per-epoch learning-rate decay.
threshold	Finite non-negative gradient-norm convergence threshold. A value of 0 disables this stopping rule.
validation_split	Fraction of rows held out for validation.
patience	Positive integer number of unimproved epochs before early stopping, or Inf to disable patience-based stopping.
min_delta	Finite non-negative minimum monitored-loss improvement.
scale	Single TRUE or FALSE: whether to center and scale predictors using the training split.
y_scale	Single TRUE or FALSE: whether to center and scale regression outcomes during training.
shuffle	Single TRUE or FALSE: whether to shuffle rows each epoch.
seed	Optional non-negative integer random seed.
callbacks	Functions called once per epoch. See the callback reference .
verbose	Single TRUE or FALSE: whether to print progress messages.
validation_rows	Unique held-out row indices. Overrides the validation fraction. An empty vector disables the holdout; leave at least one training row.

Details

`nn_fit()` accepts either a formula plus data, or explicit predictor and outcome objects. Data-frame predictors are expanded with `model.matrix()`, so factors are encoded with the usual R contrast machinery. Matrix inputs are used as supplied.

Named matrix predictors are aligned by name at prediction time; a missing fitted name is an error, not a request to use positional order. Unnamed matrices use positional order. Formula transformations and contrasts are retained from fitting. With a holdout, data-dependent bases such as `poly()` are learned on the training rows only. Formula offsets are not supported.

Every network layer has a bias, including the output layer. Removing the formula intercept changes the input encoding but does not remove network biases. A full set of factor indicators together with a bias can therefore be redundant, which prevents parameter intervals.

When `task = "auto"`, factor, character, and logical outcomes are treated as classification. Numeric vectors and matrices are treated as regression. Two-class classification uses a one-output sigmoid model internally, while `predict(type = "prob")` still returns a two-column probability matrix. Multiclass classification uses a softmax output. If `task` is set to `"classification"` and `y` is a multi-column matrix, each row must be a one-hot class indicator or a probability vector.

`hidden = "auto"` chooses a small architecture from the task and input width. Use an integer vector such as `c(16, 8)` for multiple hidden layers, or 0 for no hidden layer. `optimizer = "auto"` uses L-BFGS for small deterministic tabular models and Adam when stochastic training features such as dropout or callbacks are used.

Adam is also chosen when patience, a gradient threshold, learning-rate decay or finite gradient clipping is requested. Explicit L-BFGS fits reject these controls and callbacks instead of silently ignoring them.

For L-BFGS fits, `epochs` and `stepmax` are passed to `optim` as the iteration limit rather than interpreted as mini-batch epochs. The printed model and `summary()` report function evaluations for those fits.

For regression, `loss = "huber"` can be useful when a few observations are unusually influential. If `y_scale = TRUE`, `huber_delta` is measured on the scaled training target, not the original response scale.

Early stopping monitors validation loss when `validation_split > 0` and training loss otherwise. The returned model stores the training history, preprocessing information, fitted parameters, and enough blueprint information to prepare new data consistently at prediction time.

Every strictly lower monitored loss saves a new checkpoint. `min_delta` only governs the patience counter, not which parameters are returned. Training loss includes the penalty $\lambda \sum W^2 / (2 \sum w)$; biases are not penalized. The penalty gradient uses the full training-weight sum even with mini-batches. Validation loss excludes the penalty. For a batch of b rows sampled from n positive-weight training rows, the data gradient is

$$\frac{n}{b} \sum_{i \in B} \frac{w_i}{S} \nabla \ell_i, \quad S = \sum_i w_i.$$

At fixed parameters, averaging over uniformly sampled batches gives the full weighted data gradient. Weights therefore still affect one-row batches; they are not normalized within each batch. Without row weights this reduces to the usual mean batch gradient. The last batch uses its actual size.

Zero-weight training rows are omitted before fitting scalars and formula bases. Raw inputs must still satisfy encoding and finite-value requirements. Balanced class weights are calculated from training rows only, using their sample-weighted class counts; validation metrics use sample weights, not class weights. Training metrics use the combined sample and class weights. Default scaling uses ordinary unweighted positive-weight training means and standard deviations, not weighted moments.

Value

An object of class `neuralnetwork`, a list containing fitted parameters, preprocessing metadata, task information, training history, and S3 methods for printing, prediction, plotting, summaries, and coefficients. See [neuralnetwork-objects](#) for the object structure.

See Also

[Prediction](#), [evaluation](#), [tuning](#), [cross-validation](#), [callbacks](#) and [model objects](#).

Examples

```
fit <- nn_fit(Species ~ ., data = iris, hidden = c(8, 4), epochs = 10,
             batch_size = 16, verbose = FALSE, seed = 1)
predict(fit, iris[1:3, ], type = "prob")
```

nn_permutation_importance

Permutation feature importance

Description

Estimate feature importance by repeatedly permuting each encoded model input and measuring the change in model performance. Positive importance means the permuted feature made the selected metric worse.

Usage

```
nn_permutation_importance(
  model,
  newdata,
  y = NULL,
  metric = c("auto", "loss", "accuracy", "balanced_accuracy",
             "log_loss", "f1", "rmse", "mae", "rsq"),
  n_repeats = 5,
  seed = NULL,
  sample_weight = NULL
)
```

Arguments

model	A fitted neuralnetwork object.
newdata	Data containing predictors, and optionally the response.
y	Optional truth vector or matrix, with one row or value per prediction row. Required when newdata does not contain the original formula response.
metric	Metric used to score the permutations.
n_repeats	Positive integer number of permutations per feature.
seed	Optional non-negative integer random seed.
sample_weight	Optional row weights, as in nn_evaluate .

Details

The baseline metric is computed once on newdata. Then each encoded model input column is shuffled n_repeats times and scored again. Positive importance means the permutation worsened the metric. For error-like metrics such as loss, RMSE, MAE, and log loss, larger permuted values are worse. For score-like metrics such as accuracy, balanced accuracy, F1, macro F1, and R-squared, smaller permuted values are worse.

When a formula or data-frame model contains factor predictors, importance is computed on the encoded model-matrix columns rather than on the original factor as a single group. Single-row permutations leave predictors unchanged. A model without input columns returns a zero-row importance table. An undefined baseline metric, such as R-squared for constant truth, is an error rather than an importance estimate.

Value

A data frame of class `neuralnetwork_importance`. Rows are sorted by decreasing importance, with columns:

feature Encoded feature name.

importance Metric degradation caused by permutation.

baseline Baseline metric before permutation.

permuted Mean metric after permutation.

metric Metric used for scoring.

n_repeats Number of permutations per feature.

See Also

[neuralnetwork-metrics](#), [nn_evaluate](#)

Examples

```
fit <- nn_fit(mpg ~ wt + hp, mtcars, hidden = 4, epochs = 5,
             verbose = FALSE, seed = 1)
nn_permutation_importance(fit, mtcars, n_repeats = 2, seed = 2)
```

nn_save	<i>Save and load neuralnetwork models</i>
---------	---

Description

Save a fitted `neuralnetwork` model to disk and load it back.

Usage

```
nn_save(model, path, compress = TRUE)

nn_load(path)
```

Arguments

- model A fitted `neuralnetwork` object.
- path Path to an RDS file.
- compress Compression argument passed to [saveRDS](#).

Details

The saved file contains the fitted model object, including preprocessing metadata, fitted parameters, and training history. `nn_load()` checks that the path exists, the model class, parameter dimensions, finite values and preprocessing metadata before returning it. Older valid models may still be loaded for prediction, but corrected numerical diagnostics and inference metadata require refitting. `nn_save()` expects the destination directory to already exist; it does not create folders. Models with holdouts retain validation data; consider this before sharing an RDS file. Structural validation does not make an untrusted serialized R object safe to load.

Value

`nn_save()` returns the path invisibly. `nn_load()` returns a neuralnetwork model.

Examples

```
fit <- nn_fit(mpg ~ wt + hp, mtcars, hidden = 4, epochs = 5,
             verbose = FALSE, seed = 1)
path <- tempfile(fileext = ".rds")
nn_save(fit, path)
loaded <- nn_load(path)
predict(loaded, mtcars[1:3, ])
```

nn_tune

Tune neuralnetwork hyperparameters

Description

Fit a grid of `nn_fit()` candidates and rank them by validation performance.

Usage

```
nn_tune(
  x,
  y = NULL,
  data = NULL,
  grid = NULL,
  validation_split = 0.2,
  metric = c("auto", "loss", "accuracy", "balanced_accuracy",
             "log_loss", "f1", "rmse", "mae", "rsq"),
  maximize = NULL,
  error_action = c("stop", "continue"),
  seed = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

<code>x, y, data</code>	Model inputs passed to <code>nn_fit</code> .
<code>grid</code>	Named list of candidate values. Values may be vectors or lists.
<code>validation_split</code>	Validation fraction passed to <code>nn_fit</code> .
<code>metric</code>	Metric used to choose the best model. "f1" uses the positive-class F1 for binary classification and macro F1 for multiclass classification.
<code>maximize</code>	Whether larger metric values are better. Inferred by default.
<code>error_action</code>	What to do when a candidate fails. "stop" raises the first error. "continue" records the candidate error and ranks the remaining usable candidates.
<code>seed</code>	Optional non-negative integer random seed. Candidate fits receive deterministic consecutive seeds derived from this value.
<code>verbose</code>	Whether individual fits should print progress.
<code>...</code>	Additional arguments passed to <code>nn_fit</code> .

Details

`grid` should be a named list whose names are arguments accepted by `nn_fit`. Atomic vectors are expanded as candidate values. Use lists for arguments that are themselves vectors, such as `hidden`.

All candidates share one validation row set drawn before fitting. Candidate seeds vary initialization and training, not the comparison data. Data, seeds and validation-row arguments cannot be `grid` entries. Grid values override the same argument in `...`. Candidates must use a common task and metric, and only finite scores can rank. Larger values are better for accuracy, balanced accuracy, F1, macro F1, and R-squared. Loss, log loss, RMSE, and MAE are minimized. Set `maximize` explicitly to override this behavior. Scoring sample weights are supplied through `...`, not the `grid`. When ranking by loss, candidates must share the loss definition, Huber delta (if used) and target scaler. To compare different losses, choose a common reporting metric such as RMSE. Without a holdout, loss is training data loss without the regularization penalty; it is not an estimate of generalization error.

When `seed` is supplied, candidate fits receive deterministic consecutive seeds. The returned object keeps fitted candidate models; large grids can use a noticeable amount of memory.

By default, candidate errors stop the search. Use `error_action = "continue"` for wider grids where some candidate combinations may be invalid. Failed candidates and candidates without an available score are kept in the results table with `success = FALSE`, an error message, and NA score/rank values.

Value

A `neuralnetwork_tune` object, a list with:

`results` Ranked candidate table, with success and error columns.

The `candidate_id` column indexes models, even after sorting.

`best_model` The selected fitted model.

`best_params` One-row data frame of selected hyperparameters.

models List of fitted candidate models.
 errors Character vector of candidate error messages.
 maximize Logical flag indicating score direction.

See Also

[neuralnetwork-metrics](#), [nn_fit](#), [nn_cv](#)

Examples

```
tuned <- nn_tune(Species ~ ., iris,
  grid = list(hidden = list(4, c(6, 3)), learning_rate = c(0.01)),
  epochs = 3, validation_split = 0.2, seed = 1, verbose = FALSE)
tuned$best_params
```

plot.neuralnetwork *Plot neuralnetwork training loss*

Description

Plot training and optional validation loss from a fitted neuralnetwork model.

Usage

```
## S3 method for class 'neuralnetwork'
plot(x, y = NULL, type = c("loss", "network"),
  main = "Training loss",
  xlab = "Epoch", ylab = "Loss", ...)
```

Arguments

x	A fitted neuralnetwork object.
y	Unused.
type	Either "loss" for training curves or "network" for a compact network diagram.
main, xlab, ylab	Plot labels. Without an explicit xlab, L-BFGS curves are labeled by function evaluation; other optimizers use epochs.
...	Additional arguments passed to plot .

Details

Both loss series determine the vertical range unless ylim is supplied. A single stored diagnostic is drawn as a point. L-BFGS stores the final diagnostic, not the full optimization trajectory.

Value

The input model, invisibly.

Examples

```
fit <- nn_fit(mpg ~ wt + hp, mtcars, hidden = 4, epochs = 5,
             validation_split = 0.2, seed = 1, verbose = FALSE)
plot(fit)
plot(fit, type = "network")
```

predict.neuralnetwork *Predict from a neuralnetwork model*

Description

Predict classes, class probabilities, or numeric responses from a fitted neuralnetwork model.

Usage

```
## S3 method for class 'neuralnetwork'
predict(object, newdata,
        type = c("response", "class", "prob"), ...)
```

Arguments

object	A fitted neuralnetwork object.
newdata	New predictor data. Formula and data-frame fits require the same predictor variables used during fitting, with compatible factor levels.
type	Prediction type. "response" returns class labels for classification and numeric predictions for regression.
...	Unused.

Details

For classification models, type = "response" and type = "class" return class labels. type = "prob" returns a probability matrix with one column per class. For regression models, type = "response" returns numeric predictions on the original response scale. type = "class" and type = "prob" are not valid for regression.

Target scaling is undone, but a formula transformation is not inverted: $\log(y) \sim x$ predicts $\log(y)$, not raw y . Named matrix predictors are matched by fitted column names; missing names are an error. Unnamed matrices must have the fitted width and column order. Stored formula bases and contrasts are reused even if current contrast options differ.

Value

A factor for classification responses, a probability matrix for type = "prob", or numeric predictions for regression.

Examples

```
fit <- nn_fit(Species ~ ., iris, hidden = 4, epochs = 5,
             seed = 1, verbose = FALSE)
predict(fit, iris[1:3, ], type = "class")
predict(fit, iris[1:3, ], type = "prob")
```

```
print.neuralnetwork    Print a neuralnetwork model
```

Description

Print model architecture, optimizer, loss and backend.

Selected-checkpoint training metrics and available validation metrics follow the training progress.
L-BFGS reports function evaluations instead of epochs.

Usage

```
## S3 method for class 'neuralnetwork'
print(x, ...)
```

Arguments

x	A fitted neuralnetwork object.
...	Unused.

Value

The input model, invisibly.

```
summary.neuralnetwork    Summarize neuralnetwork models
```

Description

Return model metadata and selected-checkpoint training metrics, or extract raw fitted parameters.

Usage

```
## S3 method for class 'neuralnetwork'
summary(object, ...)

## S3 method for class 'neuralnetwork'
coef(object, ...)
```

Arguments

object	A fitted neuralnetwork object.
...	Unused.

Details

summary() is intended for a compact training report. For L-BFGS fits it reports function evaluations and the convergence code returned by `optim`, which helps distinguish a normal optimizer stop from an iteration limit. Use `coef()` when you need the raw weight matrices and bias vectors for inspection or custom post-processing. The summary describes the parameters returned for prediction, which may come from an earlier epoch than the last row in the complete training history.

Value

summary() returns a summary object. coef() returns a list of weight matrices and bias vectors.

Index

callback reference, [15](#)
callbacks, [16](#)
coef.neuralnetwork
 (summary.neuralnetwork), [23](#)
compatibility reference, [2](#)
cross-validation, [16](#)

evaluation, [16](#)

model objects, [16](#)

neuralnetwork (neuralnetwork-package), [2](#)
neuralnetwork-callbacks, [3](#)
neuralnetwork-compat, [4](#)
neuralnetwork-metrics, [6](#)
neuralnetwork-objects, [8](#)
neuralnetwork-package, [2](#)
nn_class_ind (neuralnetwork-compat), [4](#)
nn_compute (neuralnetwork-compat), [4](#)
nn_confint (neuralnetwork-compat), [4](#)
nn_cv, [2](#), [3](#), [7](#), [8](#), [10](#), [12](#), [21](#)
nn_evaluate, [2](#), [3](#), [7–9](#), [11](#), [11](#), [17](#), [18](#)
nn_fit, [2–5](#), [8–11](#), [13](#), [20](#), [21](#)
nn_generalized_weights
 (neuralnetwork-compat), [4](#)
nn_gwplot (neuralnetwork-compat), [4](#)
nn_hessian (neuralnetwork-compat), [4](#)
nn_load, [3](#), [8](#)
nn_load (nn_save), [18](#)
nn_multinom (neuralnetwork-compat), [4](#)
nn_permutation_importance, [2](#), [3](#), [8](#), [12](#), [17](#)
nn_save, [3](#), [8](#), [18](#)
nn_tune, [2](#), [3](#), [7](#), [8](#), [11](#), [12](#), [19](#)
nn_which_is_max (neuralnetwork-compat),
 [4](#)

optim, [9](#), [16](#), [24](#)

Permutation importance, [7](#)
plot, [21](#)
plot.neuralnetwork, [21](#)

predict.neuralnetwork, [2](#), [9](#), [22](#)
Prediction, [16](#)
print.neuralnetwork, [23](#)

saveRDS, [18](#)
summary.neuralnetwork, [9](#), [23](#)

tuning, [16](#)