Package: cito (via r-universe)

November 23, 2026

Type Package

Title Building and Training Neural Networks

Version 1.2.1

Description The 'cito' package provides a user-friendly interface for training and interpreting deep neural networks (DNN). 'cito' simplifies the fitting of DNNs by supporting the familiar formula syntax, hyperparameter tuning under cross-validation, and helps to detect and handle convergence problems. DNNs can be trained on CPU, GPU and MacOS GPUs. In addition, 'cito' has many downstream functionalities such as various explainable AI (xAI) metrics (e.g. variable importance, partial dependence plots, accumulated local effect plots, and effect estimates) to interpret trained DNNs. 'cito' optionally provides confidence intervals (and p-values) for all xAI metrics and predictions. At the same time, 'cito' is computationally efficient because it is based on the deep learning framework 'torch'. The 'torch' package is native to R, so no Python installation or other API is required for this package.

Encoding UTF-8

LazyData true

Depends R (>= 4.1.0)

Imports Rcpp, coro, jpeg, png, checkmate, torch, gridExtra, parabar, abind, progress, cli, torchvision, tibble, lme4, rlang, reformulas

License GPL (>= 3)

Suggests spelling, rmarkdown, testthat, plotly, ggraph, igraph, stats, ggplot2, knitr, tiff, raster

LinkingTo Rcpp

SystemRequirements libjpeg, libpng, libtiff (all optional; without them cito falls back to the R image readers, except raster_patches(), which requires libtiff)

VignetteBuilder knitr

BugReports https://github.com/citoverse/cito/issues

URL https://citoverse.github.io/cito/

Language en-US

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Christian Amesöder [aut], Maximilian Pichler [aut, cre] (ORCID: <https://orcid.org/0000-0003-2252-8327>), Florian Hartig [ctb] (ORCID: <https://orcid.org/0000-0002-6255-9059>), Armin Schenk [ctb]

Maintainer Maximilian Pichler

<maximilian.pichler@biologie.uni-regensburg.de>

Config/pak/sysreqs libjpeg-dev libpng-dev libtiff-dev

Repository https://cran.r-universe.dev

Date/Publication 2026-11-23 14:58:12 UTC

RemoteUrl https://github.com/cran/cito

RemoteRef HEAD

RemoteSha 23428aca4dd3d6b0ec5b0726b2bd842fa54cb3b3

Contents

ALE . . . 3
analyze_training . . . 5
avgPool . . . 6
cito . . . 8
cnn . . . 13
coef.citocnn . . . 19
coef.citodnn . . . 20
coef.citommn . . . 21
conditionalEffects . . . 22
config_lr_scheduler . . . 24
config_optimizer . . . 26
config_tuning . . . 27
continue_training . . . 28
conv . . . 30
create_architecture . . . 32
dnn . . . 34
e . . . 47
fe . . . 49
findReTrmClasses . . . 50
linear . . . 50
maxPool . . . 52
mmn . . . 53
PDP . . . 58

plot.citoarchitecture 60
plot.citocnn 61
plot.citodnn 62
plot.citommn 63
predict.citocnn 64
predict.citodnn 66
predict.citommn 67
print.citoarchitecture 68
print.citocnn 69
print.citodnn 70
print.citommn 71
print.conditionalEffects 72
print.summary.citodnn 72
raster_patches 73
residuals.citodnn 76
simulate_shapes 76
summary.citocnn 77
summary.citodnn 77
summary.citommn 79
sumTerms 80
transfer 80
tree_data 82
tune 83

Index **84**

`ALE` *Accumulated Local Effect Plot (ALE)*

Description

Performs an ALE for one or more features.

Usage

```
ALE(
  model,
  variable = NULL,
  data = NULL,
  type = "response",
  analytical = FALSE,
  center = FALSE,
  K = 10,
  ALE_type = c("equidistant", "quantile"),
  plot = TRUE,
  parallel = FALSE,
  ...
)
```

```
## S3 method for class 'citodnn'
ALE(
  model,
  variable = NULL,
  data = NULL,
  type = "response",
  analytical = TRUE,
  center = FALSE,
  K = 10,
  ALE_type = c("quantile", "equidistant"),
  plot = TRUE,
  parallel = FALSE,
  ...
)

## S3 method for class 'citodnnBootstrap'
ALE(
  model,
  variable = NULL,
  data = NULL,
  type = "response",
  analytical = TRUE,
  center = FALSE,
  K = 10,
  ALE_type = c("quantile", "equidistant"),
  plot = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

model	a model created by dnn
variable	variable (as a string) for which the ALE should be computed. If none is supplied, it is computed for all variables.
data	data on which the ALE is computed; if NULL, the training data is used
type	ALE on which scale response or link, default is response
analytical	Analytical ALE based on conditional effects or not
center	center ALE or not (only available for analytical ALE)
K	number of neighborhoods the original feature space is divided into
ALE_type	method by which the feature bins (neighborhoods) are created
plot	plot ALE or not
parallel	parallelize over bootstrap models or not
...	arguments passed to predict

Value

A list of plots made with 'ggplot2' consisting of an individual plot for each defined variable.

Explanation

Accumulated Local Effect plots (ALE) quantify how the predictions change when the features change. They are similar to partial dependency plots but are more robust to feature collinearity.

Mathematical details

If the defined variable is a numeric feature, the ALE is performed. Here, the non centered effect for feature j with k equally distant neighborhoods is defined as:

$\hat{\tilde{f}}_{j,ALE}(x) = \sum_{k=1}^{k_j(x)} \frac{1}{n_j(k)} \sum_{i:x_j^{(i)} \in N_j(k)} \left[\hat{f}(z_{k,j}, x_{\setminus j}^{(i)}) - \hat{f}(z_{k-1,j}, x_{\setminus j}^{(i)}) \right]$

Where $N_j(k)$ is the k-th neighborhood and $n_j(k)$ is the number of observations in the k-th neighborhood.

The last part of the equation, $\left[\hat{f}(z_{k,j}, x_{\setminus j}^{(i)}) - \hat{f}(z_{k-1,j}, x_{\setminus j}^{(i)}) \right]$ represents the difference in model prediction when the value of feature j is exchanged with the upper and lower border of the current neighborhood.

See Also

PDP

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris)

ALE(nn.fit, variable = "Petal.Length")
}
```

---

analyze_training *Visualize training of Neural Network*

Description

After training a model with cito, this function helps to analyze the training process and decide on best performing model. Creates a 'plotly' figure which allows to zoom in and out on training graph

Usage

```
analyze_training(object)
```

Arguments

object	a model created by dnn, cnn or mmn

Details

The baseline loss is the most important reference. If the model was not able to achieve a better (lower) loss than the baseline (which is the loss for a intercept only model), the model probably did not converge. Possible reasons include an improper learning rate, too few epochs, or too much regularization. See the ?dnn help or the `vignette("B-Training_neural_networks")`.

Value

a 'plotly' figure

Examples

```
if(torch::torch_is_installed()){
library(cito)
set.seed(222)
validation_set<- sample(c(1:nrow(datasets::iris)),25)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris[-validation_set,],validation = 0.1)

# show zoomable plot of training and validation losses
analyze_training(nn.fit)

# Use model on validation set
predictions <- predict(nn.fit, iris[validation_set,])

# Scatterplot
plot(iris[validation_set,]$Sepal.Length,predictions)
}
```

---

avgPool *Create an Average Pooling Layer for a CNN Architecture*

Description

This function creates an `avgPool` layer object of class `citolayer` for use in constructing a Convolutional Neural Network (CNN) architecture. The resulting layer object can be passed to the `create_architecture` function to define the structure of the network.

Usage

```
avgPool(kernel_size = NULL, stride = NULL, padding = NULL)
```

Arguments

kernel_size	(integer or tuple) The size of the kernel in this layer. Use a tuple if the kernel size differs across dimensions.
stride	(integer or tuple) The stride of the kernel in this layer. If NULL, the stride is set to the kernel size. Use a tuple if the stride differs across dimensions.
padding	(integer or tuple) The amount of zero-padding added to the input on both sides. Use a tuple if the padding differs across dimensions.

Details

This function creates an avgPool layer object, which represents an average pooling layer in a CNN architecture. Parameters not specified (and thus set to NULL) will be filled with default values provided to the create_architecture function.

Value

An S3 object of class "avgPool" "citolayer", representing an average pooling layer in the CNN architecture.

Author(s)

Armin Schenk

See Also

create_architecture

Examples

```
if(torch::torch_is_installed()){
library(cito)

# A average pooling layer where all available parameters are assigned
# No value will be overwritten by 'create_architecture()'
layer1 <- avgPool(3, 1, 0)

# A average pooling layer where only the kernel size is assigned
# stride and padding are filled with the defaults
# passed to the 'create_architecture()' function
layer2 <- avgPool(kernel_size=4)
}
```

cito *'cito': Building and training neural networks*

Description

The 'cito' package provides a user-friendly interface for training and interpreting deep neural networks (DNN). 'cito' simplifies the fitting of DNNs by supporting the familiar formula syntax, hyperparameter tuning under cross-validation, and helps to detect and handle convergence problems. DNNs can be trained on CPU, GPU and MacOS GPUs. In addition, 'cito' has many downstream functionalities such as various explainable AI (xAI) metrics (e.g. variable importance, partial dependence plots, accumulated local effect plots, and effect estimates) to interpret trained DNNs. 'cito' optionally provides confidence intervals (and p-values) for all xAI metrics and predictions. At the same time, 'cito' is computationally efficient because it is based on the deep learning framework 'torch'. The 'torch' package is native to R, so no Python installation or other API is required for this package.

Details

Cito is built around its main function dnn, which creates and trains a deep neural network. Various tools for analyzing the trained neural network are available.

Installation

in order to install cito please follow these steps:

```
install.packages("cito")

library(torch)

install_torch(reinstall = TRUE)

library(cito)
```

cito functions and typical workflow

- dnn: train deep neural network
- cnn, mmn: train convolutional and multi-modal neural networks
- raster_patches: train cnn() and mmn() on windows read from one large GeoTIFF
- analyze_training: check for convergence by comparing training loss with baseline loss
- continue_training: continues training of an existing cito dnn model for additional epochs
- summary.citodnn: extract xAI metrics/effects to understand how predictions are made
- PDP: plot the partial dependency plot for a specific feature
- ALE: plot the accumulated local effect plot for a specific feature

Check out the vignettes for more details on training NN and how a typical workflow with 'cito' could look like.

Author(s)

Maintainer: Maximilian Pichler <maximilian.pichler@biologie.uni-regensburg.de> (ORCID)

Authors:

- Maximilian Pichler <maximilian.pichler@biologie.uni-regensburg.de> (ORCID)
- Christian Amesöder <Christian.Amesoeder@informatik.uni-regensburg.de>

Other contributors:

- Florian Hartig <florian.hartig@biologie.uni-regensburg.de> (ORCID) [contributor]
- Armin Schenk <armin.schenk99@gmail.com> [contributor]

See Also

Useful links:

- https://citoverse.github.io/cito/
- Report bugs at https://github.com/citoverse/cito/issues

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Example workflow in cito

## Build and train  Network
### cross-entropy (softmax link) for multi-class responses (e.g., Species)
nn.fit<- dnn(Species~., data = datasets::iris, loss = "cross-entropy")

## The training loss is below the baseline loss but at the end of the
## training the loss was still decreasing, so continue training for another 50
## epochs
nn.fit <- continue_training(nn.fit, epochs = 50L)

# Structure of Neural Network
print(nn.fit)

# Plot Neural Network
if(requireNamespace("ggraph", quietly = TRUE)) plot(nn.fit, plot_type = "arch")
## 4 Input nodes (first layer) because of 4 features
## 3 Output nodes (last layer) because of 3 response species (one node for each
## level in the response variable).
## The layers between the input and output layer are called hidden layers (two
## of them)

## We now want to understand how the predictions are made, what are the
## important features? The summary function automatically calculates feature
## importance (the interpretation is similar to an anova) and calculates
```

```
## average conditional effects that are similar to linear effects:
summary(nn.fit)

## To visualize the effect (response-feature effect), we can use the ALE and
## PDP functions

# Partial dependencies
PDP(nn.fit, variable = "Petal.Length")

# Accumulated local effect plots
ALE(nn.fit, variable = "Petal.Length")



# Per se, it is difficult to get confidence intervals for our xAI metrics (or
# for the predictions). But we can use bootstrapping to obtain uncertainties
# for all cito outputs:
## Re-fit the neural network with bootstrapping
nn.fit<- dnn(Species~.,
             data = datasets::iris,
             loss = "cross-entropy",
             epochs = 150L,
             verbose = FALSE,
             bootstrap = 20L)
## convergence can be tested via the analyze_training function
analyze_training(nn.fit)

## Summary for xAI metrics (can take some time):
summary(nn.fit, importance = "permutation", type = "link")
## Now with standard errors and p-values
## Note: Take the p-values with a grain of salt! We do not know yet if they are
## correct (e.g. if you use regularization, they are likely conservative == too
## large)

## Predictions with bootstrapping:
dim(predict(nn.fit))
## predictions are by default averaged (over the bootstrap samples)

## Multinomial and conditional logit regression
m = dnn(Species~., data = iris, loss = "clogit", lr = 0.01)
m = dnn(Species~., data = iris, loss = "multinomial", lr = 0.01)

Y = t(stats::rmultinom(100, 10, prob = c(0.2, 0.2, 0.5)))
m = dnn(cbind(X1, X2, X3)~., data = data.frame(Y, A = as.factor(runif(100))),
        loss = "multinomial", lr = 0.01)
## conditional logit for size > 1 is not supported yet


## Binary response (0/1 or a two-level factor)
iris2 = transform(iris, setosa = as.integer(Species == "setosa"))
m = dnn(setosa ~ Sepal.Length + Sepal.Width, data = iris2,
     loss = "bernoulli", optimizer = "ignite_adam", lr = 0.01, epochs = 20L, verbose = FALSE)
```

```
## Output of the last hidden layer instead of the predictions
dim(predict(m, return_embeddings = TRUE))

## Blocked bootstrap: resample whole groups instead of rows
m = dnn(Sepal.Length ~ Petal.Length + Petal.Width, data = iris, epochs = 20L,
        bootstrap = 5L, bootstrap_blocking_variable = iris$Species, verbose = FALSE)

## Hooks are evaluated after every epoch inside the training loop, where
## 'model' and 'epoch' are visible; their results are collected in hooks_result
m = dnn(Sepal.Length ~ ., data = iris, epochs = 5L, verbose = FALSE,
        hooks = list(function() model$losses$train_l[epoch]))
unlist(m$hooks_result)

## Observation weights are passed to a custom likelihood with a weights argument
weighted_mse = function(pred, true, weights = NULL) {
  l = (pred - true)^2
  if(is.null(weights)) l else l * weights
}
m = dnn(Sepal.Length ~ ., data = iris, loss = weighted_mse, epochs = 5L,
        weights = runif(150), verbose = FALSE)

# Hyperparameter tuning (experimental feature)
hidden_values = matrix(c(5, 2,
                         4, 2,
                         10,2,
                         15,2), 4, 2, byrow = TRUE)
## Potential architectures we want to test, first column == number of nodes
print(hidden_values)

nn.fit = dnn(Species~.,
             data = iris,
             epochs = 30L,
             loss = "cross-entropy",
             hidden = tune(values = hidden_values),
             lr = tune(0.00001, 0.1) # tune lr between range 0.00001 and 0.1
             )
## Tuning results:
print(nn.fit$tuning)

# test = Inf means that tuning was cancelled after only one fit (within the CV)


# Advanced: Custom loss functions and additional parameters
## Normal Likelihood with sd parameter:
custom_loss = function(pred, true) {
  logLik = torch::distr_normal(pred,
                               scale = torch::nnf_relu(scale)+
                                 0.001)$log_prob(true)
  return(-logLik$mean())
}

nn.fit<- dnn(Sepal.Length~.,
             data = datasets::iris,
```

```
             loss = custom_loss,
             verbose = FALSE,
             custom_parameters = list(scale = 1.0)
)
nn.fit$loss$parameters$scale

## Multivariate normal likelihood with parametrized covariance matrix
## Sigma = L*L^t + D
## Helper function to build covariance matrix
create_cov = function(LU, Diag) {
  return(torch::torch_matmul(LU, LU$t()) + torch::torch_diag(Diag$exp()+0.01))
}

custom_loss_MVN = function(true, pred) {
  Sigma = create_cov(SigmaPar, SigmaDiag)
  logLik = torch::distr_multivariate_normal(pred,
                                            covariance_matrix = Sigma)$
    log_prob(true)
  return(-logLik$mean())
}


nn.fit<- dnn(cbind(Sepal.Length, Sepal.Width, Petal.Length)~.,
             data = datasets::iris,
             lr = 0.01,
             verbose = FALSE,
             loss = custom_loss_MVN,
             custom_parameters =
               list(SigmaDiag =  rep(0, 3),
                    SigmaPar = matrix(rnorm(6, sd = 0.001), 3, 2))
)
as.matrix(create_cov(nn.fit$loss$parameters$SigmaPar,
                     nn.fit$loss$parameters$SigmaDiag))


# Embeddings and fixed effects
## Species enters as an embedding: each species gets two learned coordinates,
## from which the network learns how the species differ in their response to
## all predictors. Petal.Width enters as a fixed (linear) effect.
nn.fit = dnn(Sepal.Length ~ Sepal.Width + Petal.Length + fe(Petal.Width) + e(Species, dim = 2),
             data = datasets::iris, loss = "mse", epochs = 50L, verbose = FALSE)

## the linear effect of Petal.Width
coef(nn.fit)$fixed_effects

## the embeddings, one row per species
coef(nn.fit)$net_parameters$e_1.weight

## architecture: Species feeds its two embedding dimensions, Petal.Width goes
## straight to the output
if(requireNamespace("ggraph", quietly = TRUE)) plot(nn.fit, plot_type = "arch")
}
```

cnn *Train a Convolutional Neural Network (CNN)*

Description

This function trains a Convolutional Neural Network (CNN) on the provided input data `X` and the target data `Y` using the specified architecture, loss function, and optimizer.

Usage

```
cnn(
  X,
  Y = NULL,
  architecture = NULL,
  custom = NULL,
 loss = c("mse", "mae", "cross-entropy", "bernoulli", "gaussian", "binomial", "poisson",
    "mvp", "nbinom", "multinomial", "clogit", "softmax"),
  custom_parameters = NULL,
 optimizer = c("sgd", "adam", "adadelta", "adagrad", "rmsprop", "rprop", "ignite_adam"),
  lr = 0.01,
  lr_scheduler = NULL,
  alpha = 0.5,
  lambda = 0,
  validation = 0,
  batchsize = NULL,
  shuffle = TRUE,
  data_augmentation = NULL,
  num_workers = 0L,
  num_threads = 4L,
  epochs = 100,
  weights = NULL,
  early_stopping = Inf,
  burnin = Inf,
  baseloss = NULL,
  device = c("cpu", "cuda", "mps"),
  plot = TRUE,
  verbose = TRUE
)
```

Arguments

`X`	An array of input data with a minimum of 3 and a maximum of 5 dimensions. The first dimension represents the samples, the second dimension represents the channels, and the third to fifth dimensions represent the input dimensions. As an alternative, you can provide the relative or absolute path to the folder containing the images. In this case, unsigned integer images are divided by their maximum value (255 for 8 bit, 65535 for 16 bit), and floating point and signed

integer TIFFs are passed unchanged. Windows of one large raster are given with raster_patches.

Y — The target data. The allowed formats of the target data differ between loss functions. See dnn for more information.

architecture — An object of class 'citoarchitecture'. See create_architecture for more information. Ignored if custom is provided.

custom — custom network architecture, replaces the architecture argument. Either an nn_module generator, an instantiated nn_module, or a list(module = generator, args = list(...), output_dim = integer). If the initialize method has arguments named input or output, they are filled in automatically (input = input shape, e.g. c(channels, height, width), output = output dimension expected by the loss). The forward method receives a float tensor of shape (batch, input_shape) and must return a tensor of shape (batch, output). See dnn for details

loss — The loss function to be used. Options include "mse", "mae", "cross-entropy", "bernoulli", "gaussian", "binomial", "poisson", "nbinom", "mvp", "multinomial", and "clogit". You can also specify your own loss function. See Details for more information. Default is "mse".

custom_parameters — Parameters for the custom loss function. See the vignette for an example. Default is NULL.

optimizer — The optimizer to be used. Options include "sgd", "adam", "adadelta", "adagrad", "rmsprop", "rprop", and "ignite_adam". See config_optimizer for further adjustments to the optimizer. Default is "sgd".

lr — Learning rate for the optimizer. Default is 0.01.

lr_scheduler — Learning rate scheduler. See config_lr_scheduler for creating a learning rate scheduler. Default is NULL.

alpha — Alpha value for L1/L2 regularization. Default is 0.5.

lambda — Lambda value for L1/L2 regularization. Default is 0.0.

validation — Proportion of the data to be used for validation. Alternatively, a vector containing the indices of the validation samples can be provided. Default is 0.0.

batchsize — Batch size for training. If NULL, batchsize is 10% of the training data. Default is NULL.

shuffle — Whether to shuffle the data before each epoch. Default is TRUE.

data_augmentation — A list of functions used for data augmentation. Elements must be either functions or strings corresponding to inbuilt data augmentation functions. See details for more information.

num_workers — (integer) Number of worker processes for the dataloader. Only effective for folder inputs; raster inputs from raster_patches always use 0 and prefetch in a reader thread instead. 0 decodes in the main process.

num_threads — (integer) Number of threads the compiled image decoder uses per batch for folder inputs, and the number of reader threads for raster inputs from raster_patches.

epochs — Number of epochs to train the model. Default is 100.

weights — weights or other values (can be also a matrix) that the likelihood has access to

`early_stopping` Number of epochs with no improvement after which training will be stopped. Default is Inf.

`burnin` Number of epochs after which the training stops if the loss is still above the baseloss. Default is Inf.

`baseloss` Baseloss used for burnin and plot. If NULL, the baseloss corresponds to intercept only models. Default is NULL.

`device` Device to be used for training. Options are "cpu", "cuda", and "mps". Default is "cpu".

`plot` Whether to plot the training progress. Default is TRUE.

`verbose` Whether to print detailed training progress. Default is TRUE.

Value

An S3 object of class `"citocnn"` is returned. It is a list containing everything there is to know about the model and its training process. The list consists of the following attributes:

`net` An object of class "nn_module". Originates from the torch package and represents the core object of this workflow.

`call` The original function call.

`loss` An object of class "nn_module". Contains all relevant information for the loss function, e.g. parameters and a function (format_Y) that transforms target data.

`data` A list. Contains the data used for the training of the model.

`model_properties`
A list of properties, that define the architecture of the model.

`training_properties`
A list of all training hyperparameters used the last time the model was trained.

`losses` A data.frame containing training and validation losses of each epoch.

`best_epoch_net_state_dict`
Serialized state dict of net from the best training epoch.

`best_epoch_loss_state_dict`
Serialized state dict of loss from the best training epoch.

`last_epoch_net_state_dict`
Serialized state dict of net from the last training epoch.

`last_epoch_loss_state_dict`
Serialized state dict of loss from the last training epoch.

`use_model_epoch`
String, either "best" or "last". Determines whether the parameters (e.g. weights, biases) from the best or the last training epoch are used (e.g. for prediction).

`loaded_model_epoch`
String, shows from which training epoch the parameters are currently loaded in `net` and `loss`.

Details:

Also check dnn for details to common arguments.

Convolutional Neural Networks:

Convolutional Neural Networks (CNNs) are a specialized type of neural network designed for processing structured data, such as images. The key components of a CNN are convolutional layers, pooling layers and fully-connected (linear) layers:

- **Convolutional layers** are the core building blocks of CNNs. They consist of filters (also called kernels), which are small, learnable matrices. These filters slide over the input data to perform element-wise multiplication, producing feature maps that capture local patterns and features. Multiple filters are used to detect different features in parallel. They help the network learn hierarchical representations of the input data by capturing low-level features (edges, textures) and gradually combining them (in subsequent convolutional layers) to form higher-level features.
- **Pooling layers** reduce the size of the feature maps created by convolutional layers, while retaining important information. A common type is max pooling, which keeps the highest value in a region, simplifying the data while preserving essential features.
- **Fully-connected (linear) layers** connect every neuron in one layer to every neuron in the next layer. These layers are found at the end of the network and are responsible for combining high-level features to make final predictions.

The architecture of the CNN that will be created and trained by this function is defined by an object of class 'citoarchitecture'. See `create_architecture` for detailed information on how to define and customize your CNN architecture.

Data Augmentation

Data augmentation is a technique used to improve the generalization of convolutional neural networks (CNNs) by increasing the diversity of the training data through random transformations. This function supports data augmentation through the `data_augmentation` argument, which accepts a list containing either user-defined functions or the names of cito's built-in data augmentation functions. Each user-defined function must take a `torch_tensor` as input and return a `torch_tensor` with the same shape. The input tensor must have 3 to 5 dimensions with the following structure:

- Dimension 1: singleton batch dimension (i.e., size 1),
- Dimension 2: channel dimension,
- Dimensions 3 to 5: spatial dimensions (e.g., X, Y, Z).

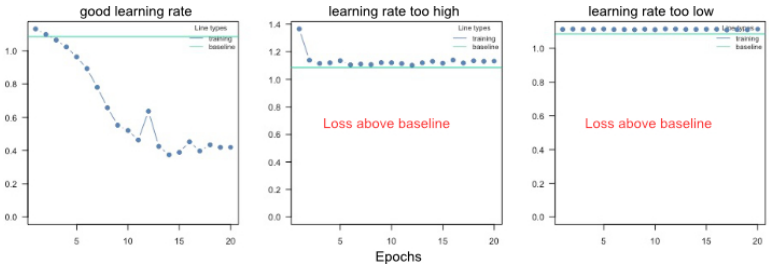
During training, the data loader re-loads each sample at every epoch, applying all provided augmentation functions sequentially each time the sample is accessed. This allows transformations to vary across epochs if the functions include randomness (e.g., randomly flipping a spatial axis) helping the model learn invariance to such changes. In addition to custom functions, the list can contain the names (as strings) of the following built-in augmentation methods:

- `"rotate90"`:
  - For 2D convolutions: randomly applies one of the 4 possible 90° rotations. The X and Y dimensions have to be equal.
  - For 3D convolutions:
    - If X, Y, and Z dimensions are equal: randomly applies one of the 24 possible 90° rotations.

    * If only two spatial dimensions are equal: randomly applies one of the 4 possible 90° rotations in the plane of the two spatial dimensions.
  - Not available for 1D convolutions.
- `"flip"`: Randomly flips each spatial dimension independently with 50% probability.
- `"noise"`: Adds a small amount of normally distributed noise to the tensor.

Training and convergence of neural networks

Ensuring convergence can be tricky when training neural networks. Their training is sensitive to a combination of the learning rate (how much the weights are updated in each optimization step), the batch size (a random subset of the data is used in each optimization step), and the number of epochs (number of optimization steps). Typically, the learning rate should be decreased with the size of the neural networks (depth of the network and width of the hidden layers). We provide a baseline loss (intercept only model) that can give hints about an appropriate learning rate:



If the training loss of the model doesn't fall below the baseline loss, the learning rate is either too high or too low. If this happens, try higher and lower learning rates.

A common strategy is to try (manually) a few different learning rates to see if the learning rate is on the right scale.

See the troubleshooting vignette (`vignette("B-Training_neural_networks")`) for more help on training and debugging neural networks.

Author(s)

Armin Schenk

See Also

`predict.citocnn`, `print.citocnn`, `plot.citocnn`, `summary.citocnn`, `coef.citocnn`, `continue_training`, `analyze_training`

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Example workflow in cito

device <- ifelse(torch::cuda_is_available(), "cuda", "cpu")

## Data
```

```
### We generate our own data:
### 320 images (3x50x50) of either rectangles or ellipsoids
shapes <- cito:::simulate_shapes(n=320, size=50, channels=3)
X <- shapes$data
Y <- shapes$labels

## Architecture
### Declare the architecture of the CNN
### Note that the output layer is added automatically by cnn()
architecture <- create_architecture(conv(5), maxPool(), conv(5), maxPool(), linear(10))

## Build and train network
### cross-entropy (softmax link) for classification
cnn.fit <- cnn(X, Y, architecture, loss = "cross-entropy",
               epochs = 2, validation = 0.1, lr = 0.05, device=device)

## The training loss is below the baseline loss but at the end of the
## training the loss was still decreasing, so continue training for another 5
## epochs
cnn.fit <- continue_training(cnn.fit, epochs = 5)

# Structure of Neural Network
print(cnn.fit)

# Plot Neural Network
plot(cnn.fit)

## Convergence can be tested via the analyze_training function
analyze_training(cnn.fit)

## Predictions, and the output of the last hidden layer (embeddings)
predict(cnn.fit, newdata = X[1:5,,,,drop = FALSE], type = "class")
dim(predict(cnn.fit, newdata = X[1:5,,,,drop = FALSE], return_embeddings = TRUE))

## Data augmentation: random 90 degree rotations and flips of every image,
## drawn anew in each epoch
cnn.aug <- cnn(X, Y, architecture, loss = "cross-entropy", epochs = 2,
               data_augmentation = list("rotate90", "flip"), device = device)

## Images on disk: X is a folder (jpeg, png or tiff), read batch by batch
## during training, so the data do not have to fit into memory
if(requireNamespace("tiff", quietly = TRUE)) {
  folder <- file.path(tempdir(), "shapes")
  dir.create(folder, showWarnings = FALSE)
  for(i in 1:50) {
   tiff::writeTIFF(aperm(X[i,,,], c(2, 3, 1)), file.path(folder, sprintf("%03d.tiff", i)))
  }
  # files are matched to Y in the order of list.files(folder)
  cnn.disk <- cnn(folder, Y[1:50], architecture, loss = "cross-entropy",
                  epochs = 1, device = device)
  predict(cnn.disk, newdata = folder, type = "class")[1:5]
  unlink(folder, recursive = TRUE)
}
```

```
## Windows cut from one large georeferenced raster around each observation
## are an input as well, see ?raster_patches

## Transfer learning
### With the transfer() function we can use predefined architectures with pretrained weights
transfer_architecture <- create_architecture(transfer("resnet18"))
resnet <- cnn(X, Y, transfer_architecture, loss = "cross-entropy",
              epochs = 1, validation = 0.1, lr = 0.05, device=device)
print(resnet)
plot(resnet)
}
```

`coef.citocnn` *Retrieve parameters of a fitted CNN model*

Description

This function returns the list of parameters (weights and biases) and buffers (e.g. running mean and variance of batch normalization layers) currently in use by the neural network model created using the cnn function.

Usage

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

Arguments

`object`	A model created by cnn.
`...`	Additional arguments (currently not used).

Value

A list with up to three components:

- `net_parameters`: A list of the model's weights and biases for the currently used model epoch.
- `net_buffers`: A list of buffers (e.g., running statistics) for the currently used model epoch.
- `loss_parameters`: A list of the loss function's parameters for the currently used model epoch.

Examples

```
if(torch::torch_is_installed()){
library(cito)

device <- ifelse(torch::cuda_is_available(), "cuda", "cpu")
```

```
set.seed(222)

## Data
shapes <- cito:::simulate_shapes(320, 28)
X <- shapes$data
Y <- shapes$labels

## Architecture
architecture <- create_architecture(conv(5), maxPool(), conv(5), maxPool(), linear(10))

## Build and train network
cnn.fit <- cnn(X, Y, architecture, loss = "cross-entropy",
               epochs = 50, validation = 0.1, lr = 0.05, device=device)

# Weights of neural network
coef(cnn.fit)
}
```

`coef.citodnn` *Returns list of parameters the neural network model currently has in use*

Description

Returns list of parameters the neural network model currently has in use

Usage

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

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

Arguments

`object`	a model created by dnn
`...`	nothing implemented yet

Value

list of parameters of neural network and loss

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)
validation_set<- sample(c(1:nrow(datasets::iris)),25)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris[-validation_set,])

# Structure of Neural Network
print(nn.fit)

#analyze weights of Neural Network
coef(nn.fit)
}
```

`coef.citommn` *Retrieve parameters of a fitted MMN model*

Description

This function returns the list of parameters (weights and biases) and buffers (e.g. running mean and variance of batch normalization layers) currently in use by the neural network model created using the `mmn` function.

Usage

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

Arguments

`object`	A model created by `mmn`.
`...`	Additional arguments (currently not used).

Value

A list with two components:

- `net_parameters`: A list of the model's weights and biases for the currently used model epoch.
- `net_buffers`: A list of buffers (e.g., running statistics) for the currently used model epoch.
- `loss_parameters`: A list of the loss function's parameters for the currently used model epoch.

`conditionalEffects` *Calculate average conditional effects*

Description

Average conditional effects calculate the local derivatives for each observation for each feature. They are similar to marginal effects. And the average of these conditional effects is an approximation of linear effects (see Pichler and Hartig, 2023 for more details). You can use this function to either calculate main effects (on the diagonal, take a look at the example) or interaction effects (off-diagonals) between features.

To obtain uncertainties for these effects, enable the bootstrapping option in the `dnn(..)` function (see example).

Usage

```
conditionalEffects(
  object,
  method = c("analytical", "finite"),
  subsample = FALSE,
  interactions = FALSE,
  epsilon = 0.1,
  device = c("cpu", "cuda", "mps"),
  indices = NULL,
  data = NULL,
  batchsize = NULL,
  type = "response",
  return_vars = FALSE,
  ...
)

## S3 method for class 'citodnn'
conditionalEffects(
  object,
  method = c("analytical", "finite"),
  subsample = FALSE,
  interactions = FALSE,
  epsilon = 0.1,
  device = c("cpu", "cuda", "mps"),
  indices = NULL,
  data = NULL,
  batchsize = NULL,
  type = "response",
  return_vars = FALSE,
  ...
)
```

```
## S3 method for class 'citodnnBootstrap'
conditionalEffects(
  object,
  method = c("analytical", "finite"),
  subsample = FALSE,
  interactions = FALSE,
  epsilon = 0.1,
  device = c("cpu", "cuda", "mps"),
  indices = NULL,
  data = NULL,
  batchsize = NULL,
  type = "response",
  return_vars = FALSE,
  ...
)
```

Arguments

`object`	object of class `citodnn`
`method`	Calculate the conditional effects analytically or via the finite difference
`subsample`	subsample data to decrease computational runtime, must be either FALSE or in the range of `[0,1]`
`interactions`	calculate interactions or not (computationally expensive)
`epsilon`	difference used to calculate derivatives
`device`	which device
`indices`	of variables for which the ACE are calculated
`data`	data which is used to calculate the ACE
`batchsize`	batchsize
`type`	ACE on which scale (response or link)
`return_vars`	return variances of variables (internally required)
`...`	additional arguments that are passed to the predict function

Value

an S3 object of class `"conditionalEffects"` is returned. The list consists of the following attributes:

`result`	3-dimensional array with the raw results
`mean`	Matrix, average conditional effects
`abs`	Matrix, summed absolute conditional effects
`sd`	Matrix, standard deviation of the conditional effects

Author(s)

Maximilian Pichler

References

Scholbeck, C. A., Casalicchio, G., Molnar, C., Bischl, B., & Heumann, C. (2022). Marginal effects for non-linear prediction functions. arXiv preprint arXiv:2201.08837.

Pichler, M., & Hartig, F. (2023). Can predictive models be used for causal inference?. arXiv preprint arXiv:2306.10551.

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Build and train  Network
nn.fit = dnn(Sepal.Length~., data = datasets::iris)

# Calculate average conditional effects
ACE = conditionalEffects(nn.fit)

## Main effects (categorical features are not supported)
ACE

## With interaction effects:
ACE = conditionalEffects(nn.fit, interactions = TRUE)
## The off diagonal elements are the interaction effects
ACE[[1]]$mean
## ACE is a list, elements correspond to the number of response classes
## Sepal.length == 1 Response so we have only one
## list element in the ACE object

# Re-train NN with bootstrapping to obtain standard errors
nn.fit = dnn(Sepal.Length~., data = datasets::iris, bootstrap = 30L)
## The summary method calculates also the conditional effects, and if
## bootstrapping was used, it will also report standard errors and p-values:
summary(nn.fit)


}
```

---

config_lr_scheduler *Creation of customized learning rate scheduler objects*

Description

Helps create custom learning rate schedulers for dnn.

Usage

```
config_lr_scheduler(
  type = c("lambda", "multiplicative", "reduce_on_plateau", "one_cycle", "step"),
  verbose = FALSE,
  ...
)
```

Arguments

type	String defining which type of scheduler should be used. See Details.
verbose	If TRUE, additional information about scheduler will be printed to console.
...	additional arguments to be passed to scheduler. See Details.

Details

different learning rate scheduler need different variables, these functions will tell you which variables can be set:

- lambda: lr_lambda
- multiplicative: lr_multiplicative
- reduce_on_plateau: lr_reduce_on_plateau
- one_cycle: lr_one_cycle
- step: lr_step

Value

object of class cito_lr_scheduler to give to dnn

Examples

```
if(torch::torch_is_installed()){
library(cito)

# create learning rate scheduler object
scheduler <- config_lr_scheduler(type = "step",
                         step_size = 30,
                         gamma = 0.15,
                         verbose = TRUE)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris, lr_scheduler = scheduler)

}
```

config_optimizer *Creation of customized optimizer objects*

Description

Helps you create custom optimizer for dnn. It is recommended to set learning rate in dnn.

Usage

```
config_optimizer(
  type = c("adam", "adadelta", "adagrad", "rmsprop", "rprop", "sgd", "ignite_adam"),
  verbose = FALSE,
  ...
)
```

Arguments

type	character string defining which optimizer should be used. See Details.
verbose	If TRUE, additional information about scheduler will be printed to console
...	additional arguments to be passed to optimizer. See Details.

Details

different optimizer need different variables, this function will tell you how the variables are set. For more information see the corresponding functions:

- adam: optim_adam
- adadelta: optim_adadelta
- adagrad: optim_adagrad
- rmsprop: optim_rmsprop
- rprop: optim_rprop
- sgd: optim_sgd

Value

object of class cito_optim to give to dnn

Examples

```
if(torch::torch_is_installed()){
library(cito)

# create optimizer object
opt <- config_optimizer(type = "adagrad",
                        lr_decay = 1e-04,
                        weight_decay = 0.1,
```

```
                        verbose = TRUE)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris, optimizer = opt)

}
```

---

config_tuning *Config hyperparameter tuning*

Description

Config hyperparameter tuning

Usage

```
config_tuning(
  CV = 5,
  steps = 10,
  parallel = FALSE,
  NGPU = 1,
  cancel = TRUE,
  bootstrap_final = NULL,
  bootstrap_parallel = FALSE,
  blocking = NULL,
  return_models = FALSE
)
```

Arguments

CV	numeric, specifies k-folded cross validation
steps	numeric, number of random tuning steps
parallel	numeric, number of parallel cores (tuning steps are parallelized)
NGPU	numeric, set if more than one GPU is available, tuning will be parallelized over CPU cores and GPUs, only works for NCPU > 1
cancel	CV/tuning for specific hyperparameter set if model cannot reduce loss below baseline after burnin or returns NA loss
bootstrap_final	bootstrap final model, if all models should be boostrapped it must be set globally via the bootstrap argument in the dnn() function
bootstrap_parallel	should the bootstrapping be parallelized or not
blocking	blocking variable, must be a factor
return_models	return individual models

Details

Note that hyperparameter tuning can be expensive. We have implemented an option to parallelize hyperparameter tuning, including parallelization over one or more GPUs (the hyperparameter evaluation is parallelized, not the CV). This can be especially useful for small models. For example, if you have 4 GPUs, 20 CPU cores, and 20 steps (random samples from the random search), you could run 'dnn(..., device="cuda",lr = tune(), batchsize=tune(), tuning=config_tuning(parallel=20, NGPU=4)', which will distribute 20 model fits across 4 GPUs, so that each GPU will process 5 models (in parallel).

Value

A list with the tuning settings, passed to the `tuning` argument of dnn.

`continue_training` *Continues training of a model generated with* dnn, cnn *or* mmn *for additional epochs.*

Description

If the training/validation loss is still decreasing at the end of the training, it is often a sign that the NN has not yet converged. You can use this function to continue training instead of re-training the entire model.

Usage

```
continue_training(model, ...)

## S3 method for class 'citodnn'
continue_training(
  model,
  epochs = 32,
  data = NULL,
  device = NULL,
  changed_params = NULL,
  init_optimizer = TRUE,
  X = NULL,
  Y = NULL,
  weights = NULL,
  ...
)

## S3 method for class 'citodnnBootstrap'
continue_training(
  model,
  epochs = 32,
  data = NULL,
  device = NULL,
```

```
  changed_params = NULL,
  parallel = FALSE,
  init_optimizer = TRUE,
  X = NULL,
  Y = NULL,
  ...
)

## S3 method for class 'citocnn'
continue_training(
  model,
  epochs = 32,
  X = NULL,
  Y = NULL,
  weights = NULL,
  device = NULL,
  changed_params = NULL,
  init_optimizer = TRUE,
  ...
)

## S3 method for class 'citommn'
continue_training(
  model,
  epochs = 32,
  dataList = NULL,
  device = NULL,
  changed_params = NULL,
  init_optimizer = TRUE,
  weights = NULL,
  ...
)
```

Arguments

model	a model created by dnn, cnn or mmn
...	class-specific arguments
epochs	additional epochs the training should continue for
data	matrix or data.frame. If not provided data from original training will be used
device	can be used to overwrite device used in previous training
changed_params	list of arguments to change compared to original training setup, see dnn which parameter can be changed
init_optimizer	re-initialize optimizer or not
X	Predictor data. If not provided X from original training will be used
Y	Target data. If not provided Y from original training will be used

weights	observation weights (vector or matrix) passed to the likelihood. If not provided, the weights from the original training are reused when continuing on the original data
parallel	train bootstrapped model in parallel
dataList	A list containing the data for training the model. The list should contain all variables used in the formula. If not provided dataList from original training will be used

Value

a model of class citodnn, citodnnBootstrap, citocnn or citommn created by dnn, cnn or mmn

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)
validation_set<- sample(c(1:nrow(datasets::iris)),25)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris[-validation_set,], epochs = 32)

# continue training for another 32 epochs
nn.fit<- continue_training(nn.fit,epochs = 32)

# Use model on validation set
predictions <- predict(nn.fit, iris[validation_set,])
}
```

---

conv *Create a Convolutional Layer for a CNN Architecture*

Description

This function creates a conv layer object of class citolayer for use in constructing a Convolutional Neural Network (CNN) architecture. The resulting layer object can be passed to the create_architecture function to define the structure of the network.

Usage

```
conv(
  n_kernels = NULL,
  kernel_size = NULL,
  stride = NULL,
  padding = NULL,
  dilation = NULL,
```

```
  bias = NULL,
  activation = NULL,
  normalization = NULL,
  dropout = NULL
)
```

Arguments

n_kernels	(integer) The number of kernels (or filters) in this layer.
kernel_size	(integer or tuple) The size of the kernels in this layer. Use a tuple if the kernel size is different in each dimension.
stride	(integer or tuple) The stride of the kernels in this layer. If NULL, the default_stride of create_architecture is used (1 for convolutional layers). Use a tuple if the stride is different in each dimension.
padding	(integer or tuple) The amount of zero-padding added to the input on both sides. Use a tuple if the padding is different in each dimension.
dilation	(integer or tuple) The dilation of the kernels in this layer. Use a tuple if the dilation is different in each dimension.
bias	(boolean) If TRUE, a learnable bias is added to the kernels of this layer.
activation	(character) The activation function applied after this layer. Supported activation functions include "relu", "leaky_relu", "tanh", "elu", "rrelu", "prelu", "softplus", "celu", "selu", "gelu", "relu6", "sigmoid", "softsign", "hardtanh", "tanhshrink", "softshrink", "hardshrink", "log_sigmoid".
normalization	(boolean) If TRUE, batch normalization is applied after this layer.
dropout	(numeric) The dropout rate for this layer. Set to 0 to disable dropout.

Details

This function creates a conv layer object, which is used to define a convolutional layer in a CNN architecture. Parameters that are not specified (and thus set to NULL) will be filled with default values provided to the create_architecture function.

Value

An S3 object of class "conv" "citolayer", representing a convolutional layer in the CNN architecture.

Author(s)

Armin Schenk

See Also

create_architecture

Examples

```
if(torch::torch_is_installed()){
library(cito)

# A convolutional layer where all available parameters are assigned
# No value will be overwritten by 'create_architecture()'
layer1 <- conv(10, 3, 1, 0, 1, TRUE, "relu", FALSE, 0.5)

# A convolutional layer where only the activation function is assigned
# n_kernels, kernel_size, stride, padding, dilation, bias,
# normalization and dropout are filled with the defaults
# passed to the 'create_architecture()' function
layer2 <- conv(activation="selu")
}
```

`create_architecture` *Create a CNN Architecture*

Description

This function constructs a `citoarchitecture` object that defines the architecture of a Convolutional Neural Network (CNN). The `citoarchitecture` object can be used by the `cnn` function to specify the structure of the network, including layer types, parameters, and default values.

Usage

```
create_architecture(
  ...,
  default_n_neurons = 10,
  default_n_kernels = 10,
  default_kernel_size = list(conv = 3, maxPool = 2, avgPool = 2),
  default_stride = list(conv = 1, maxPool = NULL, avgPool = NULL),
  default_padding = list(conv = 0, maxPool = 0, avgPool = 0),
  default_dilation = list(conv = 1, maxPool = 1),
  default_bias = list(conv = TRUE, linear = TRUE),
  default_activation = list(conv = "relu", linear = "relu"),
  default_normalization = list(conv = FALSE, linear = FALSE),
  default_dropout = list(conv = 0, linear = 0)
)
```

Arguments

`...`	Objects of class `citolayer` created by `linear`, `conv`, `maxPool`, `avgPool`, or `transfer`. These layers define the architecture of the CNN.
`default_n_neurons`	(integer) Default number of neurons in a linear layer. Default is 10.

default_n_kernels
: (integer) Default number of kernels in a convolutional layer. Default is 10.

default_kernel_size
: (integer or tuple) Default size of kernels in convolutional and pooling layers. Can be a single integer or a tuple if sizes differ across dimensions. Default is `list(conv = 3, maxPool = 2, avgPool = 2)`.

default_stride
: (integer or tuple) Default stride of kernels in convolutional and pooling layers. Can be a single integer, a tuple if strides differ across dimensions, or `NULL` to use the kernel size. Default is `list(conv = 1, maxPool = NULL, avgPool = NULL)`.

default_padding
: (integer or tuple) Default zero-padding added to both sides of the input. Can be a single integer or a tuple if padding differs across dimensions. Default is `list(conv = 0, maxPool = 0, avgPool = 0)`.

default_dilation
: (integer or tuple) Default dilation of kernels in convolutional and max pooling layers. Can be a single integer or a tuple if dilation differs across dimensions. Default is `list(conv = 1, maxPool = 1)`.

default_bias
: (boolean) Default value indicating if a learnable bias should be added to neurons of linear layers and kernels of convolutional layers. Default is `list(conv = TRUE, linear = TRUE)`.

default_activation
: (character) Default activation function used after linear and convolutional layers. Supported activation functions include "relu", "leaky_relu", "tanh", "elu", "rrelu", "prelu", "softplus", "celu", "selu", "gelu", "relu6", "sigmoid", "softsign", "hardtanh", "tanhshrink", "softshrink", "hardshrink", "log_sigmoid". Default is `list(conv = "relu", linear = "relu")`.

default_normalization
: (boolean) Default value indicating if batch normalization should be applied after linear and convolutional layers. Default is `list(conv = FALSE, linear = FALSE)`.

default_dropout
: (numeric) Default dropout rate for linear and convolutional layers. Set to 0 for no dropout. Default is `list(conv = 0.0, linear = 0.0)`.

Details

This function creates a `citoarchitecture` object that outlines the CNN's architecture based on the provided layers and default parameters. The final architecture consists of layers in the order they are provided. Any unspecified parameters in the `citolayer` objects are filled with the provided default values for their respective layer types. Defaults can be specified for each layer type individually or for all layers at once.

Value

An S3 object of class `"citoarchitecture"` that encapsulates the architecture of the CNN.

Author(s)

Armin Schenk

See Also

cnn, linear, conv, maxPool, avgPool, transfer, print.citoarchitecture, plot.citoarchitecture

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Convolutional layers with different n_kernels and kernel_sizes
c1 <- conv(n_kernels = 8, kernel_size = 5)
c2 <- conv(n_kernels = 16, kernel_size = 3)

# Linear layer
l <- linear(n_neurons = 100)

# MaxPooling layer
mP <- maxPool(kernel_size = 2)

# Create the architecture by using the created layers
# Change the defaults with which the not assigned layer parameters will be filled e.g.
# change default dropout to different values for linear and convolutional layer
# only change the default normalization for linear layers
# change default activation of both linear and convolutional layers to 'selu'
architecture <- create_architecture(c1, c1, mP, c2, c2, mP, l,
                                    default_dropout = list(linear=0.6, conv=0.4),
                                    default_normalization = list(linear=TRUE),
                                    default_activation = "selu")

# See how the finished CNN would look like for specific input and output shapes
print(architecture, c(3,128,128), 10)

# To use predefined architectures  use the transfer() layer
alexnet <- transfer("alexnet")

# No other linear layers are used after the transfer layer:
# The cnn() function will replace the linear classifier of the
# alexnet architecture with a single linear output layer
architecture <- create_architecture(alexnet)
print(architecture, c(3,128,128), 10)

# Some linear layers are used after the transfer layer:
# The cnn() function will replace the linear classifier of the alexnet architecture
# with the specified linear layers + an output layer that matches the output dimensions
architecture <- create_architecture(alexnet, linear(300), linear(100))
print(architecture, c(3,128,128), 10)
}
```

---

dnn *DNN*

Description

fits a custom deep neural network using the Multilayer Perceptron architecture. `dnn()` supports the formula syntax and allows to customize the neural network to a maximal degree.

Usage

```
dnn(
  formula = NULL,
  data = NULL,
  hidden = c(50L, 50L),
  activation = "selu",
  bias = TRUE,
  dropout = 0,
  custom = NULL,
 loss = c("mse", "mae", "cross-entropy", "bernoulli", "gaussian", "binomial", "poisson",
    "mvp", "nbinom", "multinomial", "clogit", "softmax"),
  custom_parameters = NULL,
 optimizer = c("sgd", "adam", "adadelta", "adagrad", "rmsprop", "rprop", "ignite_adam"),
  lr = 0.01,
  lr_scheduler = NULL,
  alpha = 0.5,
  lambda = 0,
  validation = 0,
  batchsize = NULL,
  shuffle = TRUE,
  epochs = 100,
  weights = NULL,
  early_stopping = Inf,
  burnin = Inf,
  baseloss = NULL,
  device = c("cpu", "cuda", "mps"),
  plot = TRUE,
  verbose = TRUE,
  bootstrap = NULL,
  bootstrap_parallel = FALSE,
  bootstrap_blocking_variable = NULL,
  tuning = config_tuning(),
  hooks = NULL,
  ce = TRUE,
  ce_method = c("analytical", "finite"),
  X = NULL,
  Y = NULL
)
```

Arguments

`formula`	an object of class "formula": a description of the model that should be fitted
`data`	matrix or data.frame with features/predictors and response variable

hidden	hidden units in layers, length of hidden corresponds to number of layers
activation	activation functions, can be of length one, or a vector of different activation functions for each layer
bias	whether use biases in the layers, can be of length one, or a vector (number of hidden layers + 1 (last layer)) of logicals for each layer.
dropout	dropout rate, probability of a node getting left out during training (see nn_dropout)
custom	custom network architecture, replaces the default MLP that is built from hidden, activation, bias, and dropout (these are ignored if custom is provided). Either an nn_module generator, an instantiated nn_module, or a list(module = generator, args = list(...), output_dim = integer). See details
loss	loss after which network should be optimized. Can also be distribution from the stats package or own function, see details
custom_parameters	List of parameters/variables to be optimized. Can be used in a custom loss function. See Vignette for example.
optimizer	which optimizer used for training the network, for more adjustments to optimizer see config_optimizer
lr	learning rate given to optimizer
lr_scheduler	learning rate scheduler created with config_lr_scheduler
alpha	add L1/L2 regularization to training $(1-\alpha)*\|weights\|+\alpha\|\|weights\|\|_2$ will get added for each layer (the L2 part is the Euclidean norm, not its square). Must be between 0 and 1
lambda	strength of regularization: lambda penalty, $\lambda * (L1 + L2)$ (see alpha)
validation	percentage of data set that should be taken as validation set (chosen randomly). Alternatively, a vector containing the indices of the validation samples can be provided.
batchsize	number of samples that are used to calculate one learning rate step, default is 10% of the training data
shuffle	if TRUE, data in each batch gets reshuffled every epoch
epochs	epochs the training goes on for
weights	weights or other values (can be also a matrix) that the likelihood has access to. Only custom loss functions with a weights argument accept them.
early_stopping	if set to an integer, training stops when the loss has increased for the specified number of epochs in a row. The validation loss is used when a validation set is available, otherwise the training loss is used.
burnin	training is aborted if the training loss is not below the baseline loss after burnin epochs
baseloss	baseloss, if null baseloss corresponds to intercept only models
device	device on which network should be trained on. mps correspond to M1/M2 GPU devices.
plot	plot training loss
verbose	print training and validation loss of epochs

`bootstrap` bootstrap neural network or not, numeric corresponds to number of bootstrap samples

`bootstrap_parallel` parallelize (CPU) bootstrapping

`bootstrap_blocking_variable` variable/vector that will be used for blocked bootstrapping (should be a factor)

`tuning` tuning options created with `config_tuning`

`hooks` list of functions that are executed after each epoch (can be used to calculate summary statistics after each epoch)

`ce` whether to calculate the conditional effects after training. TRUE calculates them on at most 5000 observations, and on fewer if the model has many features (the stored effects grow with the number of features squared); a number sets the number of observations directly. FALSE turns them off, in which case `conditionalEffects` and the "ce"/"ale" importances calculate them on demand.

`ce_method` analytical or finite method

`X` Feature matrix or data.frame, alternative data interface

`Y` Response vector, factor, matrix or data.frame, alternative data interface

Value

An S3 object of class `"citodnn"` is returned. It is a list containing everything there is to know about the model and its training process. The list consists of the following attributes:

`net` An object of class "nn_module". Originates from the torch package and represents the core object of this workflow.

`call` The original function call.

`loss` An object of class "nn_module". Contains all relevant information for the loss function, e.g. parameters and a function (format_Y) that transforms target data.

`data` A list. Contains the data used for the training of the model.

`model_properties` A list of properties, that define the architecture of the model.

`training_properties` A list of all training hyperparameters used the last time the model was trained.

`losses` A data.frame containing training and validation losses of each epoch.

`best_epoch_net_state_dict` Serialized state dict of net from the best training epoch.

`best_epoch_loss_state_dict` Serialized state dict of loss from the best training epoch.

`last_epoch_net_state_dict` Serialized state dict of net from the last training epoch.

`last_epoch_loss_state_dict` Serialized state dict of loss from the last training epoch.

`use_model_epoch`
: String, either "best" or "last". Determines whether the parameters (e.g. weights, biases) from the best or the last training epoch are used (e.g. for prediction).

`loaded_model_epoch`
: String, shows from which training epoch the parameters are currently loaded in `net` and `loss`.

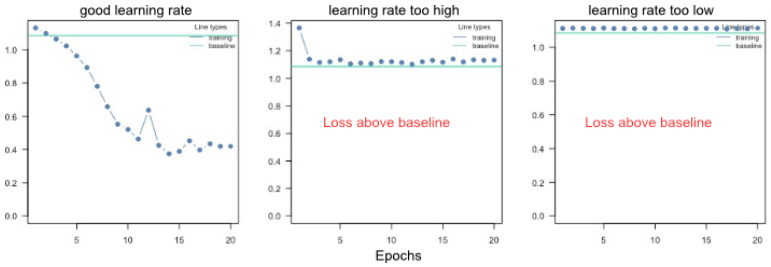
How neural networks work

In Multilayer Perceptron (MLP) networks, each neuron is connected to every neuron in the previous layer and every neuron in the subsequent layer. The value of each neuron is computed using a weighted sum of the outputs from the previous layer, followed by the application of an activation function. Specifically, the value of a neuron is calculated as the weighted sum of the outputs of the neurons in the previous layer, combined with a bias term. This sum is then passed through an activation function, which introduces non-linearity into the network. The calculated value of each neuron becomes the input for the neurons in the next layer, and the process continues until the output layer is reached. The choice of activation function and the specific weight values determine the network's ability to learn and approximate complex relationships between inputs and outputs.

Therefore the value of each neuron can be calculated using: $a(\sum_j w_j * a_j)$. Where w_j is the weight and a_j is the value from neuron j to the current one. a() is the activation function, e.g. $relu(x) = max(0, x)$

Training and convergence of neural networks

Ensuring convergence can be tricky when training neural networks. Their training is sensitive to a combination of the learning rate (how much the weights are updated in each optimization step), the batch size (a random subset of the data is used in each optimization step), and the number of epochs (number of optimization steps). Typically, the learning rate should be decreased with the size of the neural networks (depth of the network and width of the hidden layers). We provide a baseline loss (intercept only model) that can give hints about an appropriate learning rate:



If the training loss of the model doesn't fall below the baseline loss, the learning rate is either too high or too low. If this happens, try higher and lower learning rates.

A common strategy is to try (manually) a few different learning rates to see if the learning rate is on the right scale.

See the troubleshooting vignette (`vignette("B-Training_neural_networks")`) for more help on training and debugging neural networks.

Loss functions / Likelihoods

We support loss functions and likelihoods for different tasks:

Name	Explanation	Example / Task
mse	mean squared error	Regression, predicting continuous values
mae	mean absolute error	Regression, predicting continuous values
cross-entropy	categorical cross entropy	Multi-class, species classification
softmax	categorical cross entropy (deprecated alias for `cross-entropy`)	Multi-class, species classification
gaussian	Normal likelihood	Regression, residual error is also estimated
binomial	Binomial likelihood	Classification/Logistic regression, mortality
bernoulli	Bernoulli likelihood	Classification, binary (0/1) outcome per ob
poisson	Poisson likelihood	Regression, count data, e.g. species abunda
nbinom	Negative binomial likelihood	Regression, count data with dispersion para
mvp	multivariate probit model	joint species distribution model, multi spec
multinomial	Multinomial likelihood	step selection in animal movement models
clogit	conditional binomial	step selection in animal movement models

Additional structures in the formula

Two special terms change how predictors enter the model:

Structure	Syntax	What it does
Embedding, `e`	`Y ~ X1 + X2 + e(Site, dim = 2)`	every level of a factor gets a learned vector of `dim` numbers that ent
Fixed effect, `fe`	`Y ~ X1 + X2 + fe(Temp + I(Temp^2))`	the terms bypass the network and are added linearly on the link sca

Both can be combined, e.g. `dnn(Y ~ . + fe(Temp) + e(Site))`. A variable in `fe()` enters only the linear part.

Why embeddings. One-hot coding (or contrasts) treats every level of a factor as equally distinct: no level is closer to any other. An embedding represents each level by a vector of `dim` coordinates that is estimated together with the network (Guo and Berkhahn, 2016, doi:10.48550/arXiv.1604.06737), so the model can discover which levels behave similarly and place them close together. The coordinates are passed to the first hidden layer together with the other predictors, and because the network learns the interactions itself, they modulate all effects jointly and non-linearly. This is like a joint, non-linear random effect on everything: a random intercept or slope lets a group deviate in one effect, and similarity learned for one predictor does not carry over to another, whereas an embedding maps all differences between the groups into a `dim`-dimensional space. The network learns how similar the groups are in their response and whether they differ at all. Similar to an ordination, groups with similar responses lie close together in the embedding space, so the fitted embeddings can be inspected afterwards (`coef(model)[[1]]$e_1.weight`, one row per level; see `e` for a biplot of the effects).

`dim` controls how much structure the embedding can capture and can be tuned. `e(lambda = , alpha = )` regularizes the embeddings separately from the network. Embeddings are identified only up to rotation, reflection and scale: two fits of the same data give the same configuration in different coordinates, so align them (e.g. Procrustes) before comparing, and inspect them across several refits. Embeddings can also be fixed, e.g. to trait or phylogenetic coordinates, with `e(weights = , train = FALSE)`.

Why fixed effects. A DNN learns every effect flexibly, which makes single effects hard to report. `fe()` adds selected terms as in a GLM, while the network models the rest: β is returned by `coef(model)$fixed_effects` and is estimated like a parametric term in a GAM. The intercept comes from the network. The fixed effects are not regularized by the global `lambda`, only by `fe(lambda = )`. If the fixed-effect variables are correlated with the network inputs, a strongly regularized network can bias β; see `fe` for details.

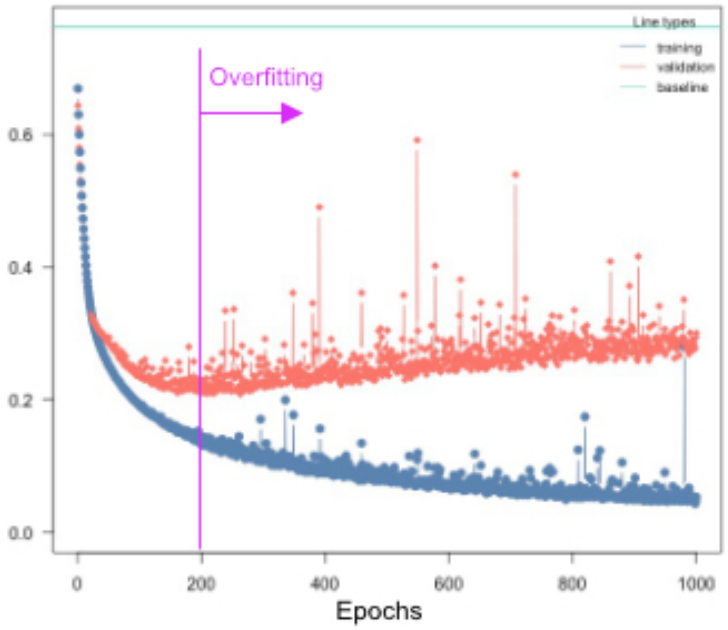
Finding the right architecture

As with the learning rate, there is no definitive guide to choosing the right architecture for the right task. However, there are some general rules/recommendations: In general, wider, and deeper neural networks can improve generalization - but this is a double-edged sword because it also increases the risk of overfitting. So, if you increase the width and depth of the network, you should also add regularization (e.g., by increasing the lambda parameter, which corresponds to the regularization strength). Furthermore, in Pichler & Hartig, 2023, we investigated the effects of the hyperparameters on the prediction performance as a function of the data size. For example, we found that the `selu` activation function outperforms `relu` for small data sizes (<100 observations).

We recommend starting with moderate sizes (like the defaults), and if the model doesn't generalize/converge, try larger networks along with a regularization that helps minimize the risk of overfitting (see `vignette("B-Training_neural_networks")`).

Overfitting

Overfitting means that the model fits the training data well, but generalizes poorly to new observations. We can use the validation argument to detect overfitting. If the validation loss starts to increase again at a certain point, it often means that the models are starting to overfit your training data:



Solutions:

- Re-train with epochs = point where model started to overfit
- Early stopping, stop training when model starts to overfit, can be specified using the `early_stopping=...` argument
- Use regularization (dropout or elastic-net, see next section)

Regularization

Elastic Net regularization combines the strengths of L1 (Lasso) and L2 (Ridge) regularization. It introduces a penalty term that encourages sparse weight values while maintaining overall weight shrinkage. By controlling the sparsity of the learned model, Elastic Net regularization helps avoid overfitting while allowing for meaningful feature selection. We advise using elastic net (e.g. lambda = 0.001 and alpha = 0.2).

Dropout regularization helps prevent overfitting by randomly disabling a portion of neurons during training. This technique encourages the network to learn more robust and generalized representations, as it prevents individual neurons from relying too heavily on specific input patterns. Dropout has been widely adopted as a simple yet effective regularization method in deep learning.

By utilizing these regularization methods in your neural network training with the cito package, you can improve generalization performance and enhance the network's ability to handle unseen data. These techniques act as valuable tools in mitigating overfitting and promoting more robust and reliable model performance.

Uncertainty

We can use bootstrapping to generate uncertainties for all outputs. Bootstrapping can be enabled by setting `bootstrap = ...` to the number of bootstrap samples to be used. Note, however, that the computational cost can be excessive.

In some cases it may be worthwhile to parallelize bootstrapping, for example if you have a GPU and the neural network is small. Parallelization for bootstrapping can be enabled by setting the `bootstrap_parallel = ...` argument to the desired number of calls to run in parallel.

Custom Optimizer and Learning Rate Schedulers

When training a network, you have the flexibility to customize the optimizer settings and learning rate scheduler to optimize the learning process. In the cito package, you can initialize these configurations using the `config_lr_scheduler` and `config_optimizer` functions.

`config_lr_scheduler` allows you to define a specific learning rate scheduler that controls how the learning rate changes over time during training. This is beneficial in scenarios where you want to adaptively adjust the learning rate to improve convergence or avoid getting stuck in local optima.

Similarly, the `config_optimizer` function enables you to specify the optimizer for your network. Different optimizers, such as stochastic gradient descent (SGD), Adam, or RMSprop, offer various strategies for updating the network's weights and biases during training. Choosing the right optimizer can significantly impact the training process and the final performance of your neural network.

Hyperparameter tuning

We have implemented experimental support for hyperparameter tuning. We can mark hyperparameters that should be tuned by cito by setting their values to `tune()`, for example `dnn (..., lr = tune()`. `tune()` is a function that creates a range of random values for the given hyperparameter. You can change the maximum and minimum range of the potential hyperparameters or pass custom values to the `tune(values = c(...))` function. The following table lists the hyperparameters that can currently be tuned:

Hyperparameter	Example	Details
hidden	`dnn(. . . ,hidden=tune(10, 20, fixed='depth'))`	Depth and width can be both tuned or only one o
bias	`dnn(. . . , bias=tune())`	Should the bias be turned on or off for all hidden
lambda	`dnn(. . . , lambda = tune(0.0001, 0.1))`	lambda will be tuned within the range (0.0001, 0.
alpha	`dnn(. . . , alpha = tune(0.2, 0.4))`	alpha will be tuned within the range (0.2, 0.4)
activation	`dnn(. . . , activation = tune())`	activation functions of the hidden layers will be t
dropout	`dnn(. . . , dropout = tune())`	Dropout rate will be tuned (globally for all layers
lr	`dnn(. . . , lr = tune())`	Learning rate will be tuned
batchsize	`dnn(. . . , batchsize = tune())`	batch size will be tuned

The hyperparameters are tuned by random search (i.e., random values for the hyperparameters within a specified range) and by cross-validation. The exact tuning regime can be specified with config_tuning.

Note that hyperparameter tuning can be expensive. We have implemented an option to parallelize hyperparameter tuning, including parallelization over one or more GPUs (the hyperparameter evaluation is parallelized, not the CV). This can be especially useful for small models. For example, if you have 4 GPUs, 20 CPU cores, and 20 steps (random samples from the random search), you could run `dnn(..., device="cuda",lr = tune(), batchsize=tune(), tuning=config_tuning(parallel=20, NGPU=4)`, which will distribute 20 model fits across 4 GPUs, so that each GPU will process 5 models (in parallel).

As this is an experimental feature, we welcome feature requests and bug reports on our github site.

For the custom values, all hyperparameters except for the hidden layers require a vector of values. Hidden layers expect a two-column matrix where the first column is the number of hidden nodes and the second column corresponds to the number of hidden layers.

Custom architectures

Instead of the default MLP, a custom `nn_module` can be passed via the `custom` argument. The module must fulfill the following contract:

- If the initialize method has arguments named `input`, `output`, or `embeddings`, they are filled in automatically by cito (unless provided in `args`): `input` = number of columns of the model.matrix built from the formula (for cnn() the input shape), `output` = the output dimension expected by the loss (within mmn(): `output_dim`), `embeddings` = named vector with the number of levels of each e() term. This way the number of predictors does not have to be hard coded in the custom module.
- forward(x) receives a 2D float tensor (batch, ncol(X), i.e. the columns of the model.matrix built from the formula) and must return a 2D tensor (batch, output), where output matches the dimension expected by the loss (checked at fit time).

- The output must be on the link scale, i.e. the raw output of the last linear layer. Do not apply softmax/sigmoid/exp in the forward method, the loss functions apply the (inverse) link internally (e.g. the cross-entropy loss applies softmax, the poisson loss applies exp) and `predict(..., type = "response")` returns predictions on the response scale via the loss' invlink. Applying the link twice prevents the model from converging properly.
- If embeddings (e()) are used in the formula, forward(x, z) receives a second 2D long tensor with the 1-based factor codes (one column per e() term, in formula order). Embedding layers, including their initialization and regularization, must be handled inside the custom module, e(weights=), e(train=), e(lambda=), and e(alpha=) are therefore not supported.
- The forward method should be differentiable with respect to x if conditionalEffects/summary are to be used.
- lambda/alpha regularization is applied to all parameters of the custom module with more than one dimension.
- If a generator is passed (recommended), the model can be saved/loaded with saveRDS as usual, but the generator's enclosing environment must not capture torch tensors. If an instantiated module is passed, it is serialized via `torch_serialize` internally.
- Architecture hyperparameters (hidden, activation, bias, dropout) are ignored and cannot be tuned; tuning of training hyperparameters (lr, lambda, alpha, batchsize) is still supported.

Activation functions

Supported activation functions: "relu", "leaky_relu", "tanh", "elu", "rrelu", "prelu", "softplus", "celu", "selu", "gelu", "relu6", "sigmoid", "softsign", "hardtanh", "tanhshrink", "softshrink", "hardshrink", "log_sigmoid"

Training on graphic cards

If you have an NVIDIA CUDA-enabled device and a CUDA build of 'torch', you can take advantage of GPU acceleration for training your neural networks (`device = "cuda"`). The supported CUDA versions depend on the installed 'torch' version. For detailed installation instructions and more information on utilizing GPUs for training, please refer to the mlverse: 'torch' documentation.

Note: GPU training is optional, and the package can still be used for training on CPU even without CUDA and cuDNN installations.

Author(s)

Christian Amesoeder, Maximilian Pichler

See Also

`predict.citodnn`, `plot.citodnn`, `coef.citodnn`,`print.citodnn`, `summary.citodnn`, `continue_training`, `analyze_training`, `PDP`, `ALE`,

Examples

```
if(torch::torch_is_installed()){
library(cito)
```

```
# Example workflow in cito

## Build and train  Network
### cross-entropy (softmax link) for multi-class responses (e.g., Species)
nn.fit<- dnn(Species~., data = datasets::iris, loss = "cross-entropy")

## The training loss is below the baseline loss but at the end of the
## training the loss was still decreasing, so continue training for another 50
## epochs
nn.fit <- continue_training(nn.fit, epochs = 50L)

# Structure of Neural Network
print(nn.fit)

# Plot Neural Network
if(requireNamespace("ggraph", quietly = TRUE)) plot(nn.fit, plot_type = "arch")
## 4 Input nodes (first layer) because of 4 features
## 3 Output nodes (last layer) because of 3 response species (one node for each
## level in the response variable).
## The layers between the input and output layer are called hidden layers (two
## of them)

## We now want to understand how the predictions are made, what are the
## important features? The summary function automatically calculates feature
## importance (the interpretation is similar to an anova) and calculates
## average conditional effects that are similar to linear effects:
summary(nn.fit)

## To visualize the effect (response-feature effect), we can use the ALE and
## PDP functions

# Partial dependencies
PDP(nn.fit, variable = "Petal.Length")

# Accumulated local effect plots
ALE(nn.fit, variable = "Petal.Length")



# Per se, it is difficult to get confidence intervals for our xAI metrics (or
# for the predictions). But we can use bootstrapping to obtain uncertainties
# for all cito outputs:
## Re-fit the neural network with bootstrapping
nn.fit<- dnn(Species~.,
             data = datasets::iris,
             loss = "cross-entropy",
             epochs = 150L,
             verbose = FALSE,
             bootstrap = 20L)
## convergence can be tested via the analyze_training function
analyze_training(nn.fit)
```

```
## Summary for xAI metrics (can take some time):
summary(nn.fit, importance = "permutation", type = "link")
## Now with standard errors and p-values
## Note: Take the p-values with a grain of salt! We do not know yet if they are
## correct (e.g. if you use regularization, they are likely conservative == too
## large)

## Predictions with bootstrapping:
dim(predict(nn.fit))
## predictions are by default averaged (over the bootstrap samples)

## Multinomial and conditional logit regression
m = dnn(Species~., data = iris, loss = "clogit", lr = 0.01)
m = dnn(Species~., data = iris, loss = "multinomial", lr = 0.01)

Y = t(stats::rmultinom(100, 10, prob = c(0.2, 0.2, 0.5)))
m = dnn(cbind(X1, X2, X3)~., data = data.frame(Y, A = as.factor(runif(100))),
        loss = "multinomial", lr = 0.01)
## conditional logit for size > 1 is not supported yet


## Binary response (0/1 or a two-level factor)
iris2 = transform(iris, setosa = as.integer(Species == "setosa"))
m = dnn(setosa ~ Sepal.Length + Sepal.Width, data = iris2,
     loss = "bernoulli", optimizer = "ignite_adam", lr = 0.01, epochs = 20L, verbose = FALSE)

## Output of the last hidden layer instead of the predictions
dim(predict(m, return_embeddings = TRUE))

## Blocked bootstrap: resample whole groups instead of rows
m = dnn(Sepal.Length ~ Petal.Length + Petal.Width, data = iris, epochs = 20L,
        bootstrap = 5L, bootstrap_blocking_variable = iris$Species, verbose = FALSE)

## Hooks are evaluated after every epoch inside the training loop, where
## 'model' and 'epoch' are visible; their results are collected in hooks_result
m = dnn(Sepal.Length ~ ., data = iris, epochs = 5L, verbose = FALSE,
        hooks = list(function() model$losses$train_l[epoch]))
unlist(m$hooks_result)

## Observation weights are passed to a custom likelihood with a weights argument
weighted_mse = function(pred, true, weights = NULL) {
  l = (pred - true)^2
  if(is.null(weights)) l else l * weights
}
m = dnn(Sepal.Length ~ ., data = iris, loss = weighted_mse, epochs = 5L,
        weights = runif(150), verbose = FALSE)

# Hyperparameter tuning (experimental feature)
hidden_values = matrix(c(5, 2,
                         4, 2,
                         10,2,
                         15,2), 4, 2, byrow = TRUE)
## Potential architectures we want to test, first column == number of nodes
```

```
print(hidden_values)

nn.fit = dnn(Species~.,
             data = iris,
             epochs = 30L,
             loss = "cross-entropy",
             hidden = tune(values = hidden_values),
             lr = tune(0.00001, 0.1) # tune lr between range 0.00001 and 0.1
             )
## Tuning results:
print(nn.fit$tuning)

# test = Inf means that tuning was cancelled after only one fit (within the CV)


# Advanced: Custom loss functions and additional parameters
## Normal Likelihood with sd parameter:
custom_loss = function(pred, true) {
  logLik = torch::distr_normal(pred,
                               scale = torch::nnf_relu(scale)+
                                 0.001)$log_prob(true)
  return(-logLik$mean())
}

nn.fit<- dnn(Sepal.Length~.,
             data = datasets::iris,
             loss = custom_loss,
             verbose = FALSE,
             custom_parameters = list(scale = 1.0)
)
nn.fit$loss$parameters$scale

## Multivariate normal likelihood with parametrized covariance matrix
## Sigma = L*L^t + D
## Helper function to build covariance matrix
create_cov = function(LU, Diag) {
  return(torch::torch_matmul(LU, LU$t()) + torch::torch_diag(Diag$exp()+0.01))
}

custom_loss_MVN = function(true, pred) {
  Sigma = create_cov(SigmaPar, SigmaDiag)
  logLik = torch::distr_multivariate_normal(pred,
                                            covariance_matrix = Sigma)$
    log_prob(true)
  return(-logLik$mean())
}


nn.fit<- dnn(cbind(Sepal.Length, Sepal.Width, Petal.Length)~.,
             data = datasets::iris,
             lr = 0.01,
             verbose = FALSE,
             loss = custom_loss_MVN,
```

```
              custom_parameters =
                list(SigmaDiag =  rep(0, 3),
                     SigmaPar = matrix(rnorm(6, sd = 0.001), 3, 2))
)
as.matrix(create_cov(nn.fit$loss$parameters$SigmaPar,
                     nn.fit$loss$parameters$SigmaDiag))


# Embeddings and fixed effects
## Species enters as an embedding: each species gets two learned coordinates,
## from which the network learns how the species differ in their response to
## all predictors. Petal.Width enters as a fixed (linear) effect.
nn.fit = dnn(Sepal.Length ~ Sepal.Width + Petal.Length + fe(Petal.Width) + e(Species, dim = 2),
             data = datasets::iris, loss = "mse", epochs = 50L, verbose = FALSE)

## the linear effect of Petal.Width
coef(nn.fit)$fixed_effects

## the embeddings, one row per species
coef(nn.fit)$net_parameters$e_1.weight

## architecture: Species feeds its two embedding dimensions, Petal.Width goes
## straight to the output
if(requireNamespace("ggraph", quietly = TRUE)) plot(nn.fit, plot_type = "arch")
}
```

---

e *Embeddings*

Description

Can be used to create an embedding structure for categorical variables in the function interface

Usage

```
e(dim = 1L, weights = NULL, train = TRUE, lambda = 0, alpha = 1)
```

Arguments

dim	integer, embedding dimension
weights	matrix, to use custom embedding matrices
train	logical, should the embeddings be trained or not
lambda	regularization strength on the embeddings
alpha	mix between L1 and L2 regularization

Details

The e() structure must be used in the function interface. Although not document in the function signature, the first argument to the e() structure is the categorical variable that codes a group in the data, as in

predictors + e(group, ...)

For more details, see the example below

Value

A list with the settings of the embedding; it is only meaningful inside the formula of dnn.

Examples

```
if(torch::torch_is_installed()){

# The following example shows that groups with similar responses will cluster in embedding space
set.seed(123)

n = 10000 # observations
m = 100 # groups / individuals
k = 10 # cluster of groups with the same behavior

dat = data.frame(f1 = runif(n),
                 f2 = runif(n),
                 f3 = runif(n),
                 ind = factor(rep(1:m, each = n/m)),
                 cluster = rep(1:k, each = n/k),
                 response = NA)

slopes = matrix(runif(3*k, min = -10,max = 10), nrow = k, ncol = 3)

for(i in 1:k) dat$response[dat$cluster == i] =
  as.matrix(dat[dat$cluster == i, 1:3]) %*% slopes[i,] + rnorm(n/k, sd = 0.2)

mod <- dnn(response~f1+f2+f3 + e(ind,dim = 2),
           data = dat, epochs = 200L, optimizer = config_optimizer("adam"))

embeddings = coef(mod)[[1]][[1]] # extract embeddings
plot(embeddings, col = c(rep(1:k, each = m/k))) # plot clusters in embedding space
abline(h = 0, lty = 2)
abline(v = 0, lty = 2)

ace = conditionalEffects(mod) # extract conditional effects
# now average conditional effects per cluster
ind_ace =
  sapply(1:m, function(ind) {
    tmp = ace[[1]]$result[dat$ind==ind,,]
    return(diag(apply(tmp, 2:3, mean)))
  })

# to create biplot, multiply beta of each cluster with coordinates
```

```
coord = ind_ace %*% embeddings/m
arrows(x0 = rep(0, 3), x1 = coord[,1], y0 = rep(0,3), y1 =coord[,2])

}
```

`fe` *Fixed effects*

Description

Adds predictors with linear effects to the DNN in the formula interface: `dnn(Y ~ X1 + fe(Temp + I(Temp^2)))`.

Usage

```
fe(lambda = 0, alpha = 1)
```

Arguments

`lambda`	regularization strength on the fixed effects (not tunable with `tune()`)
`alpha`	mix between L1 (`alpha = 0`) and L2 (`alpha = 1`) regularization

Details

The first argument of `fe()`, not part of its signature, is the right-hand side of a formula, so `I()`, interactions and factors work as in `lm()` (factors with treatment contrasts; `0 +` or `- 1` inside `fe()` are ignored). Its terms enter the linear predictor (the link scale) as $\eta = f(X) + \beta^T x_{fe}$, without an intercept, which the output layer of the DNN f provides (so do not combine `fe()` with `bias = FALSE`). The variables in `fe()` are not passed to the DNN, and `.` in the formula expands to all other variables; a variable in `fe()` and in the DNN or in `e()` is an error. Use one `fe()` term. Transformations that depend on the data, such as `poly()` or `scale()`, are re-evaluated on `newdata` in `predict()`; compute them beforehand.

The global `lambda` and `alpha` of `dnn()` do not apply to the fixed effects; they are regularized only by the `lambda` and `alpha` given to `fe()` (default: not at all), on the scale of the model matrix, so center and scale the fixed-effect variables when regularizing them.

β is returned by `coef()` as `fixed_effects`, one row per model-matrix column and one column per output of the network. It is estimated like a parametric term in a GAM, and is close to the GLM coefficient only if the training converged and the fixed-effect variables are not functions of the DNN inputs. If they are correlated with the DNN inputs, regularization of the DNN (`lambda`, dropout, early stopping) can bias β. For losses with a softmax (`"cross-entropy"`, `"multinomial"`, `"clogit"`) only the differences between the columns of `fixed_effects` are identified. `coef()` gives no standard errors; fit with `bootstrap =` for their spread. In `summary()` and `conditionalEffects()`, the effect of a fixed effect is the effect of its model-matrix column, e.g. separate rows for `Temp` and `I(Temp^2)`.

`fe()` can be combined with `e()`, but is not supported in `mmn()` or with `custom`.

Value

A list with `lambda` and `alpha`; it is only meaningful inside the formula of `dnn`.

Examples

```
if(torch::torch_is_installed()){
library(cito)

# linear effects of Temp and Temp^2, a non-linear effect of X1 learned by the DNN
set.seed(42)
n = 1000
dat = data.frame(X1 = runif(n, -2, 2), Temp = rnorm(n), Precp = rnorm(n))
dat$Y = sin(2 * dat$X1) + 1.0 * dat$Temp - 0.5 * dat$Temp^2 + rnorm(n, sd = 0.3)

m = dnn(Y ~ X1 + fe(Temp + I(Temp^2)), data = dat, loss = "mse", epochs = 50L, verbose = FALSE)

# the fixed effects, close to 1.0 and -0.5
coef(m)$fixed_effects

# "." uses every variable that is not in fe(); here X1 and Precp
m2 = dnn(Y ~ . + fe(Temp + I(Temp^2), lambda = 0.001), data = dat, loss = "mse",
         epochs = 50L, verbose = FALSE)
}
```

---

findReTrmClasses *list of specials – taken from enum.R*

Description

list of specials – taken from enum.R

Usage

```
findReTrmClasses()
```

---

linear *Create a Linear Layer for a CNN Architecture*

Description

This function creates a `linear` layer object of class `citolayer` for use in constructing a Convolutional Neural Network (CNN) architecture. The resulting layer object can be passed to the `create_architecture` function to define the structure of the network.

Usage

```
linear(
  n_neurons = NULL,
  bias = NULL,
  activation = NULL,
  normalization = NULL,
  dropout = NULL
)
```

Arguments

n_neurons	(integer) The number of hidden neurons in this layer.
bias	(boolean) If TRUE, a learnable bias is added to the neurons of this layer.
activation	(character) The activation function applied after this layer. Supported activation functions include "relu", "leaky_relu", "tanh", "elu", "rrelu", "prelu", "softplus", "celu", "selu", "gelu", "relu6", "sigmoid", "softsign", "hardtanh", "tanhshrink", "softshrink", "hardshrink", "log_sigmoid".
normalization	(boolean) If TRUE, batch normalization is applied after this layer.
dropout	(numeric) The dropout rate for this layer. Set to 0 to disable dropout.

Details

This function creates a linear layer object, which is used to define a linear layer in a CNN architecture. Parameters not specified (and thus set to NULL) will be filled with default values provided to the create_architecture function.

Value

An S3 object of class "linear" "citolayer", representing a linear layer in the CNN architecture.

Author(s)

Armin Schenk

See Also

create_architecture

Examples

```
if(torch::torch_is_installed()){
library(cito)

# A linear layer where all available parameters are assigned
# No value will be overwritten by 'create_architecture()'
layer1 <- linear(100, TRUE, "relu", FALSE, 0.5)

# A linear layer where only the activation function is assigned
# n_neurons, bias, normalization and dropout are filled with the defaults
```

```
# passed to the 'create_architecture()' function
layer2 <- linear(activation="selu")
}
```

---

maxPool *Create a Maximum Pooling Layer for a CNN Architecture*

Description

This function creates a `maxPool` layer object of class `citolayer` for use in constructing a Convolutional Neural Network (CNN) architecture. The resulting layer object can be passed to the `create_architecture` function to define the structure of the network.

Usage

```
maxPool(kernel_size = NULL, stride = NULL, padding = NULL, dilation = NULL)
```

Arguments

`kernel_size`	(integer or tuple) The size of the kernel in this layer. Use a tuple if the kernel size varies across dimensions.
`stride`	(integer or tuple) The stride of the kernel in this layer. If `NULL`, the stride is set to the kernel size. Use a tuple if the stride differs across dimensions.
`padding`	(integer or tuple) The amount of zero-padding added to the input on both sides. Use a tuple if the padding differs across dimensions.
`dilation`	(integer or tuple) The dilation of the kernel in this layer. Use a tuple if the dilation varies across dimensions.

Details

This function creates a `maxPool` layer object, which represents a maximum pooling layer in a CNN architecture. Parameters not specified (and thus set to `NULL`) will be filled with default values provided to the `create_architecture` function.

Value

An S3 object of class `"maxPool" "citolayer"`, representing a maximum pooling layer in the CNN architecture.

Author(s)

Armin Schenk

See Also

`create_architecture`

Examples

```
if(torch::torch_is_installed()){
library(cito)

# A maximum pooling layer where all available parameters are assigned
# No value will be overwritten by 'create_architecture()'
layer1 <- maxPool(3, 1, 0, 1)

# A maximum pooling layer where only the kernel size is assigned
# stride, padding and dilation are filled with the defaults
# passed to the 'create_architecture()' function
layer2 <- maxPool(kernel_size=4)
}
```

---

mmn *Train a Multi-Modal Neural Network (MMN)*

Description

This function trains a Multi-Modal Neural Network (MMN) which consists of a combination of DNNs and CNNs.

Usage

```
mmn(
  formula,
  dataList = NULL,
  fusion_hidden = c(50L, 50L),
  fusion_activation = "relu",
  fusion_bias = TRUE,
  fusion_dropout = 0,
 loss = c("mse", "mae", "cross-entropy", "bernoulli", "gaussian", "binomial", "poisson",
    "mvp", "nbinom", "multinomial", "clogit", "softmax"),
  custom_parameters = NULL,
 optimizer = c("sgd", "adam", "adadelta", "adagrad", "rmsprop", "rprop", "ignite_adam"),
  lr = 0.01,
  lr_scheduler = NULL,
  alpha = 0.5,
  lambda = 0,
  validation = 0,
  batchsize = NULL,
  shuffle = TRUE,
  data_augmentation = NULL,
  epochs = 100,
  weights = NULL,
  early_stopping = Inf,
  burnin = Inf,
```

```
  baseloss = NULL,
  device = c("cpu", "cuda", "mps"),
  plot = TRUE,
  verbose = TRUE
)
```

Arguments

formula	A formula object specifying the model structure. See examples for more information.
dataList	A list containing the data for training the model. The list should contain all variables used in the formula.
fusion_hidden	A numeric vector specifying the number of units (nodes) in each hidden layer of the fusion network. The length of this vector determines the number of hidden layers created, with each element specifying the number of units in the corresponding layer.
fusion_activation	A character vector specifying the activation function(s) applied after each hidden layer in the fusion network. If a single character string is provided, the same activation function will be applied to all hidden layers. Alternatively, a character vector of the same length as `fusion_hidden` can be provided to apply different activation functions to each layer. Available options include: `"relu"`, `"leaky_relu"`, `"tanh"`, `"elu"`, `"rrelu"`, `"prelu"`, `"softplus"`, `"celu"`, `"selu"`, `"gelu"`, `"relu6"`, `"sigmoid"`, `"softsign"`, `"hardtanh"`, `"tanhshrink"`, `"softshrink"`, `"hardshrink"`, `"log_sigmoid"`.
fusion_bias	A logical value or a vector indicating whether to include bias terms in each layer of the fusion network. If a single logical value is provided, it will apply to all layers. To specify bias inclusion for each layer individually, provide a logical vector of length `length(fusion_hidden) + 1`, where each element corresponds to a hidden layer, and the final element controls whether a bias term is added to the output layer.
fusion_dropout	The dropout rate(s) to apply to each hidden layer in the fusion network. This can be a single numeric value (between 0 and 1) to apply the same dropout rate to all hidden layers, or a numeric vector of length `length(fusion_hidden)` to set different dropout rates for each layer individually. The dropout rate is not applied to the output layer.
loss	The loss function to be used. Options include "mse", "mae", "cross-entropy", "bernoulli", "gaussian", "binomial", "poisson", "nbinom", "mvp", "multinomial", and "clogit". You can also specify your own loss function. See Details for more information. Default is "mse".
custom_parameters	Parameters for the custom loss function. See the vignette for an example. Default is NULL.
optimizer	The optimizer to be used. Options include "sgd", "adam", "adadelta", "adagrad", "rmsprop", "rprop", and "ignite_adam". See `config_optimizer` for further adjustments to the optimizer. Default is "sgd".
lr	Learning rate for the optimizer. Default is 0.01.

`lr_scheduler` Learning rate scheduler. See `config_lr_scheduler` for creating a learning rate scheduler. Default is NULL.

`alpha` Alpha value for L1/L2 regularization. Default is 0.5.

`lambda` Lambda value for L1/L2 regularization. Default is 0.0.

`validation` Proportion of the data to be used for validation. Alternatively, a vector containing the indices of the validation samples can be provided. Default is 0.0.

`batchsize` Batch size for training. If NULL, batchsize is 10% of the training data. Default is NULL.

`shuffle` Whether to shuffle the data before each epoch. Default is TRUE.

`data_augmentation` A list of functions used for data augmentation. Elements must be either functions or strings corresponding to inbuilt data augmentation functions. See details for more information.

`epochs` Number of epochs to train the model. Default is 100.

`weights` weights or other values (can be also a matrix) that the likelihood has access to.

`early_stopping` Number of epochs with no improvement after which training will be stopped. Default is Inf.

`burnin` Number of epochs after which the training stops if the loss is still above the baseloss. Default is Inf.

`baseloss` Baseloss used for burnin and plot. If NULL, the baseloss corresponds to intercept only models. Default is NULL.

`device` Device to be used for training. Options are "cpu", "cuda", and "mps". Default is "cpu".

`plot` Whether to plot the training progress. Default is TRUE.

`verbose` Whether to print detailed training progress. Default is TRUE.

Value

An S3 object of class `"citommn"` is returned. It is a list containing everything there is to know about the model and its training process. The list consists of the following attributes:

`net` An object of class "nn_module". Originates from the torch package and represents the core object of this workflow.

`call` The original function call.

`loss` An object of class "nn_module". Contains all relevant information for the loss function, e.g. parameters and a function (format_Y) that transforms target data.

`data` A list. Contains the data used for the training of the model.

`model_properties` A list of properties, that define the architecture of the model.

`training_properties` A list of all training hyperparameters used the last time the model was trained.

`losses` A data.frame containing training and validation losses of each epoch.

`best_epoch_net_state_dict`
: Serialized state dict of net from the best training epoch.

`best_epoch_loss_state_dict`
: Serialized state dict of loss from the best training epoch.

`last_epoch_net_state_dict`
: Serialized state dict of net from the last training epoch.

`last_epoch_loss_state_dict`
: Serialized state dict of loss from the last training epoch.

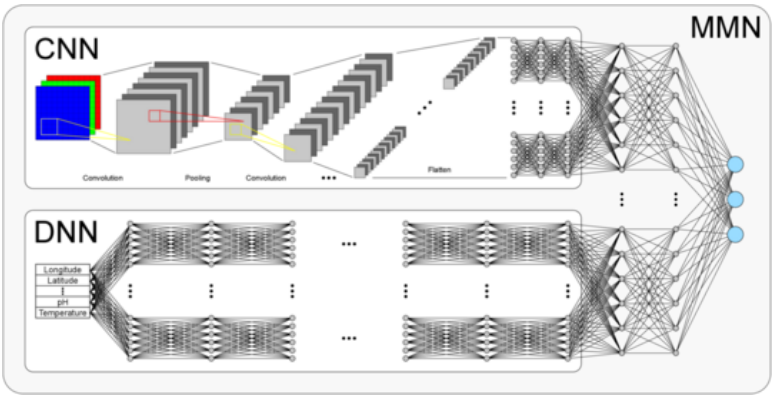
`use_model_epoch`
: String, either "best" or "last". Determines whether the parameters (e.g. weights, biases) from the best or the last training epoch are used (e.g. for prediction).

`loaded_model_epoch`
: String, shows from which training epoch the parameters are currently loaded in `net` and `loss`.

Details:

Also check dnn and cnn for details to common arguments.

MMN architecture:



The MMN combines multiple CNNs and DNNs. This allows the model to process data in different formats (e.g., DNN+CNN for tabular data and images, or CNN+CNN for images with different spatial resolutions). The architecture of the MMN is defined by the arguments `formula`, `fusion_hidden`, `fusion_activation`, `fusion_bias` and `fusion_dropout`:

- `formula` specifies the architecture of the individual networks as well as their respective inputs, and the target data of the MMN (which specifies the shape of the output layer).
- `fusion_hidden`, `fusion_activation`, `fusion_bias` and `fusion_dropout` define the architecture of the DNN that fuses the outputs of the individual networks. See dnn for details.

`mmn(Y ~ dnn(X=tabular_data1) + dnn(~., data=tabular_data2) + cnn(X=image_data), dataList=mmn_data, ...)`

In this example, **Y** (left side of ~) is the target data of the MMN. On the right side of ~ you can specify as many DNNs and CNNs as required. The specification works exactly as in dnn and cnn with the following restrictions:

- Only specify arguments that relate to the architecture and input data of the network (bold arguments mandatory):
  - dnn(): **formula**, **data**, hidden, activation, bias, dropout, (**X**, alternatively to formula and data)
  - cnn(): **X**, **architecture**
- Arguments relating to the training (e.g. loss, lr, epochs, ...) have to be passed to mmn() instead.
- The names of the data variables (in this example: Y, tabular_data1, tabular_data2, image_data) must be available in `dataList` (named list).

Author(s)

Armin Schenk

See Also

`predict.citommn`, `print.citommn`, `summary.citommn`, `coef.citommn`, `continue_training`, `analyze_training`

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Example workflow in cito

device <- ifelse(torch::cuda_is_available(), "cuda", "cpu")

## Simulated data
shapes <- cito:::simulate_shapes(n=320, size=50, channels=3)
X_cnn <- shapes$data
X_dnn <- matrix(runif(320*3),320,3)
Y <- (as.integer(shapes$labels)-1)*2 + 0.5*X_dnn[,1] + 0.3*X_dnn[,2] - 0.8*X_dnn[,3]

data <- list(Y=Y, X_cnn=X_cnn, X_dnn=X_dnn)

## Architecture of the CNN
architecture <- create_architecture(conv(5), maxPool(), conv(5), maxPool(), linear(10))

## Build and train network
mmn.fit <- mmn(Y ~
                 dnn(~., data=X_dnn, hidden = c(100,100,100), activation = "relu") +
                 cnn(X=X_cnn, architecture = architecture),
               dataList = data, loss = "mse",
               epochs = 1, validation = 0.1, lr = 0.05, device=device)

## If the loss is still decreasing you can continue training for additional epochs:
mmn.fit <- continue_training(mmn.fit, epochs = 1)

## Predictions for new data: a list with the same names as in the formula
new_data <- list(X_cnn = X_cnn[1:5,,,,drop = FALSE], X_dnn = X_dnn[1:5,])
predict(mmn.fit, newdata = new_data)
```

```
## Output of the last hidden layer of the fusion network, e.g. for a downstream model
dim(predict(mmn.fit, newdata = new_data, return_embeddings = TRUE))

## Importance of the data streams: each stream is permuted across the samples
## and the increase in loss is what it contributes
summary(mmn.fit, n_permute = 3L)
plot(mmn.fit, n_permute = 3L)

## Architecture: one row per data stream, the concatenation, and the fusion network
plot(mmn.fit, plot_type = "arch")

## Weights and biases of all submodels
names(coef(mmn.fit)$net_parameters)

## Images can also stay on disk, see ?cnn (folder input) and ?raster_patches
## (windows cut from a large GeoTIFF); both work as data streams of mmn()
}
```

PDP *Partial Dependence Plot (PDP)*

Description

Calculates the Partial Dependency Plot for one feature, either numeric or categorical. Returns it as a plot.

Usage

```
PDP(
  model,
  variable = NULL,
  data = NULL,
  ice = FALSE,
  resolution.ice = 20,
  plot = TRUE,
  parallel = FALSE,
  ...
)

## S3 method for class 'citodnn'
PDP(
  model,
  variable = NULL,
  data = NULL,
  ice = FALSE,
  resolution.ice = 20,
  plot = TRUE,
```

```
  parallel = FALSE,
  ...
)

## S3 method for class 'citodnnBootstrap'
PDP(
  model,
  variable = NULL,
  data = NULL,
  ice = FALSE,
  resolution.ice = 20,
  plot = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

`model`	a model created by `dnn`
`variable`	variable (as a string) for which the PDP should be computed. If none is supplied, it is computed for all variables.
`data`	new data on which the PDP should be computed. If NULL, the PDP is computed on the training data.
`ice`	if TRUE, the Individual Conditional Expectation (ICE) curves are also shown
`resolution.ice`	resolution (number of grid points) at which the ICE curves are computed
`plot`	whether to plot the PDP
`parallel`	whether to parallelize over the bootstrap models
`...`	arguments passed to `predict`

Value

A list of plots made with 'ggplot2', one for each selected variable.

Description

Performs a partial dependence plot (PDP) estimation to analyze the relationship between a selected feature and the model's predictions.

The PDP function estimates the partial function $\hat{f}_S$:

$\hat{f}_S(x_S) = \frac{1}{n} \sum_{i=1}^{n} \hat{f}(x_S, x_C^{(i)})$

using a Monte Carlo estimation, i.e. it computes the average prediction over the data while the selected feature is held fixed at a given value and the remaining features are kept at their observed values.

For categorical features, every observation is set to each level of the feature in turn, the average prediction per level is calculated, and the result is shown as a bar plot.

If `ice = TRUE`, the Individual Conditional Expectation (ICE) curves are also shown, with the PDP highlighted in yellow. Each ICE curve illustrates how the prediction for a single observation changes

as the feature varies. ICE curves are computed on a value grid rather than at every observed feature value, and are not available for categorical features.

See Also

ALE

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris)

PDP(nn.fit, variable = "Petal.Length")
}
```

---

plot.citoarchitecture *Plot method for citoarchitecture objects*

Description

This method provides a visual representation of the network architecture defined by an object of class `citoarchitecture`, including information about each layer's configuration. It helps in understanding the structure of the architecture defined by `create_architecture`.

Usage

```
## S3 method for class 'citoarchitecture'
plot(x, input_shape, output_shape = NULL, ...)
```

Arguments

`x`	An object of class `citoarchitecture`, created by `create_architecture`.
`input_shape`	A numeric vector specifying the dimensions of a single sample (e.g., `c(3, 28, 28)` for an RGB image with height and width of 28 pixels). This argument is required for a detailed output.
`output_shape`	An integer specifying the number of nodes in the output layer. If `NULL`, no output layer is printed.
`...`	Additional arguments (currently not used).

Value

The original `citoarchitecture` object, returned invisibly.

Examples

```
if(torch::torch_is_installed()){
library(cito)

c1 <- conv(n_kernels = 8, kernel_size = 5)
c2 <- conv(n_kernels = 16, kernel_size = 3)
l <- linear(n_neurons = 100)
mP <- maxPool(kernel_size = 2)
architecture <- create_architecture(c1, c1, mP, c2, c2, mP, l,
                                    default_dropout = list(linear=0.6, conv=0.4),
                                    default_normalization = list(linear=TRUE),
                                    default_activation = "selu")

# See how the finished CNN would look like for specific input and output shapes
plot(architecture, c(3,128,128), 10)
}
```

`plot.citocnn` *Plot a fitted CNN model*

Description

This function plots the architecture of a Convolutional Neural Network (CNN) model created using the cnn function.

Usage

```
## S3 method for class 'citocnn'
plot(x, ...)
```

Arguments

`x`	A model created by cnn.
`...`	Additional arguments (currently not used).

Value

The original model object `x`, returned invisibly.

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)

device <- ifelse(torch::cuda_is_available(), "cuda", "cpu")
```

```
## Data
shapes <- cito:::simulate_shapes(320, 28)
X <- shapes$data
Y <- shapes$labels

## Architecture
architecture <- create_architecture(conv(5), maxPool(), conv(5), maxPool(), linear(10))

## Build and train network
cnn.fit <- cnn(X, Y, architecture, loss = "cross-entropy", epochs = 50,
               validation = 0.1, lr = 0.05, device = device)

## Structure of Neural Network
plot(cnn.fit)
}
```

`plot.citodnn` *Plot the effects or the architecture of a DNN*

Description

`plot_type = "ALE"` (default) plots the accumulated local effects of the features (see ALE); `plot_type = "arch"` draws the network architecture as a graph, including embeddings and fixed effects, and needs the packages 'ggraph' and 'igraph'.

Usage

```
## S3 method for class 'citodnn'
plot(x, plot_type = c("ALE", "arch"), node_size = 1, scale_edges = FALSE, ...)

## S3 method for class 'citodnnBootstrap'
plot(
  x,
  plot_type = c("ALE", "arch"),
  node_size = 1,
  scale_edges = FALSE,
  which_model = 1,
  ...
)
```

Arguments

`x`	a model created by dnn
`plot_type`	plot ALE or architecture
`node_size`	size of node in plot
`scale_edges`	edge weight gets scaled according to other weights (layer specific)
`...`	no further functionality implemented yet
`which_model`	which model from the ensemble should be plotted

Value

For "ALE", the plots of ALE; for "arch", a 'ggraph' plot of the network.

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)
validation_set<- sample(c(1:nrow(datasets::iris)),25)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris[-validation_set,])

# accumulated local effects
plot(nn.fit)

# architecture
if(requireNamespace("ggraph", quietly = TRUE)) plot(nn.fit, plot_type = "arch")
}
```

`plot.citommn` *Plot a fitted MMN model*

Description

Visualizes either the permutation importance of the data streams or the architecture of a Multi-Modal Neural Network.

Usage

```
## S3 method for class 'citommn'
plot(
  x,
  plot_type = c("importance", "arch"),
  n_permute = 10L,
  device = NULL,
  ...
)
```

Arguments

`x`	a model created by `mmn`
`plot_type`	`"importance"` draws the permutation importance of each data stream, `"arch"` the architecture: one row per data stream with its submodel, the concatenation, the fusion network and the output. `plot()` on a single `create_architecture` object still draws that submodel's layers in full.

n_permute	number of permutations per data stream, only used for "importance"
device	device on which to evaluate, defaults to the device the model was trained on
...	passed to barplot, only used for "importance"

Value

invisibly, the importance table for "importance" and NULL otherwise

---

predict.citocnn *Predict with a fitted CNN model*

Description

This function generates predictions from a Convolutional Neural Network (CNN) model that was created using the cnn function.

Usage

```
## S3 method for class 'citocnn'
predict(
  object,
  newdata = NULL,
  type = c("link", "response", "class"),
  device = NULL,
  batchsize = NULL,
  return_embeddings = FALSE,
  ...
)
```

Arguments

object	a model created by cnn.
newdata	A multidimensional array representing the new data for which predictions are to be made. The dimensions of newdata should match those of the training data, except for the first dimension which represents the number of samples. As an alternative, you can provide the relative or absolute path to the folder containing the images. In this case, unsigned integer images are divided by their maximum value (255 for 8 bit, 65535 for 16 bit), and floating point and signed integer TIFFs are passed unchanged. Windows of a raster are given with raster_patches (use its like argument, so that they carry the preprocessing of the training windows); they are not rescaled. If NULL, the function uses the data the model was trained on.
type	A character string specifying the type of prediction to be made. Options are: • "link": Scale of the linear predictor. • "response": Scale of the response.

- `"class"`: The predicted class labels (for classification tasks).

`device` Device to be used for making predictions. Options are "cpu", "cuda", and "mps". If `NULL`, the function uses the same device that was used when training the model. Default is `NULL`.

`batchsize` An integer specifying the number of samples to be processed at the same time. If `NULL`, the function uses the same batchsize that was used when training the model. Default is `NULL`.

`return_embeddings`
Return embeddings instead of predictions

`...` Additional arguments (currently not used).

Value

A matrix of predictions. If `type` is `"class"`, a factor of predicted class labels is returned.

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)

device <- ifelse(torch::cuda_is_available(), "cuda", "cpu")

## Data
shapes <- cito:::simulate_shapes(320, 28)
X <- shapes$data
Y <- shapes$labels

## Architecture
architecture <- create_architecture(conv(5), maxPool(), conv(5), maxPool(), linear(10))

## Build and train network
cnn.fit <- cnn(X, Y, architecture, loss = "cross-entropy", epochs = 50,
               validation = 0.1, lr = 0.05, device = device)

## Get predictions of the validation set
valid <- cnn.fit$data$validation
predictions <- predict(cnn.fit, newdata = X[valid,,,,drop=FALSE], type="class")

## Classification accuracy
accuracy <- sum(predictions == Y[valid])/length(valid)

}
```

`predict.citodnn` *Predict from a fitted dnn model*

Description

Predict from a fitted dnn model

Usage

```
## S3 method for class 'citodnn'
predict(
  object,
  newdata = NULL,
  type = c("link", "response", "class"),
  device = NULL,
  batchsize = NULL,
  return_embeddings = FALSE,
  ...
)

## S3 method for class 'citodnnBootstrap'
predict(
  object,
  newdata = NULL,
  type = c("link", "response", "class"),
  device = NULL,
  batchsize = NULL,
  reduce = c("mean", "median", "none"),
  ...
)
```

Arguments

`object`	a model created by dnn
`newdata`	new data for predictions
`type`	type of predictions. The default is on the scale of the linear predictor, "response" is on the scale of the response, and "class" means that class predictions are returned (if it is a classification task)
`device`	device on which network should be trained on.
`batchsize`	number of samples that are predicted at the same time
`return_embeddings`	Return embeddings instead of predictions
`...`	additional arguments
`reduce`	predictions from bootstrapped model are by default reduced (mean, optional median or none)

Value

prediction matrix

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)
validation_set<- sample(c(1:nrow(datasets::iris)),25)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris[-validation_set,])

# Use model on validation set
predictions <- predict(nn.fit, iris[validation_set,])
# Scatterplot
plot(iris[validation_set,]$Sepal.Length,predictions)
# MAE
mean(abs(predictions-iris[validation_set,]$Sepal.Length))
}
```

---

predict.citommn *Predict with a fitted MMN model*

Description

This function generates predictions from a Multi-Modal Neural Network (MMN) model that was created using the mmn function.

Usage

```
## S3 method for class 'citommn'
predict(
  object,
  newdata = NULL,
  type = c("link", "response", "class"),
  device = NULL,
  batchsize = NULL,
  return_embeddings = FALSE,
  ...
)
```

Arguments

object	a model created by mmn

newdata	A list containing the new data for which predictions are to be made. The dimensions of the elements in newdata should match those of the training data, except for the respective first dimensions which represents the number of samples. If NULL, the function uses the data the model was trained on.
type	A character string specifying the type of prediction to be made. Options are: • "link": Scale of the linear predictor. • "response": Scale of the response. • "class": The predicted class labels (for classification tasks).
device	Device to be used for making predictions. Options are "cpu", "cuda", and "mps". If NULL, the function uses the same device that was used when training the model. Default is NULL.
batchsize	An integer specifying the number of samples to be processed at the same time. If NULL, the function uses the same batchsize that was used when training the model. Default is NULL.
return_embeddings	return the embeddings (output of the last hidden layer of the fusion network) instead of the predictions. Default is FALSE.
...	Additional arguments (currently not used).

Value

A matrix of predictions. If type is "class", a factor of predicted class labels is returned.

---

print.citoarchitecture

Print method for citoarchitecture objects

Description

This method provides a visual representation of the network architecture defined by an object of class citoarchitecture, including information about each layer's configuration. It helps in understanding the structure of the architecture defined by create_architecture.

Usage

```
## S3 method for class 'citoarchitecture'
print(x, input_shape, output_shape = NULL, ...)
```

Arguments

x	An object of class citoarchitecture, created by create_architecture.
input_shape	A numeric vector specifying the dimensions of a single sample (e.g., c(3, 28, 28) for an RGB image with height and width of 28 pixels). This argument is required for a detailed output.
output_shape	An integer specifying the number of nodes in the output layer. If NULL, no output layer is printed.
...	Additional arguments (currently not used).

Value

The original `citoarchitecture` object, returned invisibly.

Examples

```
if(torch::torch_is_installed()){
library(cito)

c1 <- conv(n_kernels = 8, kernel_size = 5)
c2 <- conv(n_kernels = 16, kernel_size = 3)
l <- linear(n_neurons = 100)
mP <- maxPool(kernel_size = 2)
architecture <- create_architecture(c1, c1, mP, c2, c2, mP, l,
                                    default_dropout = list(linear=0.6, conv=0.4),
                                    default_normalization = list(linear=TRUE),
                                    default_activation = "selu")

# See how the finished CNN would look like for specific input and output shapes
print(architecture, c(3,128,128), 10)
}
```

---

print.citocnn *Print a fitted CNN model*

Description

This function prints the architecture of a Convolutional Neural Network (CNN) model created using the cnn function.

Usage

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

Arguments

x	A model created by cnn.
...	Additional arguments (currently not used).

Value

The original model object x, returned invisibly.

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)

device <- ifelse(torch::cuda_is_available(), "cuda", "cpu")

## Data
shapes <- cito:::simulate_shapes(320, 28)
X <- shapes$data
Y <- shapes$labels

## Architecture
architecture <- create_architecture(conv(5), maxPool(), conv(5), maxPool(), linear(10))

## Build and train network
cnn.fit <- cnn(X, Y, architecture, loss = "cross-entropy",
               epochs = 50, validation = 0.1, lr = 0.05, device=device)

# Structure of Neural Network
print(cnn.fit)
}
```

`print.citodnn` *Print class citodnn*

Description

Print class citodnn

Usage

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

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

Arguments

`x`	a model created by dnn
`...`	additional arguments

Value

original object x gets returned

Examples

```
if(torch::torch_is_installed()){
library(cito)

set.seed(222)
validation_set<- sample(c(1:nrow(datasets::iris)),25)

# Build and train  Network
nn.fit<- dnn(Sepal.Length~., data = datasets::iris[-validation_set,])

# Structure of Neural Network
print(nn.fit)
}
```

`print.citommn` *Print a fitted MMN model*

Description

This function prints the architecture of a Multi-Modal Neural Network (MMN) model created using the `mmn` function.

Usage

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

Arguments

`x`	A model created by `mmn`.
`...`	Additional arguments (currently not used).

Value

The original model object `x`, returned invisibly.

print.conditionalEffects *Print average conditional effects*

Description

Print average conditional effects

Usage

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

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

Arguments

x	print ACE calculated by conditionalEffects
...	optional arguments for compatibility with the generic function, no function implemented

Value

Matrix with average conditional effects

print.summary.citodnn *Print method for class summary.citodnn*

Description

Print method for class summary.citodnn

Usage

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

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

Arguments

x	a summary object created by summary.citodnn
...	additional arguments

Value

List with Matrices for importance, average CE, absolute sum of CE, and standard deviation of the CE

`raster_patches` *Image patches cut from one large raster*

Description

Describes, for every observation, a square window of a large GeoTIFF centered on the observation's coordinates. The result is passed as `X` to `cnn` (or to a `cnn()` term of `mmn`) in place of an array of images or a folder. The windows are never extracted to disk: each batch is read from the raster when it is needed, and with libtiff 4.5 or newer the next batch is read in the background while the current one trains.

Usage

```
raster_patches(
  raster,
  coords,
  extent,
  center = NULL,
  scale = NULL,
  standardize = FALSE,
  transform = NULL,
  fill = 0,
  na_channel = FALSE,
  like = NULL
)

## S3 method for class 'citoraster'
as.array(x, ...)
```

Arguments

`raster`	Path to a GeoTIFF. Any number of bands; 8, 16 and 32 bit integer and 32 and 64 bit float samples. Values are read as stored and converted to 32 bit float: integers above 2^24 are rounded, and 64 bit values beyond the float range become missing.
`coords`	Matrix or data.frame with two columns, x and y, giving the center of each window in the coordinate reference system of `raster` (which is not checked; a warning is given when windows lie entirely outside the raster). For a TIFF without georeferencing, x and y are pixel units from the top-left corner of the image, y pointing down, so that the center of the top-left pixel is (0.5, 0.5).

extent	Side length of the window in map units, rounded to whole pixels at the resolution of raster (halves round up). The window is placed on the pixel grid of the raster: its center lies at most half a pixel from coords, and pixel values are never interpolated.
center, scale	NULL, or one value per channel (or a single value for all channels). Every channel is transformed as (value - center) / scale, after transform.
standardize	Logical. TRUE estimates center and scale as the mean and standard deviation of every channel over all windows of coords (after transform, without missing pixels). This reads every window once. Build it on the training coordinates only and pass like for validation, test and prediction sites, so that their statistics do not enter the scaling.
transform	NULL or a function applied to every batch before center and scale: it receives a float tensor [windows, bands, height, width] in the units of the raster, with NaN for missing pixels, and returns a tensor with the same first dimension, e.g. function(x) torch::torch_log1p(x) or a band ratio. Channels and spatial size may change. It must treat every window on its own: statistics over the batch would make predictions depend on the batch size.
fill	Value for missing pixels (outside the raster, NoData, NaN, and non-finite results of transform), in the units after center and scale: with standardize = TRUE, the default 0 imputes the mean; without it, 0 is a value in the units of the raster, and na_channel = TRUE lets the network tell it from an observed 0.
na_channel	Logical. TRUE appends one channel that is 1 where any channel of the pixel is missing and 0 otherwise, so that the network can tell an imputed pixel from an observed one.
like	A citoraster whose raster, extent and preprocessing are reused for new coords, e.g. to predict at new sites with the center and scale estimated on the training windows. Only coords may be given together with like.
x	A citoraster.
...	Not used.

Details

Reading speed depends almost entirely on how the raster is stored. Every read decompresses whole storage blocks. A raster stored in strips (the GeoTIFF default) has blocks of full-width rows, so a window of 100 x 100 pixels from a 37800 pixel wide raster decompresses 100 rows of 37800 pixels; a tiled raster has square blocks, and a window decompresses only the few tiles it overlaps. The gain grows with the width of the raster: in our benchmarks a tiled copy was about 2 to 7 times faster for rasters 10000 to 12000 pixels wide and roughly 35 to 125 times faster for one 37800 pixels wide, and tiles of about the window size were best (128 x 128 pixel tiles about twice as fast as 256 x 256 for windows of 100 pixels). Whether a file is tiled is shown by X$info$tiled and X$info$block, and a large raster in strips gives a warning. Converting it is a single step, done once before training: terra::writeRaster(terra::rast("x.tif"), "x_tiled.tif", gdal = c("TILED=YES", "BLOCKXSIZE=128","BLOCKYSIZE=128", "COMPRESS=DEFLATE", "PREDICTOR=2")), with "PREDICTOR=3" for floating point rasters, or outside R, with less memory, gdal_translate -co TILED=YES -co BLOCKXSIZE=128 -co BLOCKYSIZE=128 -co COMPRESS=DEFLATE -co PREDICTOR=2 -co BIGTIFF=IF_SAFER x.tif x_tiled.tif. The values are unchanged; only their arrangement in the file differs. Only the

first image of the file is read (no overviews). The number of reader threads is `num_threads` of `cnn`; `mmn` uses 4.

Missing pixels are those outside the raster, `NaN`, and pixels equal to the NoData value of the GDAL_NODATA tag. Masks, alpha bands (read as an ordinary band), and scale and offset metadata are not applied; apply an offset or scale factor with `center` and `scale` or `transform`.

Every batch is preprocessed in this order: the raw values are read (missing pixels as `NaN`), `transform` is applied, then `(value - center) / scale`, then missing pixels are set to `fill` and the `na_channel` is appended. Unlike images read from a folder, values are not rescaled to the unit interval: rasters carry physical quantities, so normalization is left to these arguments. Preprocessing is part of the `citoraster`, so a model fitted on it predicts on the same scale; for new sites pass `like`. `predict()` refuses windows of another shape and warns when the preprocessing differs from that of the training windows.

The raster is opened once per session and kept open. A file replaced by another one with the same modification time is not noticed; a file whose dimensions changed is refused.

Value

An object of class `citoraster`, which can be subset by observation with `[`.

`as.array()`: the windows as the network receives them, an array `[windows, channels, height, width]` with all preprocessing applied. It reads every window of `x`, so subset first, e.g. `as.array(X[1:16])`.

Examples

```
if(torch::torch_is_installed() && cito:::cito_decoder_available()[["tiff"]]){
library(cito)

# a small georeferenced raster stands in for a large satellite scene
f = system.file("extdata", "raster.tif", package = "cito")
coords = cbind(x = runif(200, 600100, 600900), y = runif(200, 5300100, 5300700))
# log-transform, then standardize every band with the mean and sd over the windows;
# missing pixels become 0 (the mean) and are flagged in an extra channel
X = raster_patches(f, coords, extent = 200, transform = function(x) torch::torch_log1p(x),
                   standardize = TRUE, na_channel = TRUE)
X
# the first four windows as the network sees them: [windows, channels, height, width]
dim(as.array(X[1:4]))
Y = matrix(rnorm(200), ncol = 1)

nn.fit = cnn(X = X, Y = Y, loss = "mse", epochs = 2L, batchsize = 32L,
             architecture = create_architecture(conv(4), maxPool(), linear(10)))

# new sites, with the preprocessing of the training windows
new_coords = cbind(x = runif(5, 600100, 600900), y = runif(5, 5300100, 5300700))
predict(nn.fit, newdata = raster_patches(coords = new_coords, like = X))
}
```

residuals.citodnn *Extract Model Residuals*

Description

Returns residuals of training set.

Usage

```
## S3 method for class 'citodnn'
residuals(object, ...)
```

Arguments

object	a model created by dnn
...	no additional arguments implemented

Value

residuals of training set

simulate_shapes *Data Simulation for CNN*

Description

generates images of rectangles and ellipsoids

Usage

```
simulate_shapes(n, size, channels = 1)
```

Arguments

n	number of images
size	size of the (quadratic) images
channels	number of channels the generated data has (in each channel a new rectangle/ellipsoid is created)

Details

This function generates simple data to demonstrate the usage of cnn(). The generated images are of centered rectangles and ellipsoids with random widths and heights.

Value

array of dimension (n, 1, size, size)

Author(s)

Armin Schenk

`summary.citocnn` *Summarize a fitted CNN model*

Description

This function provides a summary of a Convolutional Neural Network (CNN) model created using the `cnn` function. It currently replicates the output of the `print.citocnn` method.

Usage

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

Arguments

`object`	A model created by `cnn`.
`...`	Additional arguments (currently not used).

Value

The original model object `object`, returned invisibly.

`summary.citodnn` *Summarize a Neural Network of class citodnn*

Description

Calculates feature importance and average conditional effects for a trained model.

Usage

```
## S3 method for class 'citodnn'
summary(
  object,
  importance = c("ale", "ce", "permutation"),
  n_permute = 10L,
  device = NULL,
  type = c("response", "link"),
  ...
)

## S3 method for class 'citodnnBootstrap'
summary(
  object,
  importance = c("ce", "permutation"),
  n_permute = 10,
  device = NULL,
  adjust_se = FALSE,
  type = c("response", "link"),
  ...
)
```

Arguments

object	a model of class citodnn created by dnn
importance	method used to compute feature importance: "ale" (default) or "ce" (both derived from the conditional effects, numerical features only), or "permutation" (all features, computationally expensive). For bootstrapped models "ce" (default) or "permutation".
n_permute	number of permutations used to compute the permutation feature importance. Default is 10.
device	device used to calculate the feature importance and conditional effects
type	scale on which the average conditional effects are calculated ("response" or "link")
...	additional arguments (currently not used)
adjust_se	if TRUE, the standard errors of the importance are multiplied by 1/sqrt(3)

Details

The summary reports the feature importance (following Fisher, Rudin, and Dominici, 2018) together with the mean and standard deviation of the average conditional effects (following Pichler & Hartig, 2023).

Feature importances are interpreted similarly to an ANOVA: main and interaction effects are absorbed into the individual features. They are also sensitive to collinearity between features, i.e. if two features are collinear, their importances may be overestimated.

Average conditional effects (ACE) are similar to marginal effects and approximate linear effects, i.e. their interpretation is similar to the coefficients in a linear regression model.

The standard deviation of the ACE quantifies the non-linearity of a feature effect: higher values indicate stronger non-linearities.

For the permutation importance, the loss of the model is evaluated on the original and on the permuted feature (e_{perm} and e_{orig}), and the importance of feature j is reported as the relative increase $FI_j = e_{perm}/e_{orig} - 1$.

The "ale" and "ce" importances are derived from the conditional effects and are therefore only defined for numerical features; categorical features are left out of the importance table (a message says which). Use `importance = "permutation"` to include them. For bootstrapped models, only "ce" and "permutation" are available.

Value

an object of class "summary.citodnn" containing the feature importance and average conditional effects

---

`summary.citommn` *Summarize a fitted MMN model*

Description

Reports the permutation importance of each data stream: one stream is shuffled across the batch while the others are left intact, and the increase in loss is what that stream contributed.

Usage

```
## S3 method for class 'citommn'
summary(object, n_permute = 10L, device = NULL, ...)

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

Arguments

`object`	A model created by `mmn`.
`n_permute`	Number of permutations per data stream. Default is 10.
`device`	Device on which to evaluate. Defaults to the device the model was trained on.
`...`	Additional arguments (currently not used).
`x`	a summary of a fitted MMN

Details

This is the stream-level analogue of permutation importance for a DNN, and is the interpretive tool a multi-modal model admits, since per-feature effects are not comparable across modalities of different kinds. The standard deviation across permutations is reported beside the estimate, because a stream that contributes little can come out negative by chance.

Value

An object of class "summary.citommn" holding one row per data stream with the increase in loss, its standard deviation across permutations, and the share of the total.

---

sumTerms *combine a list of formula terms as a sum*

Description

combine a list of formula terms as a sum

Usage

```
sumTerms(termList)
```

Arguments

termList	a list of formula terms

---

transfer *Include a Pretrained Model in a CNN Architecture*

Description

This function creates a transfer layer object of class citolayer for use in constructing a Convolutional Neural Network (CNN) architecture. The resulting layer object allows the use of pretrained models available in the 'torchvision' package within cito.

Usage

```
transfer(
  name = c("alexnet", "inception_v3", "mobilenet_v2", "resnet101", "resnet152",
   "resnet18", "resnet34", "resnet50", "resnext101_32x8d", "resnext50_32x4d", "vgg11",
    "vgg11_bn", "vgg13", "vgg13_bn", "vgg16", "vgg16_bn", "vgg19", "vgg19_bn",
    "wide_resnet101_2", "wide_resnet50_2"),
  pretrained = TRUE,
  freeze = TRUE,
  rgb = TRUE
)
```

Arguments

name	(character) The name of the pretrained model. Available options include: "alexnet", "inception_v3", "mobilenet_v2", "resnet101", "resnet152", "resnet18", "resnet34", "resnet50", "resnext101_32x8d", "resnext50_32x4d", "vgg11", "vgg11_bn", "vgg13", "vgg13_bn", "vgg16", "vgg16_bn", "vgg19", "vgg19_bn", "wide_resnet101_2", "wide_resnet50_2".
pretrained	(boolean) If TRUE, the model uses its pretrained weights. If FALSE, random weights are initialized.
freeze	(boolean) If TRUE, the weights of the pretrained model (except the "classifier" part at the end) are not updated during training. This setting only applies if pretrained = TRUE.
rgb	(boolean) If FALSE, the pretrained weights of the first convolutional layer are averaged across the channel dimension. This is useful if your data has 3 channels but isn't an RGB image. This setting only applies if pretrained = TRUE.

Details

This function creates a transfer layer object, which represents a pretrained model of the torchvision package with the linear "classifier" part removed. This allows the pretrained features of the model to be utilized while enabling customization of the classifier. When using this function with create_architecture, only linear layers can be added after the transfer layer. These linear layers define the "classifier" part of the network. If no linear layers are provided following the transfer layer, the default classifier will consist of a single output layer.

Additionally, the pretrained argument specifies whether to use the pretrained weights or initialize the model with random weights. If freeze is set to TRUE, only the weights of the final linear layers (the "classifier") are updated during training, while the rest of the pretrained model remains unchanged. Note that freeze has no effect unless pretrained is set to TRUE.

If your data has three channels but is not an RGB image set rgb to FALSE to average the pretrained weights of the first convolutional layer, so that each channel is treated equally. This is also done if your data has more or less channels than 3.

Value

An S3 object of class "transfer" "citolayer", representing a pretrained model of the torchvision package in the CNN architecture.

Author(s)

Armin Schenk

See Also

create_architecture

Examples

```
if(torch::torch_is_installed()){
library(cito)

# Creates a "transfer" "citolayer" object that later tells the cnn() function that
# the alexnet architecture and its pretrained weights should be used, but none
# of the weights are frozen
alexnet <- transfer(name="alexnet", pretrained=TRUE, freeze=FALSE)

# Creates a "transfer" "citolayer" object that later tells the cnn() function that
# the resnet18 architecture and its pretrained weights should be used.
# Also all weights except from the linear layer at the end are frozen (and
# therefore not changed during training)
resnet18 <- transfer(name="resnet18", pretrained=TRUE, freeze=TRUE)
}
```

---

tree_data *Tree Data*

Description

A simulated data set of 500 sites for trying out multimodal networks (mmn): a binary image per site, three standardized environmental predictors, and three responses.

Usage

```
tree_data
```

Format

`tree_data`**:**

List with the following objects (500 observations)

images Satellite images

Env data.frame, Environment (tabulardata)

Species Species 0 or Species 1

Trees Number of Trees

dbh DBH in cm

tune *Tune hyperparameter*

Description

Control hyperparameter tuning

Usage

```
tune(
  lower = NULL,
  upper = NULL,
  fixed = NULL,
  additional = NULL,
  values = NULL
)
```

Arguments

lower	numeric, numeric vector, character, lower boundaries of tuning space
upper	numeric, numeric vector, character, upper boundaries of tuning space
fixed	character, used for multi-dimensional hyperparameters such as hidden, which dimensions should be fixed
additional	numeric, additional control parameter which sets the value of the fixed argument
values	custom values from which hyperparameters are sampled, must be a matrix for hidden layers (first column == nodes, second column == number of layers)

Value

An object of class "tune" that marks a hyperparameter of dnn for tuning.

Index

∗ **datasets**
    tree_data, 82

ALE, 3, *8*, *43*, *60*, *62, 63*
analyze_training, 5, *8*, *17*, *43*, *57*
as.array.citoraster *(*raster_patches*)*, 73
avgPool, 6, *32*, *34*

barplot, *64*

cito, 8
cito-package *(*cito*)*, 8
cnn, *6*, *8*, 13, *19*, *28–30*, *32*, *34*, *56*, *61*, *64*, *69*, *73*, *75*, *77*
coef.citocnn, *17*, 19
coef.citodnn, 20, *43*
coef.citodnnBootstrap *(*coef.citodnn*)*, 20
coef.citommn, 21, *57*
conditionalEffects, 22, *37*, *72*
config_lr_scheduler, *14*, 24, *36*, *41*, *55*
config_optimizer, *14*, 26, *36*, *41*, *54*
config_tuning, 27, *37*, *42*
continue_training, *8*, *17*, 28, *43*, *57*
conv, 30, *32*, *34*
create_architecture, *6, 7*, *14*, *16*, *30, 31*, 32, *50–52*, *60*, *63*, *68*, *81*

dnn, *4*, *6*, *8*, *14, 15*, *20*, *24–26*, *28–30*, 34, *48*, *50*, *56*, *59*, *62*, *66*, *70*, *76*, *78*, *83*
dnn(), *27*

e, *39*, 47

fe, *39, 40*, 49
findReTrmClasses, 50
formula, *35*

linear, *32*, *34*, 50
lr_lambda, *25*
lr_multiplicative, *25*
lr_one_cycle, *25*
lr_reduce_on_plateau, *25*
lr_step, *25*

maxPool, *32*, *34*, 52
mmn, *6*, *8*, *21*, *28–30*, 53, *63*, *67*, *71*, *73*, *75*, *79*, *82*

nn_dropout, *36*
nn_module, *14*, *36*, *42*

optim_adadelta, *26*
optim_adagrad, *26*
optim_adam, *26*
optim_rmsprop, *26*
optim_rprop, *26*
optim_sgd, *26*

PDP, *5*, *8*, *43*, 58
plot.citoarchitecture, *34*, 60
plot.citocnn, *17*, 61
plot.citodnn, *43*, 62
plot.citodnnBootstrap *(*plot.citodnn*)*, 62
plot.citommn, 63
predict, *4*, *59*
predict.citocnn, *17*, 64
predict.citodnn, *43*, 66
predict.citodnnBootstrap *(*predict.citodnn*)*, 66
predict.citommn, *57*, 67
print.citoarchitecture, *34*, 68
print.citocnn, *17*, 69, *77*
print.citodnn, *43*, 70
print.citodnnBootstrap *(*print.citodnn*)*, 70
print.citommn, *57*, 71
print.conditionalEffects, 72
print.conditionalEffectsBootstrap *(*print.conditionalEffects*)*, 72
print.summary.citodnn, 72
print.summary.citodnnBootstrap *(*print.summary.citodnn*)*, 72

print.summary.citommn
(summary.citommn), 79

raster_patches, 8, 14, 64, 73
residuals.citodnn, 76

simulate_shapes, 76
summary.citocnn, 17, 77
summary.citodnn, 8, 43, 72, 77
summary.citodnnBootstrap
(summary.citodnn), 77
summary.citommn, 57, 79
sumTerms, 80

torch_serialize, 43
transfer, 32, 34, 80
tree_data, 82
tune, 83
tune(), 42