DeeProb-kit

Release 1.1.0

The Deep Probabilistic Modeling Organization

May 28, 2023

HOME PAGE

CHAPTER

ONE

DEEPROB-KIT

DeeProb-kit is a unified library written in Python consisting of a collection of deep probabilistic models (DPMs) that are tractable and exact representations for the modelled probability distributions. The availability of a representative selection of DPMs in a single library makes it possible to combine them in a straightforward manner, a common practice in deep learning research nowadays. In addition, it includes efficiently implemented learning techniques, inference routines, statistical algorithms, and provides high-quality fully-documented APIs. The development of DeeProb-kit will help the community to accelerate research on DPMs as well as to standardise their evaluation and better understand how they are related based on their expressivity.

1.1 Features

- Inference algorithms for SPNs.[1][4]
- Learning algorithms for SPNs structure.[1][2][3][4][5]
- Chow-Liu Trees (CLT) as SPN leaves.[13]
- Cutset Networks (CNets) with various learning criteria.[12]
- Batch Expectation-Maximization (EM) for SPNs with arbitrarily leaves.[14][15]
- Structural marginalization and pruning algorithms for SPNs.
- High-order moments computation for SPNs.
- JSON I/O operations for SPNs and CLTs.[4]
- Plotting operations based on NetworkX for SPNs and CLTs.[4]
- Randomized And Tensorized SPNs (RAT-SPNs).[6]
- Deep Generalized Convolutional SPNs (DGC-SPNs).[11]
- Masked Autoregressive Flows (MAFs).[7]

[1] Peharz et al. *On Theoretical Properties of Sum-Product Networks*. AISTATS (2015).
[4] Molina, Vergari et al. *SPFLOW : An easy and extensible library for deep probabilistic learning using Sum-Product Networks*. CoRR (2019).
[2] Poon and Domingos. *Sum-Product Networks: A New Deep Architecture*. UAI (2011).
[3] Molina, Vergari et al. *Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains*. AAAI (2018).
[5] Di Mauro et al. *Sum-Product Network structure learning by efficient product nodes discovery*. AIxIA (2018).
[13] Di Mauro, Gala et al. *Random Probabilistic Circuits*. UAI (2021).
[12] Rahman et al. *Cutset Networks: A Simple, Tractable, and Scalable Approach for Improving the Accuracy of Chow-Liu Trees*. ECML-PKDD (2014).
[14] Desana and Schnörr. *Learning Arbitrary Sum-Product Network Leaves with Expectation-Maximization*. CoRR (2016).
[15] Peharz et al. *Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits*. ICML (2020).
[6] Peharz et al. *Probabilistic Deep Learning using Random Sum-Product Networks*. UAI (2020).
[11] Van de Wolfshaar and Pronobis. *Deep Generalized Convolutional Sum-Product Networks for Probabilistic Image Representations*. PGM (2020).
[7] Papamakarios et al. *Masked Autoregressive Flow for Density Estimation*. NeurIPS (2017).

- Real Non-Volume-Preserving (RealNVP) flows.[8]
- Non-linear Independent Component Estimation (NICE) flows.[9]

The collection of implemented models is summarized in the following table.

Model	Description
Binary-CLT	Binary Chow-Liu Tree (CLT)
Binary-CNet	Binary Cutset Network (CNet)
SPN	Vanilla Sum-Product Network
MSPN	Mixed Sum-Product Network
XPC	Random Probabilistic Circuit
RAT-SPN	Randomized and Tensorized Sum-Product Network
DGC-SPN	Deep Generalized Convolutional Sum-Product Network
MAF	Masked Autoregressive Flow
NICE	Non-linear Independent Components Estimation Flow
RealNVP	Real-valued Non-Volume-Preserving Flow

1.2 Installation

The library can be installed either from PIP repository or by source code.

```
# Install from PIP repository
pip install deeprob-kit
```

```
# Install from `main` git branch
pip install -e git+https://github.com/deeprob-org/deeprob-kit.git@main#egg=deeprob-kit
```

1.3 Project Directories

The documentation is generated automatically by Sphinx using sources stored in the *docs* directory.

A collection of code examples and experiments can be found in the *examples* and *experiments* directories respectively. Moreover, benchmark code can be found in the *benchmark* directory.

1.4 Cite

```
@misc{loconte2022deeprob,
  doi = {10.48550/ARXIV.2212.04403},
  url = {https://arxiv.org/abs/2212.04403},
  author = {Loconte, Lorenzo and Gala, Gennaro},
  title = {{DeeProb-kit}: a Python Library for Deep Probabilistic Modelling},
  publisher = {arXiv},
  year = {2022}
}
```

[8] Dinh et al. *Density Estimation using RealNVP*. ICLR (2017).
[9] Dinh et al. *NICE: Non-linear Independent Components Estimation*. ICLR (2015).

1.5 Related Repositories

- SPFlow
- RAT-SPN
- Random-PC
- LibSPN-Keras
- MAF
- RealNVP

1.6 References

CHAPTER

TWO

DOCUMENTATION

The source code documentation is hosted using GitHub Pages at deeprob-kit.readthedocs.io/en/latest.

The documentation is generated automatically by Sphinx, using sources stored in the `docs` directory (with a *Read-the-Docs* theme).

If you wish to build the documentation yourself, you will need to install the required dependencies from the root directory as follows.

```
pip install -e .[docs]
```

Finally, it's possible to execute the Makefile script as following:

```
# Clean existing documentation (optional)
make clean
# Build HTML documentation
make html
```

The output HTML documentation can be found inside the `_build` directory.

CHAPTER

THREE

CODE EXAMPLES

A collection of code examples can be found in the `examples` directory. In order to run the code examples, it is necessary clone the repository. However, additional datasets are not required. Note that the given examples are not intended to produce state-of-the-art results, but only to present the library.

The following table contains a description about them and a code complexity ranging from one to three stars. The *Complexity* column consists of a measure that roughly represents how many features of the library are used, as well as the expected time required to run the script.

Example	Description	Complexity
naive_model.py	Learn, evaluate and print statistics about a naive factorized model.	
spn_plot.py	Instantiate, prune, marginalize and plot some SPNs.	
clt_plot.py	Learn a Binary-CLT and plot it.	
cnet_bd.py	Learn a Binary-CNet using the BDeu score criteria.	
spn_moments.py	Instantiate and compute moments statistics about the random variables.	
sklearn_interface.py	Learn and evaluate a SPN using the scikit-learn interface.	
spn_custom_leaf.py	Learn, evaluate and serialize a SPN with a user-defined leaf distribution.	
clt_to_spn.py	Learn a Binary-CLT, convert it to a structured decomposable SPN and plot it.	
spn_clt_em.py	Instantiate a SPN with Binary CLTs, apply EM algorithm and sample some data.	
clt_queries.py	Learn a Binary-CLT, plot it, run some queries and sample some data.	
ratspn_mnist.py	Train and evaluate a RAT-SPN on MNIST.	
dgcspn_olivetti.py	Train, evaluate and complete some images with DGC-SPN on Olivetti-Faces.	
dgcspn_mnist.py	Train and evaluate a DGC-SPN on MNIST.	
nvp1d_moons.py	Train and evaluate a 1D RealNVP on Moons dataset.	
rat-spn_nvp1d_mnist.py	Train and evaluate a 1D RealNVP with a RAT-SPN as base distribution.	
maf_cifar10.py	Train and evaluate a MAF on CIFAR10.	
nvp2d_mnist.py	Train and evaluate a 2D RealNVP on MNIST.	
nvp2d_cifar10.py	Train and evaluate a 2D RealNVP on CIFAR10.	
spn_latent_mnist.py	Train and evaluate a SPN on MNIST using the features extracted by an autoencoder.	

CHAPTER

FOUR

EXPERIMENTS

A collection of 29 binary datasets, which most of them are used in *Probabilistic Circuits* literature, can be found at UCLA-StarAI-Binary-Datasets. Moreover, a collection of 5 continuous datasets, commonly present in works regarding *Normalizing Flows*, can be found at MAF-Continuous-Datasets.

In order to run the experiments, it is necessary to clone the repository. After downloading the datasets, they must be stored in the `experiments/datasets` directory to be able to run the experiments. Finally, install the development packages from the root directory as follows.

```
pip install -e .[develop]
```

The experiments scripts are available in the `experiments` directory and can be launched using the command line by specifying the dataset and hyperparameters. The following table shows the available experiments scripts.

Experiment	Description
energy.py	Fit Sum-Product Networks (SPNs) and Normalizing Flows on energy functions.[1]
spn.py	Experiments for Sum-Product Networks (SPNs).
ratspn.py	Experiments for Randomized And Tensorized Sum-Product Networks (RAT-SPNs).
dgcspn.py	Experiments for Deep Generalized Convolutional Sum-Product Networks (DGC-SPNs).
flows.py	Experiments for several Normalizing Flows models.

[1] Rezende and Mohamed. *Variational Inference with Normalizing Flows*. ICML (2015).

CHAPTER

FIVE

BENCHMARK

The `benchmark` directory contains benchmark scripts of models and algorithms implemented in the library. All the scripts can be launched by command line. Every script will print an estimation of the time required to run each algorithm (in seconds) to a JSON file. In order to run the benchmarks, install the development packages from the root directory as follows.

```
pip install -e .[develop]
```

CHAPTER

SIX

DEEPROB

6.1 deeprob package

6.1.1 Subpackages

deeprob.flows package

Subpackages

deeprob.flows.layers package

Submodules

deeprob.flows.layers.autoregressive module

class deeprob.flows.layers.autoregressive.**AutoregressiveLayer**(*in_features*, *depth*, *units*, *activation*, *reverse=False*, *sequential=True*, *random_state=None*)

Bases: *Bijector*

Build an autoregressive layer as specified in Masked Autoregressive Flow paper.

Parameters

- **in_features** (*int*) – The number of input features.
- **depth** (*int*) – The number of hidden layers of the conditioner.
- **units** (*int*) – The number of units of each hidden layer of the conditioner.
- **activation** (*str*) – The activation used for inner layers of the conditioner.
- **reverse** (*bool*) – Whether to reverse the mask used in the autoregressive layer. Used only if sequential is True.
- **sequential** (*bool*) – Whether to use sequential degrees for inner layers masks.
- **random_state** (*Optional[RandomState]*) – The random state used to generate the masks degrees. Used only if sequential is False.

Raises

ValueError – If a parameter is out of domain.

`apply_backward`(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`apply_forward`(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`build_degrees_sequential`(*depth*, *units*, *reverse*)

Build sequential degrees for the linear layers of the autoregressive network.

Parameters

- **depth** (*int*) – The number of hidden layers of the conditioner.
- **units** (*int*) – The number of units of each hidden layer of the conditioner.
- **reverse** (*bool*) – Whether to reverse the mask used in the autoregressive layer. Used only if sequential is True.

Returns
The masks to use for each hidden layer of the autoregressive network.

Return type
List[*ndarray*]

`build_degrees_random`(*depth*, *units*, *random_state*)

Create random degrees for the linear layers of the autoregressive network.

Parameters

- **depth** (*int*) – The number of hidden layers of the conditioner.
- **units** (*int*) – The number of units of each hidden layer of the conditioner.
- **random_state** (*RandomState*) – The random state.

Returns
The masks to use for each hidden layer of the autoregressive network.

Return type
List[*ndarray*]

`static build_masks`(*degrees*)

Build masks from degrees.

Parameters
degrees (*List[ndarray]*) – A sequence of units degrees for each hidden layer.

Returns
The masks to use for each hidden layer of the autoregressive network.

Return type
List[*ndarray*]

`training: bool`

deeprob.flows.layers.coupling module

`class deeprob.flows.layers.coupling.CouplingLayer1d(in_features, depth, units, affine=True, reverse=False)`

Bases: *Bijector*

Build a RealNVP (or NICE) 1D coupling layer.

Parameters

- **in_features** (*int*) – The number of input features.
- **depth** (*int*) – The number of hidden layers of the conditioner.
- **units** (*int*) – The number of units of each hidden layer of the conditioner.
- **affine** (*bool*) – Whether to use affine transformation. If False then use only translation (as in NICE).
- **reverse** (*bool*) – Whether to reverse the mask used in the coupling layer. Useful for alternating masks.

`build_alternating_masks()`

Build the alterning masks.

Returns
The alterning mask and its inverse.

Return type
Tuple[*ndarray*, *ndarray*]

`apply_backward(x)`

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`apply_forward(u)`

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

training: *bool*

class deeprob.flows.layers.coupling.**CouplingLayer2d**(*in_features*, *network*, *n_blocks*, *channels*, *affine=True*, *channelwise=False*, *reverse=False*)

Bases: *Bijector*

Build a RealNVP (or NICE) 2D coupling layer.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The size of the input.
- **network** (*str*) – The network conditioner to use. It can be either 'resnet' or 'densenet'.
- **n_blocks** (*int*) – The number of residual blocks or dense blocks.
- **channels** (*int*) – The number of output channels of each convolutional layer.
- **affine** (*bool*) – Whether to use affine transformation. If False then use only translation (as in NICE).
- **channelwise** (*bool*) – Whether to use channel-wise coupling mask. Defaults to False, i.e. checkerboard coupling mask.
- **reverse** (*bool*) – Whether to reverse the mask used in the coupling layer. Useful for alternating masks.

property in_channels: *int*

property in_height: *int*

property in_width: *int*

build_checkerboard_masks()

Build the checkerboard coupling masks.

Returns
The checkerboard mask and its inverse.

Return type
Tuple[*ndarray*, *ndarray*]

apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

apply_forward(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

class deeprob.flows.layers.coupling.**CouplingBlock2d**(*in_features*, *network*, *n_blocks*, *channels*, *affine=True*, *last_block=False*)

Bases: *Bijector*

Build a RealNVP (or NICE) 2D coupling block, consisting of check-board/channel-wise couplings and squeeze operation.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The size of the input.
- **network** (*str*) – The network conditioner to use. It can be either 'resnet' or 'densenet'.
- **n_blocks** (*int*) – The number of residual blocks or dense blocks.
- **channels** (*int*) – The number of output channels of each convolutional layer.
- **affine** (*bool*) – Whether to use affine transformation. If False then use only translation (as in NICE).
- **last_block** (*bool*) – Whether it is the last block (i.e. no channelwise-masked couplings) or not.

property in_channels: int

property in_height: int

property in_width: int

apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

apply_forward(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

deeprob.flows.layers.densenet module

class deeprob.flows.layers.densenet.**DenseLayer**(*in_channels*, *out_channels*, *use_checkpoint=False*)

Bases: Module

Initialize a dense layer as in DenseNet.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **out_channels** (*int*) – The number of output channels.
- **use_checkpoint** (*bool*) – Whether to use a checkpoint in order to reduce memory usage (by increasing training time caused by re-computations).

bottleneck(*inputs*)

Pass through the bottleneck layer.

Parameters
inputs (*List[Tensor]*) – A list of previous feature maps.

Returns
The outputs of the bottleneck.

Return type
Tensor

checkpoint_bottleneck(*inputs*)

Pass through the bottleneck layer (by using a checkpoint).

Parameters
inputs (*List[Tensor]*) – A list of previous feature maps.

Returns
The outputs of the bottleneck.

Return type
Tensor

forward(*inputs*)

Evaluate the dense layer.

Parameters
inputs (*List[Tensor]*) – A list of previous feature maps.

Returns
The outputs of the layer.

Return type
Tensor

training: bool

class deeprob.flows.layers.densenet.**DenseBlock**(*n_layers*, *in_channels*, *out_channels*, *use_checkpoint=False*)

Bases: Module

Initialize a dense block as in DenseNet.

Parameters

- **n_layers** (*int*) – The number of dense layers.

- **in_channels** (*int*) – The number of input channels.
- **out_channels** (*int*) – The number of output channels.
- **use_checkpoint** (*bool*) – Whether to use a checkpoint in order to reduce memory usage (by increasing training time caused by re-computations).

forward(*x*)

Evaluate the dense block.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

training: bool

class deeprob.flows.layers.densenet.**Transition**(*in_channels*, *out_channels*, *bias=True*)

Bases: Module

Initialize a transition layer as in DenseNet.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **out_channels** (*int*) – The number of output channels.
- **bias** (*bool*) – Whether to use bias in the last convolutional layer.

forward(*x*)

Evaluate the layer.

Parameters
x – The inputs.

Returns
The outputs of the layer.

training: bool

class deeprob.flows.layers.densenet.**DenseNetwork**(*in_channels*, *mid_channels*, *out_channels*, *n_blocks*, *use_checkpoint=False*)

Bases: Module

Initialize a dense network (DenseNet) with only one dense block.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **mid_channels** (*int*) – The number of mid channels.
- **out_channels** (*int*) – The number of output channels.
- **n_blocks** (*int*) – The number of dense blocks.
- **use_checkpoint** (*bool*) – Whether to use a checkpoint in order to reduce memory usage (by increasing training time caused by re-computations).

forward(*x*)

Evaluate the dense network.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

training: bool

deeprob.flows.layers.resnet module

class deeprob.flows.layers.resnet.**ResidualBlock**(*n_channels*)

Bases: Module

Build a basic residual block as in ResNet.

Parameters
n_channels (*int*) – The number of channels.

forward(*x*)

Evaluate the residual block.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

training: bool

class deeprob.flows.layers.resnet.**ResidualNetwork**(*in_channels*, *mid_channels*, *out_channels*, *n_blocks*)

Bases: Module

Initialize a residual network (ResNet) with skip connections.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **mid_channels** (*int*) – The number of mid channels.
- **out_channels** (*int*) – The number of output channels.
- **n_blocks** (*int*) – The number of residual blocks.

Raises
ValueError – If a parameter is out of domain.

forward(*x*)

Evaluate the residual network.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

`training: bool`

Module contents

deeprob.flows.models package

Submodules

deeprob.flows.models.base module

`class deeprob.flows.models.base.NormalizingFlow`(*in_features*, *dequantize=False*, *logit=None*, *in_base=None*)

Bases: `ProbabilisticModel`

Initialize an abstract Normalizing Flow model.

Parameters

- **in_features** – The input size.
- **dequantize** (*bool*) – Whether to apply the dequantization transformation.
- **logit** (*Optional[float]*) – The logit factor to use. Use None to disable the logit transformation.
- **in_base** (*Optional[Union[*ProbabilisticModel*, Distribution]]*) – The input base distribution to use. If None, the standard Normal distribution is used.

Raises

- **ValueError** – If the number of input features is invalid.
- **ValueError** – If the logit value is invalid.

`has_rsample = True`

`train`(*mode=True*, *base_mode=True*)

Set the training mode.

Parameters

- **mode** (*bool*) – The training mode for the flows layers.
- **base_mode** (*bool*) – The training mode for the in_base distribution.

Returns
Itself.

`eval()`

Turn off the training mode for both the flows layers and the in_base distribution.

Returns
Itself.

preprocess(*x*)

Preprocess the data batch before feeding it to the probabilistic model (forward mode).

Parameters
x (*Tensor*) – The input data batch.

Returns
The preprocessed data batch and the inv-log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

unpreprocess(*x*)

Preprocess the data batch before feeding it to the probabilistic model (backward mode).

Parameters
x (*Tensor*) – The input data batch.

Returns
The unpreprocessed data batch and the log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

forward(*x*)

Compute the log-likelihood given complete evidence.

Parameters
x (*Tensor*) – The inputs.

Returns
The log-likelihoods.

Return type
Tensor

sample(*n_samples*, *y=None*)

Sample some values from the modeled distribution.

Parameters

- **n_samples** (*int*) – The number of samples.
- **y** (*Optional[Tensor]*) – The samples labels. It can be None.

Returns
The samples.

Return type
Tensor

rsample(*n_samples*, *y=None*)

Sample some values from the modeled distribution by reparametrization. Unlike *NormalizingFlow.sample()*, this method allows backpropagation.

Parameters

- **n_samples** (*int*) – The number of samples.
- **y** (*Optional[Tensor]*) – The samples labels. It can be None.

Returns
The samples.

Return type
Tensor

`apply_backward`(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`training: bool`

`apply_forward`(*x*)

Apply the forward transformation.

Parameters
x (*Tensor*) – the inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`loss`(*x*, *y=None*)

Compute the loss of the model.

Parameters

- **x** (*Tensor*) – The outputs of the model.
- **y** (*Optional[Tensor]*) – The ground-truth. It can be None.

Returns
The loss.

Return type
Tensor

deeprob.flows.models.maf module

`class deeprob.flows.models.maf.MAF`(*in_features*, *dequantize=False*, *logit=None*, *in_base=None*, *n_flows=5*, *depth=1*, *units=128*, *batch_norm=True*, *activation='relu'*, *sequential=True*, *random_state=None*)

Bases: *NormalizingFlow*

Initialize a Masked Autorefressive Flow (MAF) model.

Parameters

- **in_features** (*int*) – The number of input features.
- **dequantize** (*bool*) – Whether to apply the dequantization transformation.
- **logit** (*Optional[float]*) – The logit factor to use. Use None to disable the logit transformation.

- **in_base** (*Optional[Union[ProbabilisticModel, Distribution]]*) – The input base distribution to use. If None, the standard Normal distribution is used.
- **n_flows** (*int*) – The number of sequential autoregressive layers.
- **depth** (*int*) – The number of hidden layers of flows conditioners.
- **units** (*int*) – The number of hidden units per layer of flows conditioners.
- **batch_norm** (*bool*) – Whether to apply batch normalization after each autoregressive layer.
- **activation** (*str*) – The activation function name to use for the flows conditioners hidden layers.
- **sequential** (*bool*) – If True build masks degrees sequentially, otherwise randomly.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state used to generate the masks degrees. Used only if sequential is False. It can be either a seed integer or a np.random.RandomState instance.

Raises

ValueError – If a parameter is out of domain.

training: bool

deeprob.flows.models.realnvp module

class deeprob.flows.models.realnvp.**RealNVP1d**(*in_features*, *dequantize=False*, *logit=None*, *in_base=None*, *n_flows=5*, *depth=1*, *units=128*, *batch_norm=True*, *affine=True*)

Bases: *NormalizingFlow*

Real Non-Volume-Preserving (RealNVP) 1D normalizing flow model.

Parameters

- **in_features** (*int*) – The number of input features.
- **dequantize** (*bool*) – Whether to apply the dequantization transformation.
- **logit** (*Optional[float]*) – The logit factor to use. Use None to disable the logit transformation.
- **in_base** (*Optional[Union[ProbabilisticModel, Distribution]]*) – The input base distribution to use. If None, the standard Normal distribution is used.
- **n_flows** (*int*) – The number of sequential coupling flows.
- **depth** (*int*) – The number of hidden layers of flows conditioners.
- **units** (*int*) – The number of hidden units per layer of flows conditioners.
- **batch_norm** (*bool*) – Whether to apply batch normalization after each coupling layer.
- **affine** (*bool*) – Whether to use affine transformation. If False then use only translation (as in NICE).

Raises

ValueError – If a parameter is out of scope.

training: bool

class deeprob.flows.models.realnvp.**RealNVP2d**(*in_features*, *dequantize=False*, *logit=None*, *in_base=None*, *network='resnet'*, *n_flows=1*, *n_blocks=2*, *channels=32*, *affine=True*)

Bases: *NormalizingFlow*

Real Non-Volume-Preserving (RealNVP) 2D normalizing flow model.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The input size as a (C, H, W) tuple.
- **dequantize** (*bool*) – Whether to apply the dequantization transformation.
- **logit** (*Optional[float]*) – The logit factor to use. Use None to disable the logit transformation.
- **in_base** (*Optional[Union[ProbabilisticModel, Distribution]]*) – The input base distribution to use. If None, the standard Normal distribution is used.
- **network** (*str*) – The neural network conditioner to use. It can be either 'resnet' or 'densenet'.
- **n_flows** (*int*) – The number of sequential multi-scale architectures.
- **n_blocks** (*int*) – The number of residual blocks or dense blocks.
- **channels** (*int*) – The number of output channels of each convolutional layer.
- **affine** (*bool*) – Whether to use affine transformation. If False then use only translation (as in NICE).

static build_permutation_matrix(*channels*)

Build the permutation matrix that defines (a non-trivial) variables ordering when downscaling or upscaling as in RealNVP.

Parameters
channels (*int*) – The number of input channels.

Returns
The permutation matrix tensor.

Return type
Tensor

apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

apply_forward(*x*)

Apply the forward transformation.

Parameters
x (*Tensor*) – the inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`training: bool`

Module contents

Submodules

deeprob.flows.utils module

deeprob.flows.utils.**squeeze_depth2d**(*x*)

Squeeze operation (as in RealNVP).

Parameters
x (*Tensor*) – The input tensor of size [N, C, H, W].

Returns
The output tensor of size [N, C * 4, H // 2, W // 2].

Return type
Tensor

deeprob.flows.utils.**unsqueeze_depth2d**(*x*)

Un-squeeze operation (as in RealNVP).

Parameters
x (*Tensor*) – The input tensor of size [N, C * 4, H // 2, W // 2].

Returns
The output tensor of size [N, C, H, W].

Return type
Tensor

class deeprob.flows.utils.**Bijector**(*in_features*)

Bases: ABC, Module

Initialize a bijector module.

Parameters
in_features (*Union[int, Tuple[int, int, int]]*) – The number of input features.

Raises
ValueError – If the number of input features is invalid.

forward(*x*, *backward=False*)

Apply the bijector transformation.

Parameters

- **x** (*Tensor*) – The inputs.
- **backward** (*bool*) – Whether to apply the backward transformation.

Returns
The transformed samples and the corresponding log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

abstract apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

abstract apply_forward(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

class deeprob.flows.utils.**BatchNormLayer1d**(*in_features*, *momentum=0.9*, *eps=1e-05*)

Bases: *Bijector*

Build a Batch Normalization 1D layer.

Parameters

- **in_features** (*int*) – The number of input features.
- **momentum** (*float*) – The momentum used to update the running parameters.
- **eps** (*float*) – Epsilon value, an arbitrarily small value.

Raises
ValueError – If a parameter is out of domain.

apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

apply_forward(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

class deeprob.flows.utils.**BatchNormLayer2d**(*in_features*, *momentum=0.9*, *eps=1e-05*)

Bases: *Bijector*

Build a Batch Normalization 2D layer.

Parameters

- **in_features** (*int*) – The number of input features.
- **momentum** (*float*) – The momentum used to update the running parameters.
- **eps** (*float*) – An arbitrarily small value.

Raises
ValueError – If a parameter is out of domain.

apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

apply_forward(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

class deeprob.flows.utils.**DequantizeLayer**(*in_features*, *n_bits=8*)

Bases: *Bijector*

Build a Dequantization transformation layer.

Parameters

- **in_features** (*Union[int, Tuple[int, int, int]]*) – The number of input features.
- **n_bits** (*int*) – The number of bits to use.

Raises
ValueError – If a parameter is out of domain.

apply_backward(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`apply_forward`(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`training: bool`

`class deeprob.flows.utils.LogitLayer`(*in_features*, *alpha=0.05*)

Bases: *Bijector*

Build a Logit transformation layer.

Parameters

- **in_features** (*Union[int, Tuple[int, int, int]]*) – The number of input features.
- **alpha** (*float*) – The alpha parameter for logit transformation.

Raises
ValueError – If a parameter is out of domain.

`apply_backward`(*x*)

Apply the backward transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The transformed samples and the backward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`apply_forward`(*u*)

Apply the forward transformation.

Parameters
u (*Tensor*) – The inputs.

Returns
The transformed samples and the forward log-det-jacobian.

Return type
Tuple[*Tensor*, *Tensor*]

`training: bool`

Module contents

deeprob.spn package

Subpackages

deeprob.spn.algorithms package

Submodules

deeprob.spn.algorithms.evaluation module

deeprob.spn.algorithms.evaluation.**parallel_layerwise_eval**(*layers*, *eval_func*, *reverse=False*, *n_jobs=- 1*)

Execute a function per node layerwise in parallel.

Parameters

- **layers** (*List[List[Node]]*) – The layers, i.e., the layered topological ordering.
- **eval_func** (*Callable[[Node], None]*) – The evaluation function for a given node.
- **reverse** (*bool*) – Whether to reverse the layered topological ordering.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention.

deeprob.spn.algorithms.evaluation.**eval_bottom_up**(*root*, *x*, *leaf_func*, *node_func*, *leaf_func_kwargs=None*, *node_func_kwargs=None*, *return_results=False*, *n_jobs=0*)

Evaluate the SPN bottom up given some inputs and leaves and nodes evaluation functions.

Parameters

- **root** (Node) – The root of the SPN.
- **x** (*ndarray*) – The inputs.
- **leaf_func** (*Callable[[Leaf, ndarray, Any], ndarray]*) – The function to compute at each leaf node.
- **node_func** (*Callable[[Node, ndarray, Any], ndarray]*) – The function to compute at each inner node.
- **leaf_func_kwargs** (*Optional[dict]*) – The optional parameters of the leaf evaluation function.
- **node_func_kwargs** (*Optional[dict]*) – The optional parameters of the inner nodes evaluation function.
- **return_results** (*bool*) – A flag indicating if this function must return the log likelihoods of each node of the SPN.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention. Set to 0 to disable.

Returns

The outputs. Additionally, it returns the output of each node.

Raises

ValueError – If a parameter is out of domain.

Return type
Union[*ndarray*, *Tuple*[*ndarray*, *ndarray*]]

deeprob.spn.algorithms.evaluation.**eval_top_down**(*root*, *x*, *lls*, *leaf_func*, *sum_func*, *leaf_func_kwargs=None*, *sum_func_kwargs=None*, *inplace=False*, *n_jobs=0*)

Evaluate the SPN top down given some inputs, the likelihoods of each node and a leaves evaluation function. The leaves to evaluate are chosen by following the nodes given by the sum nodes evaluation function.

Parameters

- **root** (*Node*) – The root of the SPN.
- **x** (*ndarray*) – The inputs with some NaN values.
- **lls** (*ndarray*) – The log-likelihoods of each node.
- **leaf_func** (*Callable[[Leaf, ndarray, Any], ndarray]*) – The leaves evaluation function.
- **sum_func** (*Callable[[Sum, ndarray, Any], ndarray]*) – The sum nodes evaluation function.
- **leaf_func_kwargs** (*Optional[dict]*) – The optional parameters of the leaf evaluation function.
- **sum_func_kwargs** (*Optional[dict]*) – The optional parameters of the sum nodes evaluation function.
- **inplace** (*bool*) – Whether to make inplace assignments.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention. Set to 0 to disable.

Returns
The NaN-filled inputs.

Raises
ValueError – If a parameter is out of domain.

Return type
ndarray

deeprob.spn.algorithms.gradient module

deeprob.spn.algorithms.gradient.**eval_backward**(*root*, *lls*)

Compute the log-gradients at each SPN node.

Parameters

- **root** (*Node*) – The root of the SPN.
- **lls** (*ndarray*) – The log-likelihoods at each node.

Returns
The log-gradients w.r.t. the nodes.

Raises
ValueError – If a parameter is out of domain.

Return type
ndarray

deeprob.spn.algorithms.inference module

deeprob.spn.algorithms.inference.**likelihood**(*root*, *x*, *return_results=False*, *n_jobs=0*)

Compute the likelihoods of the SPN given some inputs.

Parameters

- **root** (Node) – The root of the SPN.
- **x** (*ndarray*) – The inputs. They can be marginalized using NaNs.
- **return_results** (*bool*) – A flag indicating if this function must return the likelihoods of each node of the SPN.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention. Set to 0 to disable.

Returns

The likelihood values. Additionally, it returns the likelihood values of each node.

Return type

Union[*ndarray*, *Tuple*[*ndarray*, *ndarray*]]

deeprob.spn.algorithms.inference.**log_likelihood**(*root*, *x*, *return_results=False*, *n_jobs=0*)

Compute the logarithmic likelihoods of the SPN given some inputs.

Parameters

- **root** (Node) – The root of the SPN.
- **x** (*ndarray*) – The inputs. They can be marginalized using NaNs.
- **return_results** (*bool*) – A flag indicating if this function must return the log likelihoods of each node of the SPN.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention. Set to 0 to disable.

Returns

The log likelihood values. Additionally, it returns the log likelihood values of each node.

Return type

Union[*ndarray*, *Tuple*[*ndarray*, *ndarray*]]

deeprob.spn.algorithms.inference.**mpe**(*root*, *x*, *inplace=False*, *n_jobs=0*)

Compute the Most Probable Explanation of a SPN given some inputs.

Parameters

- **root** (Node) – The root of the SPN.
- **x** (*ndarray*) – The inputs. They can be marginalized using NaNs.
- **inplace** (*bool*) – Whether to make inplace assignments.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention. Set to 0 to disable.

Returns

The NaN-filled inputs.

Return type

ndarray

deeprob.spn.algorithms.inference.**node_likelihood**(*node*, *x*)

Compute the likelihood of a node given the list of likelihoods of its children.

Parameters

- **node** (Node) – The internal node.
- **x** (*ndarray*) – The array of likelihoods of the children.

Returns

The likelihoods of the node given the inputs.

Return type

ndarray

deeprob.spn.algorithms.inference.**node_log_likelihood**(*node*, *x*)

Compute the log-likelihood of a node given the list of log-likelihoods of its children.

Parameters

- **node** (Node) – The internal node.
- **x** (*ndarray*) – The array of log-likelihoods of the children.

Returns

The log-likelihoods of the node given the inputs.

Return type

ndarray

deeprob.spn.algorithms.inference.**leaf_mpe**(*node*, *x*)

Compute the maximum likelihood estimate of a leaf node.

Parameters

- **node** (Leaf) – The leaf node.
- **x** (*ndarray*) – The inputs with some NaN values.

Returns

The most proable explanation.

Return type

ndarray

deeprob.spn.algorithms.inference.**sum_mpe**(*node*, *lls*)

Choose the branch that maximize the posterior estimate likelihood.

Parameters

- **node** (Sum) – The sum node.
- **lls** (*ndarray*) – The log-likelihoods of the children nodes.

Returns

The branch that maximize the posterior estimate likelihood.

Return type

ndarray

deeprob.spn.algorithms.moments module

deeprob.spn.algorithms.moments.**moment**(*root*, *order=1*)

Compute non-central moments of a given order of a smooth and decomposable SPN.

Parameters

- **root** (Node) – The root of the SPN.
- **order** (*int*) – The order of the moment. If scalar, it will be used for all the random variables.

Returns
The non-central moments with respect to each variable in the scope.

Raises
ValueError – If the order of the moment is negative.

Return type
ndarray

deeprob.spn.algorithms.moments.**leaf_moment**(*node*, *x*, *order*)

Compute the moment of a leaf node.

Parameters

- **node** (Leaf) – The leaf node.
- **x** (*ndarray*) – The inputs of the leaf. Actually, it's used only to infer the output shape.
- **order** (*int*) – The order of the moment.

Returns
The moment of the leaf node.

Return type
ndarray

deeprob.spn.algorithms.moments.**expectation**(*root*)

Compute the expectation values of a SPN w.r.t. each of the random variables.

Parameters
root (Node) – The root of the SPN.

Returns
The expectation w.r.t. each of the random variables.

Return type
ndarray

deeprob.spn.algorithms.moments.**variance**(*root*)

Compute the variance values of a SPN w.r.t. each of the random variables.

Parameters
root (Node) – The root of the SPN.

Returns
The variance w.r.t. each of the random variables.

Return type
ndarray

deeprob.spn.algorithms.moments.**skewness**(*root*)

Compute the skewness values of a SPN w.r.t. each of the random variables.

Parameters
root (Node) – The root of the SPN.

Returns
The skewness w.r.t. each of the random variables.

Return type
ndarray

deeprob.spn.algorithms.moments.**kurtosis**(*root*)

Compute the kurtosis values of a SPN w.r.t. each of the random variables. This function returns the kurtosis based on Fisher's definition, i.e. 3.0 is subtracted from the result to give 0.0 for a normal distribution.

Parameters
root (Node) – The root of the SPN.

Returns
The kurtosis w.r.t. each of the random variables.

Return type
ndarray

deeprob.spn.algorithms.sampling module

deeprob.spn.algorithms.sampling.**sample**(*root*, *x*, *inplace=False*, *n_jobs=0*)

Sample some features from the distribution represented by the SPN.

Parameters

- **root** (Node) – The root of the SPN.
- **x** (*ndarray*) – The inputs with possible NaN values to fill with sampled values.
- **inplace** (*bool*) – Whether to make inplace assignments.
- **n_jobs** (*int*) – The number of parallel jobs. It follows the joblib's convention. Set to 0 to disable. Warning: disrupts seed determinism.

Returns
The inputs that are NaN-filled with samples from appropriate distributions.

Return type
ndarray

deeprob.spn.algorithms.sampling.**leaf_sample**(*node*, *x*)

Sample some values from the distribution leaf.

Parameters

- **node** (Leaf) – The distribution leaf node.
- **x** (*ndarray*) – The inputs with possible NaN values to fill with sampled values.

Returns
The completed samples.

Return type
ndarray

deeprob.spn.algorithms.sampling.**sum_sample**(*node*, *lls*)

Choose the sub-distribution from which sample.

Parameters

- **node** (Sum) – The sum node.
- **lls** (*ndarray*) – The log-likelihoods of the children nodes.

Returns
The index of the sub-distribution to follow.

Return type
ndarray

deeprob.spn.algorithms.structure module

deeprob.spn.algorithms.structure.**prune**(*root*, *copy=True*)

Prune (or simplify) the given SPN to a minimal and equivalent SPN.

Parameters

- **root** (Node) – The root of the SPN.
- **copy** (*bool*) – Whether to copy the SPN before pruning it.

Returns
A minimal and equivalent SPN.

Raises

- **ValueError** – If the SPN structure is not a directed acyclic graph (DAG).
- **ValueError** – If an unknown node type is found.

Return type
Node

deeprob.spn.algorithms.structure.**marginalize**(*root*, *keep_scope*, *copy=True*)

Marginalize some random variables of a SPN, obtaining the compilation of a marginal query.

Parameters

- **root** (Node) – The root of the SPN to marginalize.
- **keep_scope** (*List[int]*) – The scope of the random variables to keep. All the other random variables will be marginalized.
- **copy** (*bool*) – Whether to copy the SPN before marginalizing it.

Returns
A SPN in which an EVI query is equivalent to a MAR query under the given scope.

Raises

- **ValueError** – If the scope of the random variables to keep is not valid.
- **ValueError** – If the SPN structure is not a directed acyclic graph (DAG).
- **ValueError** – If an unknown node type is found.
- **NotImplementedError** – If non-BinaryCLT multivariate leaves are found.

Return type
Node

Module contents

deeprob.spn.layers package

Submodules

deeprob.spn.layers.dgcspn module

class deeprob.spn.layers.dgcspn.**SpatialGaussianLayer**(*in_features*, *out_channels*, *optimize_scale=False*, *dropout=None*, *quantiles_loc=None*, *uniform_loc=None*)

Bases: Module

Initialize a Spatial Gaussian input layer.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The number of input features.
- **out_channels** (*int*) – The number of output channels.
- **optimize_scale** (*bool*) – Whether to optimize scale.
- **dropout** (*Optional[float]*) – The leaf nodes dropout rate. It can be None.
- **quantiles_loc** (*Optional[ndarray]*) – The mean quantiles for location initialization. It can be None.
- **uniform_loc** (*Optional[Tuple[float, float]]*) – The uniform range for location initialization. It can be None.

Raises

ValueError – If both quantiles_loc and uniform_loc are specified.

property **in_channels**: int

property **in_height**: int

property **in_width**: int

property **out_channels**: int

property **out_height**: int

property **out_width**: int

forward(*x*)

Evaluate the layer given some inputs.

Parameters

x (*Tensor*) – The inputs.

Returns

The outputs.

Return type

Tensor

training: bool

class deeprob.spn.layers.dgcspn.**SpatialProductLayer**(*in_features*, *kernel_size*, *padding*, *stride*, *dilation*, *depthwise=True*)

Bases: Module

Initialize a Spatial Product layer.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The number of input features.
- **kernel_size** (*Union[int, Tuple[int, int]]*) – The size of the kernels.
- **stride** (*Union[int, Tuple[int, int]]*) – The strides to use.
- **padding** (*str*) – The padding mode to use. It can be 'valid', 'full' or 'final'. Valid padding means no padding used. Full padding means padding is used based on effective kernel size. Final padding means one-side padding (
- **dilation** (*Union[int, Tuple[int, int]]*) – The space between the kernel points.
- **depthwise** (*bool*) – Whether to use depthwise convolutions. If False, random sparse kernels are used.

Raises

ValueError – If a parameter is out of domain.

property **in_channels**: int

property **in_height**: int

property **in_width**: int

property **out_channels**: int

property **out_height**: int

property **out_width**: int

forward(*x*)

Evaluate the layer given some inputs.

Parameters

x (*Tensor*) – The inputs.

Returns

The outputs.

Return type

Tensor

training: bool

class deeprob.spn.layers.dgcspn.**SpatialSumLayer**(*in_features*, *out_channels*, *dropout=None*)

Bases: Module

Initialize a Spatial Sum layer.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The number of input features.
- **out_channels** (*int*) – The number of output channels.
- **dropout** (*Optional[float]*) – The input nodes dropout rate. It can be None.

property in_channels: int

property in_height: int

property in_width: int

property out_channels: int

property out_height: int

property out_width: int

forward(*x*)

Evaluate the layer given some inputs.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

training: bool

class deeprob.spn.layers.dgcspn.SpatialRootLayer(*in_features*, *out_channels*)

Bases: Module

Initialize a Spatial Root layer.

Parameters

- **in_features** (*Tuple[int, int, int]*) – The number of input features.
- **out_channels** (*int*) – The number of output channels.

training: bool

property in_channels: int

property in_height: int

property in_width: int

forward(*x*)

Evaluate the layer given some inputs.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

deeprob.spn.layers.ratspn module

class deeprob.spn.layers.ratspn.**RegionGraphLayer**(*in_features*, *out_channels*, *regions*, *rg_depth*, *dropout=None*, ***kwargs*)

Bases: ABC, Module

Initialize a Region Graph-based base distribution.

Parameters

- **in_features** (*int*) – The number of input features.
- **out_channels** (*int*) – The number of channels for each base distribution layer.
- **regions** (*List[tuple]*) – The regions of the distributions.
- **rg_depth** (*int*) – The depth of the region graph.
- **dropout** (*Optional[float]*) – The leaf nodes dropout rate. It can be None.

unpad_samples(*x*, *idx_group*)

Reorder and unpad some samples.

Parameters

- **x** (*Tensor*) – The samples.
- **idx_group** (*Tensor*) – The group indices.

Returns
The reordered samples with padding dummy variables removed.

Return type
Tensor

forward(*x*)

Execute the layer on some inputs.

Parameters
x (*Tensor*) – The inputs.

Returns
The log-likelihoods of each distribution leaf.

Return type
Tensor

abstract distribution_mode()

Get the mode of the distribution.

Returns
The mode of the distribution.

Return type
Tensor

mpe(*x*, *idx_group*, *idx_offset*)

Evaluate the layer given some inputs for maximum at posteriori estimation.

Parameters

- **x** (*Tensor*) – The inputs. Random variables can be marginalized using NaN values.
- **idx_group** (*Tensor*) – The group indices.

- **idx_offset** (*Tensor*) – The offset indices.

Returns
The samples having maximum at posteriori estimates on marginalized random variables.

Return type
Tensor

sample(*idx_group*, *idx_offset*)

Sample from a base distribution.

Parameters

- **idx_group** (*Tensor*) – The group indices.
- **idx_offset** (*Tensor*) – The offset indices.

Returns
The computed samples.

Return type
Tensor

training: bool

class deeprob.spn.layers.ratspn.**GaussianLayer**(*in_features*, *out_channels*, *regions*, *rg_depth*, *dropout=None*, *uniform_loc=None*, *optimize_scale=False*)

Bases: *RegionGraphLayer*

Initialize a Gaussian distributions input layer.

Parameters

- **in_features** (*int*) – The number of input features.
- **out_channels** (*int*) – The number of channels for each base distribution layer.
- **regions** (*List[tuple]*) – The regions of the distributions.
- **rg_depth** (*int*) – The depth of the region graph.
- **dropout** (*Optional[float]*) – The leaf nodes dropout rate. It can be None.
- **uniform_loc** (*Optional[Tuple[float, float]]*) – The optional uniform distribution parameters for location initialization.
- **optimize_scale** (*bool*) – Whether to optimize scale and location jointly.

distribution_mode()

Get the mode of the distribution.

Returns
The mode of the distribution.

Return type
Tensor

training: bool

class deeprob.spn.layers.ratspn.**BernoulliLayer**(*in_features*, *out_channels*, *regions*, *rg_depth*, *dropout=None*)

Bases: *RegionGraphLayer*

Initialize a Bernoulli distributions input layer.

Parameters

- **in_features** (*int*) – The number of input features.
- **out_channels** (*int*) – The number of channels for each base distribution layer.
- **regions** (*List[tuple]*) – The regions of the distributions.
- **rg_depth** (*int*) – The depth of the region graph.
- **dropout** (*Optional[float]*) – The leaf nodes dropout rate. It can be None.

distribution_mode()

Get the mode of the distribution.

Returns
The mode of the distribution.

Return type
Tensor

training: bool

class deeprob.spn.layers.ratspn.**ProductLayer**(*in_regions*, *in_nodes*)

Bases: Module

Initialize the Product layer.

Parameters

- **in_regions** (*int*) – The number of input regions.
- **in_nodes** (*int*) – The number of input nodes per region.

forward(*x*)

Evaluate the layer given some inputs.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

mpe(*x*, *idx_group*, *idx_offset*)

Evaluate the layer given some inputs for maximum at posteriori estimation.

Parameters

- **x** (*Tensor*) – The inputs (not used here).
- **idx_group** (*Tensor*) – The group indices.
- **idx_offset** (*Tensor*) – The offset indices.

Returns
The group and offset indices.

Return type
[<class 'torch.Tensor'>, <class 'torch.Tensor'>]

sample(*idx_group*, *idx_offset*)

Sample from a product layer.

Parameters

- **idx_group** (*Tensor*) – The group indices.
- **idx_offset** (*Tensor*) – The offset indices.

Returns
The group and offset indices.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

class deeprob.spn.layers.ratspn.**SumLayer**(*in_partitions*, *in_nodes*, *out_nodes*, *dropout=None*)

Bases: Module

Initialize the sum layer.

Parameters

- **in_partitions** (*int*) – The number of input partitions.
- **in_nodes** (*int*) – The number of input nodes per partition.
- **out_nodes** (*int*) – The number of output nodes per region.
- **dropout** (*Optional[float]*) – The input nodes dropout rate. It can be None.

forward(*x*)

Evaluate the layer given some inputs.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

mpe(*x*, *idx_group*, *idx_offset*)

Evaluate the layer given some inputs for maximum at posteriori estimation.

Parameters

- **x** (*Tensor*) – The inputs.
- **idx_group** (*Tensor*) – The group indices.
- **idx_offset** (*Tensor*) – The offset indices.

Returns
The group and offset indices.

Return type
[<class 'torch.Tensor'>, <class 'torch.Tensor'>]

sample(*idx_group*, *idx_offset*)

Sample from a sum layer.

Parameters

- **idx_group** (*Tensor*) – The group indices.
- **idx_offset** (*Tensor*) – The offset indices.

Returns
The group and offset indices.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

class deeprob.spn.layers.ratspn.**RootLayer**(*in_partitions*, *in_nodes*, *out_classes*)

Bases: Module

Initialize the root layer.

Parameters

- **in_partitions** (*int*) – The number of input partitions.
- **in_nodes** (*int*) – The number of input nodes per partition.
- **out_classes** (*int*) – The number of output nodes.

forward(*x*)

Evaluate the layer given some inputs.

Parameters
x – The inputs.

Returns
The outputs.

mpe(*x*, *y*)

Evaluate the layer given some inputs for maximum at posteriori estimation.

Parameters

- **x** (*Tensor*) – The inputs.
- **y** (*Tensor*) – The target classes.

Returns
The group and offset indices.

Return type
Tuple[*Tensor*, *Tensor*]

training: bool

sample(*y*)

Sample from the root layer.

Parameters
y (*Tensor*) – The target classes.

Returns
The group and offset indices.

Return type
Tuple[*Tensor*, *Tensor*]

Module contents

deeprob.spn.learning package

Subpackages

deeprob.spn.learning.splitting package

Submodules

deeprob.spn.learning.splitting.cluster module

deeprob.spn.learning.splitting.cluster.**gmm**(*data*, *distributions*, *domains*, *random_state*, *n=2*)

Execute GMM clustering on some data.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **n** (*int*) – The number of clusters.

Returns

An array where each element is the cluster where the corresponding data belong.

Return type

ndarray

deeprob.spn.learning.splitting.cluster.**kmeans**(*data*, *distributions*, *domains*, *random_state*, *n=2*)

Execute K-Means clustering on some data.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **n** (*int*) – The number of clusters.

Returns

An array where each element is the cluster where the corresponding data belong.

Return type

ndarray

deeprob.spn.learning.splitting.cluster.**kmeans_mb**(*data*, *distributions*, *domains*, *random_state*, *n=2*)

Execute MiniBatch K-Means clustering on some data.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.

- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **n** (*int*) – The number of clusters.

Returns
An array where each element is the cluster where the corresponding data belong.

Return type
ndarray

deeprob.spn.learning.splitting.cluster.**dbscan**(*data*, *distributions*, *domains*, *random_state*, *n=2*)

Execute DBSCAN clustering on some data (only on discrete data).

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **n** (*int*) – The number of clusters.

Returns
An array where each element is the cluster where the corresponding data belong.

Raises
ValueError – If the leaf distributions are NOT discrete.

Return type
ndarray

deeprob.spn.learning.splitting.cluster.**wald**(*data*, *distributions*, *domains*, *random_state*, *n=2*)

Execute Ward (Hierarchical) clustering on some data (only discrete data).

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **n** (*int*) – The number of clusters.

Returns
An array where each element is the cluster where the corresponding data belong.

Raises
ValueError – If the leaf distributions are NOT discrete.

Return type
ndarray

deeprob.spn.learning.splitting.cols module

deeprob.spn.learning.splitting.cols.**SplitColsFunc**

A signature for a columns splitting function.

alias of Callable[[ndarray, List[Type[Leaf]], List[Union[list, tuple]], RandomState, Any], ndarray]

deeprob.spn.learning.splitting.cols.**split_cols_clusters**(*data*, *clusters*, *scope*)

Split the data vertically given the clusters.

Parameters

- **data** (*ndarray*) – The data.
- **clusters** (*ndarray*) – The clusters.
- **scope** (*List[int]*) – The original scope.

Returns
(slices, scopes) where slices is a list of partial data and scopes is a list of partial scopes.

Return type
Tuple[*List*[*ndarray*], *List*[*List*[int]]]

deeprob.spn.learning.splitting.cols.**get_split_cols_method**(*split_cols*)

Get the columns splitting method given a string.

Parameters
split_cols (*str*) – The string of the method do get.

Returns
The corresponding columns splitting function.

Raises
ValueError – If the columns splitting method is unknown.

Return type
Callable[[*ndarray*, *List*[*Type*[Leaf]], *List*[*Union*[list, tuple]], *RandomState*, *Any*], *ndarray*]

deeprob.spn.learning.splitting.entropy module

deeprob.spn.learning.splitting.entropy.**entropy_cols**(*data*, *distributions*, *domains*, *random_state*, *e=0.3*, *alpha=0.1*)

Entropy based column splitting method.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – Distributions of the features.
- **domains** (*List[Union[list, tuple]]*) – Range of values of the features.
- **e** (*float*) – Threshold of the considered entropy to be signficant.
- **alpha** (*float*) – laplacian alpha to apply at frequence.
- **random_state** (*RandomState*) –

Returns
A partitioning of features.

Return type
ndarray

deeprob.spn.learning.splitting.entropy.**entropy_adaptive_cols**(*data*, *distributions*, *domains*, *random_state*, *e=0.3*, *alpha=0.1*, *size=None*)

Adaptive Entropy based column splitting method.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – Distributions of the features.
- **domains** (*List[Union[list, tuple]]*) – Range of values of the features.
- **e** (*float*) – Threshold of the considered entropy to be signficant.
- **alpha** (*float*) – laplacian alpha to apply at frequence.
- **size** (*Optional[int]*) – Size of whole dataset.
- **random_state** (*RandomState*) –

Returns
A partitioning of features.

Raises
ValueError – If the size of the data is missing.

Return type
ndarray

deeprob.spn.learning.splitting.gini module

deeprob.spn.learning.splitting.gini.**gini_cols**(*data*, *distributions*, *domains*, *random_state*, *e=0.3*, *alpha=0.1*)

Gini index column splitting method.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – Distributions of the features.
- **domains** (*List[Union[list, tuple]]*) – Range of values of the features.
- **e** (*float*) – Threshold of the considered entropy to be signficant.
- **alpha** (*float*) – laplacian alpha to apply at frequence.
- **random_state** (*RandomState*) –

Returns
A partitioning of features.

Return type
ndarray

deeprob.spn.learning.splitting.gini.**gini_adaptive_cols**(*data*, *distributions*, *domains*, *random_state*, *e=0.3*, *alpha=0.1*, *size=None*)

Adaptive Gini index column splitting method.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – Distributions of the features.
- **domains** (*List[Union[list, tuple]]*) – Range of values of the features.
- **e** (*float*) – Threshold of the considered entropy to be signficant.
- **alpha** (*float*) – laplacian alpha to apply at frequence.
- **size** (*Optional[int]*) – Size of whole dataset.
- **random_state** (*RandomState*) –

Returns
A partitioning of features.

Raises
ValueError – If the size of the data is missing.

Return type
ndarray

deeprob.spn.learning.splitting.gvs module

deeprob.spn.learning.splitting.gvs.**gvs_cols**(*data*, *distributions*, *domains*, *random_state*, *p=5.0*)
Greedy Variable Splitting (GVS) independence test.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The distributions.
- **domains** (*List[Union[list, tuple]]*) – The domains.
- **random_state** (*RandomState*) – The random state.
- **p** (*float*) – The threshold for the G-Test.

Returns
A partitioning of features.

Raises
ValueError – If the leaf distributions are discrete and continuous.

Return type
ndarray

deeprob.spn.learning.splitting.gvs.**rgvs_cols**(*data*, *distributions*, *domains*, *random_state*, *p=5.0*)
Random Greedy Variable Splitting (RGVS) independence test.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The distributions.
- **domains** (*List[Union[list, tuple]]*) – The domains.
- **random_state** (*RandomState*) – The random state.
- **p** (*float*) – The threshold for the G-Test.

Returns
A partitioning of features.

Raises
ValueError – If the leaf distributions are discrete and continuous.

Return type
ndarray

deeprob.spn.learning.splitting.gvs.**wrgvs_cols**(*data*, *distributions*, *domains*, *random_state*, *p=5.0*)

Wiser Random Greedy Variable Splitting (WRGVS) independence test.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The distributions.
- **domains** (*List[Union[list, tuple]]*) – The domains.
- **random_state** (*RandomState*) – The random state.
- **p** (*float*) – The threshold for the G-Test.

Returns
A partitioning of features.

Raises
ValueError – If the leaf distributions are discrete and continuous.

Return type
ndarray

deeprob.spn.learning.splitting.gvs.**gtest**(*data*, *i*, *j*, *distributions*, *domains*, *p=5.0*, *test=True*)

The G-Test independence test between two features.

Parameters

- **data** (*ndarray*) – The data.
- **i** (*int*) – The index of the first feature.
- **j** (*int*) – The index of the second feature.
- **distributions** (*List[Type[Leaf]]*) – The distributions.
- **domains** (*List[Union[list, tuple]]*) – The domains.
- **p** (*float*) – The threshold for the G-Test.
- **test** (*bool*) – If the method is called as test (true) or as value of statistics (false), default True.

Returns
False if the features are assumed to be dependent, True otherwise.

Raises
ValueError – If the leaf distributions are discrete and continuous.

Return type
Union[bool, float]

deeprob.spn.learning.splitting.random module

deeprob.spn.learning.splitting.random.**random_rows**(*data*, *distributions*, *domains*, *random_state*, *a=2.0*, *b=2.0*)

Choose a binary partition horizontally randomly. The proportion of the split is sampled from a beta distribution.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions (not used).
- **domains** (*List[Union[list, tuple]]*) – The data domains (not used).
- **random_state** (*RandomState*) – The random state.
- **a** (*float*) – The alpha parameter of the beta distribution.
- **b** (*float*) – The beta parameter of the beta distribution.

Returns
A binary partition.

Return type
ndarray

deeprob.spn.learning.splitting.random.**random_cols**(*data*, *distributions*, *domains*, *random_state*, *a=2.0*, *b=2.0*)

Choose a binary partition vertically randomly. The proportion of the split is sampled from a beta distribution.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions (not used).
- **domains** (*List[Union[list, tuple]]*) – The data domains (not used).
- **random_state** (*RandomState*) – The random state.
- **a** (*float*) – The alpha parameter of the beta distribution.
- **b** (*float*) – The beta parameter of the beta distribution.

Returns
A binary partition.

Return type
ndarray

deeprob.spn.learning.splitting.rdc module

deeprob.spn.learning.splitting.rdc.**rdc_cols**(*data*, *distributions*, *domains*, *random_state*, *d=0.3*, *k=20*, *s=0.16666666666666666*, *nl=<ufunc 'sin'>*)

Split the features using the RDC (Randomized Dependency Coefficient) method.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.

- **random_state** (*RandomState*) – The random state.
- **d** (*float*) – The threshold value that regulates the independence tests among the features.
- **k** (*int*) – The size of the latent space.
- **s** (*float*) – The standard deviation of the gaussian distribution.
- **nl** (*Callable[[ndarray], ndarray]*) – The non linear function to use.

Returns
A features partitioning.

Return type
ndarray

deeprob.spn.learning.splitting.rdc.**rdc_rows**(*data*, *distributions*, *domains*, *random_state*, *n=2*, *k=20*, *s=0.16666666666666666*, *nl=<ufunc 'sin'>*)

Split the samples using the RDC (Randomized Dependency Coefficient) method.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **n** (*int*) – The number of clusters for KMeans.
- **k** (*int*) – The size of the latent space.
- **s** (*float*) – The standard deviation of the gaussian distribution.
- **nl** (*Callable[[ndarray], ndarray]*) – The non linear function to use.

Returns
A samples partitioning.

Return type
ndarray

deeprob.spn.learning.splitting.rdc.**rdc_scores**(*data*, *distributions*, *domains*, *random_state*, *k=20*, *s=0.16666666666666666*, *nl=<ufunc 'sin'>*)

Compute the RDC (Randomized Dependency Coefficient) score for each pair of features.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **k** (*int*) – The size of the latent space.
- **s** (*float*) – The standard deviation of the gaussian distribution.
- **nl** (*Callable[[ndarray], ndarray]*) – The non linear function to use.

Returns
The RDC score matrix.

Return type
ndarray

deeprob.spn.learning.splitting.rdc.**rdc_cca**(*i*, *j*, *features*)

Compute the RDC (Randomized Dependency Coefficient) using CCA (Canonical Correlation Analysis).

Parameters

- **i** (*int*) – The index of the first feature.
- **j** (*int*) – The index of the second feature.
- **features** (*List[ndarray]*) – The list of the features.

Returns
The RDC coefficient (the largest canonical correlation coefficient).

Return type
float

deeprob.spn.learning.splitting.rdc.**rdc_transform**(*data*, *distributions*, *domains*, *random_state*, *k=20*, *s=0.16666666666666666*, *nl=<ufunc 'sin'>*)

Execute the RDC (Randomized Dependency Coefficient) pipeline on some data.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The data distributions.
- **domains** (*List[Union[list, tuple]]*) – The data domains.
- **random_state** (*RandomState*) – The random state.
- **k** (*int*) – The size of the latent space.
- **s** (*float*) – The standard deviation of the gaussian distribution.
- **nl** (*Callable[[ndarray], ndarray]*) – The non-linear function to use.

Returns
The transformed data.

Raises
ValueError – If an unknown distribution type is found.

Return type
List[*ndarray*]

deeprob.spn.learning.splitting.rows module

deeprob.spn.learning.splitting.rows.**SplitRowsFunc**

A signature for a rows splitting function.

alias of Callable[[ndarray, List[Type[*Leaf*]], List[Union[list, tuple]], RandomState, Any], ndarray]

deeprob.spn.learning.splitting.rows.**split_rows_clusters**(*data*, *clusters*)

Split the data horizontally given the clusters.

Parameters

- **data** (*ndarray*) – The data.
- **clusters** (*ndarray*) – The clusters.

Returns
(slices, weights) where slices is a list of partial data and weights is a list of proportions of the local data in respect to the original data.

Return type
Tuple[*List*[*ndarray*], *List*[float]]

deeprob.spn.learning.splitting.rows.**get_split_rows_method**(*split_rows*)

Get the rows splitting method given a string.

Parameters
split_rows (*str*) – The string of the method do get.

Returns
The corresponding rows splitting function.

Raises
ValueError – If the rows splitting method is unknown.

Return type
Callable[[*ndarray*, *List*[*Type*[Leaf]], *List*[*Union*[list, tuple]], *RandomState*, *Any*], *ndarray*]

Module contents

Submodules

deeprob.spn.learning.em module

deeprob.spn.learning.em.**expectation_maximization**(*root*, *data*, *num_iter=100*, *batch_perc=0.1*, *step_size=0.5*, *random_init=True*, *random_state=None*, *verbose=True*)

Learn the parameters of a SPN by batch Expectation-Maximization (EM). See https://arxiv.org/abs/1604.07243 and https://arxiv.org/abs/2004.06231 for details.

Parameters

- **root** (Node) – The spn structure.
- **data** (*ndarray*) – The data to use to learn the parameters.
- **num_iter** (*int*) – The number of iterations.
- **batch_perc** (*float*) – The percentage of data to use for each step.
- **step_size** (*float*) – The step size for batch EM.
- **random_init** (*bool*) – Whether to random initialize the weights of the SPN.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.
- **verbose** (*bool*) – Whether to enable verbose learning.

Returns
The spn with learned parameters.

Raises
ValueError – If a parameter is out of domain.

Return type
Node

deeprob.spn.learning.leaf module

deeprob.spn.learning.leaf.**LearnLeafFunc**

A signature for a learn SPN leaf function.

alias of Callable[[ndarray, List[Type[*Leaf*]], List[Union[list, tuple]], List[int], Any], *Node*]

deeprob.spn.learning.leaf.**get_learn_leaf_method**(*learn_leaf*)

Get the learn leaf method.

Parameters
learn_leaf (*str*) – The learn leaf method string to use.

Returns
A learn leaf function.

Raises
ValueError – If the leaf learning method is unknown.

Return type
Callable[[*ndarray*, *List*[*Type*[Leaf]], *List*[*Union*[list, tuple]], *List*[int], *Any*], Node]

deeprob.spn.learning.leaf.**learn_mle**(*data*, *distributions*, *domains*, *scope*, *alpha=0.1*, *random_state=None*)

Learn a leaf using Maximum Likelihood Estimate (MLE). If the data is multivariate, a naive factorized model is learned.

Parameters

- **data** (*ndarray*) – The data, where each column correspond to a random variable.
- **distributions** (*List[Type[Leaf]]*) – The distributions of the random variables.
- **domains** (*List[Union[list, tuple]]*) – The domains of the random variables.
- **scope** (*List[int]*) – The scope of the leaf.
- **alpha** (*float*) – Laplace smoothing factor.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be None.

Returns
A leaf distribution.

Raises
ValueError – If there are inconsistencies between the data, distributions and domains.

Return type
Node

deeprob.spn.learning.leaf.**learn_isotonic**(*data*, *distributions*, *domains*, *scope*, *alpha=0.1*, *random_state=None*)

Learn a leaf using Isotonic method. If the data is multivariate, a naive factorized model is learned.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The distribution of the random variables.
- **domains** (*List[Union[list, tuple]]*) – The domain of the random variables.
- **scope** (*List[int]*) – The scope of the leaf.
- **alpha** (*float*) – Laplace smoothing factor.

- **random_state** (*Optional[Union[int, RandomState]]*) – The random sate. It can be None.

Returns
A leaf distribution.

Raises
ValueError – If there are inconsistencies between the data, distributions and domains.

Return type
Node

deeprob.spn.learning.leaf.**learn_binary_clt**(*data*, *distributions*, *domains*, *scope*, *to_pc=False*, *alpha=0.1*, *random_state=None*)

Learn a leaf using a Binary Chow-Liu Tree (CLT). If the data is univariate, a Maximum Likelihood Estimate (MLE) leaf is returned.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The distributions of the random variables.
- **domains** (*List[Union[list, tuple]]*) – The domains of the random variables.
- **scope** (*List[int]*) – The scope of the leaf.
- **to_pc** (*bool*) – Whether to convert the CLT into an equivalent PC.
- **alpha** (*float*) – Laplace smoothing factor.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be None.

Returns
A leaf distribution.

Raises

- **ValueError** – If there are inconsistencies between the data, distributions and domains.
- **ValueError** – If the data doesn't follow a Bernoulli distribution.

Return type
Node

deeprob.spn.learning.leaf.**learn_naive_factorization**(*data*, *distributions*, *domains*, *scope*, *learn_leaf_func*, ***learn_leaf_kwargs*)

Learn a leaf as a naive factorized model.

Parameters

- **data** (*ndarray*) – The data.
- **distributions** (*List[Type[Leaf]]*) – The distribution of the random variables.
- **domains** (*List[Union[list, tuple]]*) – The domain of the random variables.
- **scope** (*List[int]*) – The scope of the leaf.
- **learn_leaf_func** (*Callable[[ndarray, List[Type[Leaf]], List[Union[list, tuple]], List[int], Any], Node]*) – The function to use to learn the sub-distributions parameters.
- **learn_leaf_kwargs** – Additional parameters for learn_leaf_func.

Returns
A naive factorized model.

Raises
`ValueError` – If there are inconsistencies between the data, distributions and domains.

Return type
Node

deeprob.spn.learning.learnspn module

class deeprob.spn.learning.learnspn.**OperationKind**(*value*)

Bases: Enum

Operation kind used by LearnSPN algorithm.

REM_FEATURES = 1

CREATE_LEAF = 2

SPLIT_NAIVE = 3

SPLIT_ROWS = 4

SPLIT_COLS = 5

class deeprob.spn.learning.learnspn.**Task**(*parent*, *data*, *scope*, *no_cols_split=False*, *no_rows_split=False*, *is_first=False*)

Bases: tuple

Create new instance of Task(parent, data, scope, no_cols_split, no_rows_split, is_first)

Parameters

- **parent** (Node) –
- **data** (*ndarray*) –
- **scope** (*List[int]*) –
- **no_cols_split** (*bool*) –
- **no_rows_split** (*bool*) –
- **is_first** (*bool*) –

parent: *Node*
Alias for field number 0

data: ndarray
Alias for field number 1

scope: List[int]
Alias for field number 2

no_cols_split: bool
Alias for field number 3

no_rows_split: bool
Alias for field number 4

is_first: bool

Alias for field number 5

deeprob.spn.learning.learnspn.**learn_spn**(*data*, *distributions*, *domains*, *learn_leaf='mle'*, *split_rows='kmeans'*, *split_cols='rdc'*, *learn_leaf_kwargs=None*, *split_rows_kwargs=None*, *split_cols_kwargs=None*, *min_rows_slice=256*, *min_cols_slice=2*, *random_state=None*, *verbose=True*)

Learn the structure and parameters of a SPN given some training data and several hyperparameters.

Parameters

- **data** (*ndarray*) – The training data.
- **distributions** (*List[Type[Leaf]]*) – A list of distribution classes (one for each feature).
- **domains** (*List[Union[list, tuple]]*) – A list of domains (one for each feature). Each domain is either a list of values, for discrete distributions, or a tuple (consisting of min value and max value), for continuous distributions.
- **learn_leaf** (*Union[str, Callable[[ndarray, List[Type[Leaf]], List[Union[list, tuple]], List[int], Any], Node]]*) – The method to use to learn a distribution leaf node, It can be either 'mle', 'isotonic', 'binary-clt' or a custom LearnLeafFunc.
- **split_rows** (*Union[str, Callable[[ndarray, List[Type[Leaf]], List[Union[list, tuple]], RandomState, Any], ndarray]]*) – The rows splitting method. It can be either 'kmeans', 'gmm', 'rdc', 'random' or a custom SplitRowsFunc function.
- **split_cols** (*Union[str, Callable[[ndarray, List[Type[Leaf]], List[Union[list, tuple]], RandomState, Any], ndarray]]*) – The columns splitting method. It can be either 'gvs', 'rgvs', 'wrgvs', 'ebvs', 'ebvs_ae', 'gbvs', 'gbvs_ag', 'rdc', 'random' or a custom SplitColsFunc function.
- **learn_leaf_kwargs** (*Optional[dict]*) – The parameters of the learn leaf method.
- **split_rows_kwargs** (*Optional[dict]*) – The parameters of the rows splitting method.
- **split_cols_kwargs** (*Optional[dict]*) – The parameters of the cols splitting method.
- **min_rows_slice** (*int*) – The minimum number of samples required to split horizontally.
- **min_cols_slice** (*int*) – The minimum number of features required to split vertically.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns

A learned valid SPN.

Raises

ValueError – If a parameter is out of scope.

Return type

Node

deeprob.spn.learning.wrappers module

deeprob.spn.learning.wrappers.**learn_estimator**(*data*, *distributions*, *domains=None*, *method='learnspn'*, ***kwargs*)

Learn a SPN density estimator given some training data, the features distributions and domains.

Parameters

- **data** (*ndarray*) – The training data.
- **distributions** (*List[Type[Leaf]]*) – A list of distribution classes (one for each feature).
- **domains** (*Optional[List[Union[list, tuple]]]*) – A list of domains (one for each feature). Each domain is either a list of values, for discrete distributions, or a tuple (consisting of min value and max value), for continuous distributions. If None, domains are determined automatically.
- **method** (*str*) – The method used for structure learning. It can be either 'learnspn', 'xpc' or 'ensemble-xpc'.
- **kwargs** – Additional parameters for structure learning.

Returns

A learned valid and optimized SPN.

Raises

- **ValueError** – If the method used for structure learning is not known.
- **ValueError** – If the method is 'xpc' or 'ensemble-xpc' but the variable domains are not binary.

Return type

Node

deeprob.spn.learning.wrappers.**learn_classifier**(*data*, *distributions*, *domains=None*, *class_idx=- 1*, *verbose=True*, ***kwargs*)

Learn a SPN classifier given some training data, the features distributions and domains and the class index in the training data.

Parameters

- **data** (*ndarray*) – The training data.
- **distributions** (*List[Type[Leaf]]*) – A list of distribution classes (one for each feature).
- **domains** (*Optional[List[Union[list, tuple]]]*) – A list of domains (one for each feature). Each domain is either a list of values, for discrete distributions, or a tuple (consisting of min value and max value), for continuous distributions. If None, domains are determined automatically.
- **class_idx** (*int*) – The index of the class feature in the training data.
- **verbose** (*bool*) – Whether to enable verbose mode.
- **kwargs** – Other parameters for structure learning.

Returns

A learned valid and optimized SPN.

Return type
Node

deeprob.spn.learning.wrappers.**compute_data_domains**(*data*, *distributions*)

Compute the domains based on the training data and the features distributions.

Parameters

- **data** (*ndarray*) – The training data.
- **distributions** (*List[Type[Leaf]]*) – A list of distribution classes.

Returns
A list of domains. Each domain is either a list of values, for discrete distributions, or a tuple (consisting of min value and max value), for continuous distributions.

Raises
ValueError – If an unknown distribution type is found.

Return type
List[*Union*[list, tuple]]

deeprob.spn.learning.xpc module

deeprob.spn.learning.xpc.**build_disjunction**(*data*, *scope*, *assignments=None*, *alpha=0.01*)

Build a disjunction (sum node) of conjunctions (product nodes). If assignments are given, every conjunction is associated to a specific assignment (the number of conjunctions is the same as the given assignments); otherwise, every conjunction will be associated to a specific assignment occurring in the input data (the number of conjunctions is the same as the unique assignments occurring in the data).

Parameters

- **data** (*ndarray*) – The input data matrix.
- **scope** (*list*) – The scope.
- **assignments** (*Optional[ndarray]*) – The optional assignments.
- **alpha** (*float*) – Laplace smoothing factor.

Return type
Node

deeprob.spn.learning.xpc.**build_leaf**(*data*, *part*, *use_clt*, *trees_dict*, *det*, *alpha*)

Build a multivariate leaf distribution for an XPC.

Parameters

- **data** (*ndarray*) – The input data matrix.
- **part** (Partition) – The partition associated to the leaf to build.
- **use_clt** (*bool*) – True if it is possible to use CLTrees as leaf nodes, False otherwise.
- **trees_dict** (*dict*) – A dictionary of trees (see the function build_trees_dict).
- **det** (*bool*) – True to force determinism, False otherwise.
- **alpha** (*float*) – Laplace smoothing factor.

Return type
Node

deeprob.spn.learning.xpc.**greedy_vars_ordering**(*data*, *conj_len*, *alpha=0.01*)

Return the ordering of the random variables according to the implemented heuristic.

Parameters

- **data** (*ndarray*) – The input data matrix.
- **conj_len** (*int*) – The conjunction length.
- **alpha** (*float*) – Laplace smoothing factor.

Return ordering
The ordering.

Return type
list

deeprob.spn.learning.xpc.**build_trees_dict**(*data*, *cl_parts_l*, *conj_vars_l*, *alpha*, *random_state*)

Return a dictionary where:

- a key refers to a scope length
- a value is a list of two lists: the first is a list of predecessors, the second its scope.

Parameters

- **data** (*ndarray*) – The input data matrix.
- **cl_parts_l** (*list*) – List of lists. Every sublist is associated to a specific XPC and contains the leaf partitions over which a CLTree will be learnt.
- **conj_vars_l** (*list*) – List of lists. Every sublist contains the variables of a conjunction (e.g. [[3, 5]]). If a sublist occurs before another, then the former has been used first. There are no duplicates.
- **alpha** (*float*) – Laplace smoothing factor.
- **random_state** (*RandomState*) – The random state.

Return tree_dict
The dictionary.

Return type
dict

deeprob.spn.learning.xpc.**build_xpc**(*data*, *part_root*, *trees_dict*, *det*, *use_clt*, *alpha*)

Build the XPC induced by the partitions tree in a bottom up way. The building process is based on the post-order traversal exploration of the partitions tree.

Parameters

- **data** (*ndarray*) – The input data matrix.
- **part_root** (*Partition*) – The root partition of the tree.
- **trees_dict** (*dict*) – None if no dependency tree has to be respected, a dictionary of trees otherwise.
- **det** (*bool*) – True to force determinism, False otherwise.
- **use_clt** (*bool*) – True to use CLTrees as leaf nodes, False otherwise.
- **alpha** (*float*) – Laplace smoothing factor.

Returns
the XPC induced by the partition tree

Return type
Node

deeprob.spn.learning.xpc.**learn_xpc**(*data*, *det*, *sd*, *min_part_inst*, *conj_len*, *arity*, *n_max_parts=200*, *use_clt=True*, *use_greedy_ordering=False*, *alpha=0.01*, *random_seed=42*)

Learn an eXtremely randomized Probabilistic Circuit (XPC).

Parameters

- **data** (*ndarray*) – The input data matrix.
- **det** (*bool*) – True to force determinism, False otherwise.
- **sd** (*bool*) – True to force structured decomposability, False otherwise.
- **min_part_inst** (*int*) – The minimum number of instances allowed per partition.
- **conj_len** (*int*) – The conjunction length.
- **arity** (*int*) – The maximum number of children for a sum node.
- **n_max_parts** (*int*) – The maximum number of partitions for the partitions tree.
- **use_clt** (*bool*) – True to use CLTrees as multivariate leaves, False otherwise.
- **use_greedy_ordering** (*Optional[bool]*) – True to use a greedy ordering, False otherwise.
- **alpha** (*int*) – Laplace smoothing factor.
- **random_seed** (*int*) – Random State.

Return type
Tuple[Node, dict]

deeprob.spn.learning.xpc.**learn_expc**(*data*, *ensemble_dim*, *det*, *sd_level*, *min_part_inst*, *conj_len*, *arity*, *n_max_parts=200*, *use_clt=True*, *alpha=0.01*, *random_seed=42*)

Learn an Ensemble (i.e. a mixture) of eXtremely randomized Probabilistic Circuit (EXPC).

Parameters

- **data** (*ndarray*) – The input data matrix.
- **ensemble_dim** (*int*) – The number of circuits in the ensemble/mixture.
- **det** (*bool*) – True to force determinism, False otherwise.
- **sd_level** (*int*) – 0 a non-SD ensemble of non-SD PCs, 1 for a non-SD ensemble of SD PCs and 2 for a SD ensemble.
- **min_part_inst** (*int*) – The minimum number of instances allowed per partition.
- **conj_len** (*int*) – The conjunction length.
- **arity** (*int*) – The maximum number of children for a Sum node.
- **n_max_parts** (*int*) – The maximum number of partitions for the partitions tree.
- **use_clt** (*bool*) – True to use CLTrees as multivariate leaves, False otherwise.
- **alpha** (*int*) – Laplace smoothing factor.
- **random_seed** (*int*) – A random seed.

Return type
Tuple[Node, list]

Module contents

deeprob.spn.models package

Submodules

deeprob.spn.models.dgcspn module

class deeprob.spn.models.dgcspn.**DgcSpn**(*in_features*, *out_classes=1*, *n_batch=8*, *sum_channels=8*, *depthwise=False*, *n_pooling=0*, *optimize_scale=False*, *in_dropout=None*, *sum_dropout=None*, *quantiles_loc=None*, *uniform_loc=None*)

Bases: *ProbabilisticModel*

Initialize a Deep Generalized Convolutional Sum-Product Network (DGC-SPN).

Parameters

- **in_features** (*Tuple[int, int, int]*) – The input size as a (C, D, D) tuple.
- **out_classes** (*int*) – The number of output classes. Specify 1 in case of plain density estimation.
- **n_batch** (*int*) – The number of output channels of the base layer.
- **sum_channels** (*int*) – The number of output channels of spatial sum layers.
- **depthwise** (*Union[bool, List[bool]]*) – Whether to use depthwise convolutions as product layers at each depth level. The last flag of the list will be considered as the one for the rest of the network. If a single boolean is passed, it will be used for all the network's product layers.
- **n_pooling** (*int*) – The number of initial pooling spatial product layers.
- **optimize_scale** (*bool*) – Whether to train scale and location jointly.
- **in_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at distributions layer outputs. It can be None.
- **sum_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at sum layers. It can be None.
- **quantiles_loc** (*Optional[ndarray]*) – The mean quantiles for location initialization. It can be None.
- **uniform_loc** (*Optional[Tuple[float, float]]*) – The uniform range for location initialization. It can be None.

Raises
ValueError – If a parameter is out of domain.

forward(*x*)

Compute the log-likelihood given some evidence. Random variables can be marginalized using NaN values.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

`mpe`(*x*)

Compute the maximum at posteriori estimation. Random variables can be marginalized using NaN values.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

`sample`(*n_samples*, *y=None*)

Sample some values from the modeled distribution.

Parameters

- **n_samples** (*int*) – The number of samples.
- **y** (*Optional[Tensor]*) – The samples labels. It can be None.

Returns
The samples.

Return type
Tensor

`loss`(*x*, *y=None*)

Compute the loss of the model.

Parameters

- **x** (*Tensor*) – The outputs of the model.
- **y** (*Optional[Tensor]*) – The ground-truth. It can be None.

Returns
The loss.

Return type
Tensor

`apply_constraints`()

Apply the constraints specified by the model.

`training: bool`

deeprob.spn.models.ratspn module

class deeprob.spn.models.ratspn.**RatSpn**(*in_features*, *base_cls*, *base_kwargs=None*, *out_classes=1*, *rg_depth=2*, *rg_repetitions=1*, *rg_batch=2*, *rg_sum=2*, *in_dropout=None*, *sum_dropout=None*, *random_state=None*)

Bases: *ProbabilisticModel*

Initialize a RAT-SPN.

Parameters

- **in_features** (*int*) – The number of input features.
- **base_cls** (*Type[RegionGraphLayer]*) – The base distribution's class. It must be a subclass of RegionGraphLayer.
- **base_kwargs** (*Optional[dict]*) – Optional additiona parameters to pass to the base distribution's class constructor.
- **out_classes** (*int*) – The number of output classes. Specify 1 in case of plain density estimation.
- **rg_depth** (*int*) – The depth of the region graph.
- **rg_repetitions** (*int*) – The number of independent repetitions of the region graph.
- **rg_batch** (*int*) – The number of base distribution batches.
- **rg_sum** (*int*) – The number of sum nodes per region.
- **in_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at distributions layer outputs. It can be None.
- **sum_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at sum layers. It can be None.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.

Raises

ValueError – If a parameter is out of domain.

forward(*x*)

Compute the log-likelihood given some evidence. Random variables can be marginalized using NaN values.

Parameters

x (*Tensor*) – The inputs.

Returns

The outputs.

Return type

Tensor

mpe(*x*, *y=None*)

Compute the maximum at posteriori estimation. Random variables can be marginalized using NaN values.

Parameters

- **x** (*Tensor*) – The inputs tensor.
- **y** (*Optional[Tensor]*) – The target classes tensor. It can be None for unlabeled maximum at posteriori estimation.

Returns
The output of the model.

Return type
Tensor

`sample`(*n_samples*, *y=None*)

Sample some values from the modeled distribution.

Parameters

- **n_samples** (*int*) – The number of samples.
- **y** (*Optional[Tensor]*) – The samples labels. It can be None.

Returns
The samples.

Return type
Tensor

`loss`(*x*, *y=None*)

Compute the loss of the model.

Parameters

- **x** (*Tensor*) – The outputs of the model.
- **y** (*Optional[Tensor]*) – The ground-truth. It can be None.

Returns
The loss.

Return type
Tensor

`training: bool`

`class deeprob.spn.models.ratspn.GaussianRatSpn`(*in_features*, *out_classes=1*, *rg_depth=2*, *rg_repetitions=1*, *rg_batch=2*, *rg_sum=2*, *in_dropout=None*, *sum_dropout=None*, *random_state=None*, *uniform_loc=None*, *optimize_scale=False*)

Bases: *RatSpn*

Initialize a Gaussian RAT-SPN.

Parameters

- **in_features** (*int*) – The number of input features.
- **out_classes** (*int*) – The number of output classes. Specify 1 in case of plain density estimation.
- **rg_depth** (*int*) – The depth of the region graph.
- **rg_repetitions** (*int*) – The number of independent repetitions of the region graph.
- **rg_batch** (*int*) – The number of base distributions batches.
- **rg_sum** (*int*) – The number of sum nodes per region.
- **in_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at distributions layer outputs. It can be None.

- **sum_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at sum layers. It can be None.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.
- **uniform_loc** (*Optional[Tuple[float, float]]*) – The optional uniform distribution parameters for location initialization.
- **optimize_scale** (*bool*) – Whether to train scale and location jointly.

apply_constraints()

Apply the constraints specified by the model.

training: bool

class deeprob.spn.models.ratspn.**BernoulliRatSpn**(*in_features*, *out_classes=1*, *rg_depth=2*, *rg_repetitions=1*, *rg_batch=2*, *rg_sum=2*, *in_dropout=None*, *sum_dropout=None*, *random_state=None*)

Bases: *RatSpn*

Initialize a Bernoulli RAT-SPN.

Parameters

- **in_features** (*int*) – The number of input features.
- **out_classes** (*int*) – The number of output classes. Specify 1 in case of plain density estimation.
- **rg_depth** (*int*) – The depth of the region graph.
- **rg_repetitions** (*int*) – The number of independent repetitions of the region graph.
- **rg_batch** (*int*) – The number of base distributions batches.
- **rg_sum** (*int*) – The number of sum nodes per region.
- **in_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at distributions layer outputs. It can be None.
- **sum_dropout** (*Optional[float]*) – The dropout rate for probabilistic dropout at product layer outputs. It can be None.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.

training: bool

deeprob.spn.models.sklearn module

class deeprob.spn.models.sklearn.**SPNEstimator**(*distributions*, *domains=None*, ***kwargs*)

Bases: BaseEstimator, DensityMixin

Scikit-learn density estimator model for Sum Product Networks (SPNs).

Parameters

- **distributions** (*List[Type[Leaf]]*) – A list of distribution classes (one for each feature).

- **domains** (*Optional[List[Union[list, tuple]]]*) – A list of domains (one for each feature).
- **kwargs** – Additional arguments to pass to the SPN learner.

fit(*X*, *y=None*)

Fit the SPN density estimator.

Parameters

- **X** (*ndarray*) – The training data.
- **y** (*Optional[ndarray]*) – Ignored, only for scikit-learn API convention.

Returns
Itself.

predict_log_proba(*X*)

Predict using the SPN density estimator, i.e. compute the log-likelihood.

Parameters
X (*ndarray*) – The inputs. They can be marginalized using NaNs.

Returns
The log-likelihood of the inputs.

Return type
ndarray

mpe(*X*)

Predict the un-observed variable by maximum at posterior estimation (MPE).

Parameters
X (*ndarray*) – The inputs having some NaN values.

Returns
The MPE assignment to un-observed variables.

Return type
ndarray

sample(*n=None*, *X=None*)

Sample from the modeled distribution.

Parameters

- **n** (*Optional[int]*) – The number of samples. It must be None if X is not None. If None, n=1 is assumed.
- **X** (*Optional[ndarray]*) – Data used for conditional sampling. It can be None for full sampling.

Returns
The samples.

Raises
ValueError – If both parameters 'n' and 'X' are passed by.

Return type
ndarray

score(*X*, *y=None*)

Return the mean log-likelihood and two standard deviations on the given test data.

Parameters

- **X** (*ndarray*) – The inputs. They can be marginalized using NaNs.
- **y** (*Optional[ndarray]*) – Ignored. Specified only for scikit-learn API compatibility.

Returns
A dictionary consisting of two keys "mean_ll" and "stddev_ll", representing respectively the mean log-likelihood and two standard deviations.

Return type
dict

class deeprob.spn.models.sklearn.**SPNClassifier**(*distributions*, *domains=None*, ***kwargs*)

Bases: BaseEstimator, ClassifierMixin

Scikit-learn classifier model for Sum Product Networks (SPNs).

Parameters

- **distributions** (*List[Type[Leaf]]*) – A list of distribution classes (one for each feature).
- **domains** (*Optional[List[Union[list, tuple]]]*) – A list of domains (one for each feature).
- **kwargs** – Additional arguments to pass to the SPN learner.

fit(*X*, *y*)

Fit the SPN density estimator.

Parameters

- **X** (*ndarray*) – The training data.
- **y** (*ndarray*) – The data labels.

Returns
Itself.

predict(*X*)

Predict using the SPN classifier.

Parameters
X (*ndarray*) – The inputs. They can be marginalized using NaNs.

Returns
The predicted classes.

Return type
ndarray

predict_proba(*X*)

Predict using the SPN classifier, using probabilities.

Parameters
X (*ndarray*) – The inputs. They can be marginalized using NaNs.

Returns
The prediction probabilities for each class.

Return type
ndarray

predict_log_proba(*X*)

Predict using the SPN classifier, using log-probabilities.

Parameters
X (*ndarray*) – The inputs. They can be marginalized using NaNs.

Returns
The prediction log-probabilities for each class.

Return type
ndarray

sample(*n=None*, *y=None*)

Sample from the modeled conditional distribution.

Parameters

- **n** (*Optional[int]*) – The number of samples. It must be None if y is not None. If None, n=1 is assumed.
- **y** (*Optional[ndarray]*) – Labels used for conditional sampling. It can be None for unconditional sampling.

Returns
The samples.

Return type
ndarray

Module contents

deeprob.spn.structure package

Submodules

deeprob.spn.structure.cltree module

class deeprob.spn.structure.cltree.**BinaryCLT**(*scope*, *root=None*, *tree=None*, *params=None*)

Bases: *Leaf*

Initialize Binary Chow-Liu Tree (CLT) multi-variate leaf node.

Parameters

- **scope** (*List[int]*) – The scope of the leaf.
- **root** (*Optional[int]*) – The root node of the CLT. If None it will be chosen randomly.
- **tree** (*Optional[Union[List[int], np.ndarray]]*) – A sequence of variable ids predecessors (encoding the tree structure).
- **params** (*Optional[Union[List[List[List[float]]], np.ndarray]]*) – The CLT conditional probability tables (CPTs), as a (N, 2, 2) Numpy array in logarithmic scale. Note that params[i, l, k] = log P(X_i=k | Pa(X_i)=l).

Raises

- **ValueError** – If the root variable is not in scope.

- **ValueError** – If the tree structure is not compatible with the number of variables and root node.
- **ValueError** – If the CPTs parameters are invalid.

LEAF_TYPE = 1

static compute_clt_parameters(*bfs*, *tree*, *priors*, *joints*)

Compute the parameters of the CLTree given the tree structure and the priors and joints distributions.

This function returns the conditional probability tables (CPTs) in a tensorized form. Note that params[i, l, k] = P(X_i=k | Pa(X_i)=l). A special case is made for the root distribution which is not conditioned. Note that params[root, :, k] = P(X_root=k).

Parameters

- **bfs** (*ndarray*) – The bfs structure, i.e. a sequence of successors in a breadth-first traversal.
- **tree** (*ndarray*) – The tree structure, i.e. a sequence of predecessors in a tree structure.
- **priors** (*ndarray*) – The priors distributions.
- **joints** (*ndarray*) – The joints distributions.

Returns
The conditional probability tables (CPTs) in a tensorized form.

Return type
ndarray

em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

fit(*data*, *domain*, *alpha=0.1*, *random_state=None*, ***kwargs*)

Fit the distribution parameters (and structure if necessary) given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*List[list]*) – The domain of the distribution leaf.
- **alpha** (*float*) – The Laplace smoothing factor.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.
- **kwargs** – Optional parameters.

Raises

- **ValueError** – If the random state is not valid.

- **ValueError** – If a parameter is out of domain.

message_passing(*x*, *obs_mask*, *return_lls=True*, *reduce='mar'*)

Compute the messages passed from the leaves to the root node.

Parameters

- **x** (*ndarray*) – The input data.
- **obs_mask** (*ndarray*) – The mask of observed values.
- **return_lls** (*bool*) – Whether to compute and return the log-likelihoods.
- **reduce** (*str*) – The method used to reduce the messages of missing values. It can be either 'mar' (marginalize the message) or 'mpe' (maximum probable explanation).

Returns

The messages array if return_lls is False. The log-likelihoods if return_lls is True.

Return type

ndarray

likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters

x (*ndarray*) – The inputs.

Returns

The resulting likelihoods.

Return type

ndarray

log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters

x (*ndarray*) – The inputs.

Returns

The resulting log-likelihoods.

Return type

ndarray

mpe(*x*)

Compute the maximum at posteriori values.

Parameters

x (*ndarray*) – The inputs.

Returns

The distribution's maximum at posteriori values.

Return type

ndarray

sample(*x*)

Sample from the leaf distribution.

Parameters

x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

`moment(k=1)`

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
float

`params_count()`

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

Return type
int

`params_dict()`

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

Return type
dict

`to_pc()`

Convert a Chow-Liu Tree into a smooth, deterministic and structured-decomposable PC

Returns
A smooth, deterministic and structured-decomposable PC.

Return type
Node

`get_scopes()`

Return a list containing the scope of every node in the PC equivalent to the current CLTree (see to_pc() method). Every scope occurs once in the list.

Returns
The list of scopes.

deeprob.spn.structure.io module

deeprob.spn.structure.io.**save_digraph_json**(*graph*, *f*)

Save a NetworkX directed graph by using the JSON format.

Parameters

- **graph** (*DiGraph*) – The NetworkX directed graph.
- **f** (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the output JSON file.

deeprob.spn.structure.io.**load_digraph_json**(*f*)

Load a NetworkX directed graph by using the JSON format.

Parameters
f (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the input JSON file.

Returns
The NetworkX directed graph.

Return type
DiGraph

deeprob.spn.structure.io.**save_spn_json**(*root*, *f*)

Save SPN to file by using the JSON format.

Parameters

- **root** (Node) – The root node of the SPN.
- **f** (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the output JSON file.

deeprob.spn.structure.io.**load_spn_json**(*f*, *leaves=None*)

Load SPN from file by using the JSON format.

Parameters

- **f** (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the input JSON file.
- **leaves** (*Optional[List[Type[Leaf]]]*) – An optional list of custom leaf classes. Useful when dealing with user-defined leaves.

Returns
The loaded SPN with initialied ids for each node.

Raises
ValueError – If multiple custom leaf classes with the same name are defined.

Return type
Node

deeprob.spn.structure.io.**save_binary_clt_json**(*clt*, *f*)

Save Binary Chow-Liu Tree (CLT) to file by using the JSON format.

Parameters

- **clt** (BinaryCLT) – The binary CLT.
- **f** (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the output JSON file.

deeprob.spn.structure.io.**load_binary_clt_json**(*f*)

Load Binary Chow-Liu Tree (CLT) from file by using the JSON format.

Parameters
f (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the input JSON file.

Returns
The loaded binary CLT.

Return type
BinaryCLT

deeprob.spn.structure.io.**spn_to_digraph**(*root*)

Convert a SPN to a NetworkX directed graph.

Parameters
root (Node) – The root node of the SPN.

Returns
The corresponding NetworkX directed graph.

Raises
ValueError – If the SPN structure is not a directed acyclic graph (DAG).

Return type
DiGraph

deeprob.spn.structure.io.**digraph_to_spn**(*graph*, *leaf_map*)

Convert a NetworkX directed graph to a SPN.

Parameters

- **graph** (*DiGraph*) – The NetworkX directed graph.
- **leaf_map** (*Dict[str, Type[Leaf]]*) – The leaf distributions mapper dictionary.

Returns
The corresponding SPN.

Raises
ValueError – If the graph is not a directed acyclic graph (DAG).

Return type
Node

deeprob.spn.structure.io.**binary_clt_to_digraph**(*clt*)

Convert a binary Chow-Liu Tree (CLT) to a NetworkX directed graph.

Parameters
clt (BinaryCLT) – The binary CLT.

Returns
The corresponding NetworkX directed graph.

Raises
ValueError – If the CLT is not initialized.

Return type
DiGraph

deeprob.spn.structure.io.**digraph_to_binary_clt**(*graph*)

Convert a NetworkX directed graph to a binary Chow-Liu Tree (CLT).

Parameters
graph (*DiGraph*) – The NetworkX directed graph.

Returns
The corresponding Chow-Liu Tree.

Raises
ValueError – If the graph is not a tree.

Return type
BinaryCLT

deeprob.spn.structure.io.**plot_spn**(*root*, *f*)

Plot a SPN into file.

Parameters

- **root** (Node) – The SPN root node.
- **f** (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the output file.

Raises

- **ValueError** – If an unknown node type is found.
- **ValueError** – If the SPN structure is not a DAG.

deeprob.spn.structure.io.**plot_binary_clt**(*clt*, *f*, *show_weights=True*)

Plot a binary Chow-Liu Tree (CLT) into file.

Parameters

- **clt** (BinaryCLT) – The binary CLT.
- **f** (*Union[IO, PathLike, str]*) – A file-like object or a filepath of the output file.
- **show_weights** (*bool*) – Whether to show the conditional probability tables (CPTs).

deeprob.spn.structure.leaf module

class deeprob.spn.structure.leaf.**LeafType**(*value*)

Bases: Enum

The type of the distribution leaf. It can be either discrete or continuous.

DISCRETE = 1

CONTINUOUS = 2

class deeprob.spn.structure.leaf.**Leaf**(*scope*)

Bases: *Node*

Initialize a leaf node given its scope.

Parameters

- **scope** (*Union[int, List[int]]*) – The scope of the leaf.
- **kwargs** – Additional arguments.

LEAF_TYPE = None

abstract em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

abstract em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

abstract fit(*data*, *domain*, ***kwargs*)

Fit the distribution parameters given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*Union[list, tuple]*) – The domain of the distribution leaf.
- **kwargs** – Optional parameters.

Raises

ValueError – If a parameter is out of domain.

abstract likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters

x (*ndarray*) – The inputs.

Returns

The resulting likelihoods.

Return type

ndarray

abstract log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters

x (*ndarray*) – The inputs.

Returns

The resulting log-likelihoods.

Return type

ndarray

abstract mpe(*x*)

Compute the maximum at posteriori values.

Parameters

x (*ndarray*) – The inputs.

Returns

The distribution's maximum at posteriori values.

Return type

ndarray

abstract sample(*x*)

Sample from the leaf distribution.

Parameters
x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

abstract moment(*k=1*)

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
float

abstract params_count()

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

Return type
int

abstract params_dict()

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

Return type
dict

class deeprob.spn.structure.leaf.**Bernoulli**(*scope*, *p=0.5*)

Bases: *Leaf*

Initialize a Bernoulli leaf node given its scope.

Parameters

- **scope** (*int*) – The scope of the leaf.
- **p** (*float*) – The Bernoulli probability.

Raises
ValueError – If a parameter is out of domain.

LEAF_TYPE = 1

fit(*data*, *domain*, *alpha=0.1*, ***kwargs*)

Fit the distribution parameters given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*list*) – The domain of the distribution leaf.
- **alpha** (*float*) – The Laplace smoothing factor.

- **kwargs** – Optional parameters.

Raises
ValueError – If a parameter is out of domain.

em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

mpe(*x*)

Compute the maximum at posteriori values.

Parameters
x (*ndarray*) – The inputs.

Returns
The distribution's maximum at posteriori values.

Return type
ndarray

sample(*x*)

Sample from the leaf distribution.

Parameters
x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

`moment`(*k=1*)

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
float

`params_count`()

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

`params_dict`()

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

`class deeprob.spn.structure.leaf.Categorical`(*scope*, *categories=None*, *probabilities=None*)

Bases: *Leaf*

Initialize a Categorical leaf node given its scope.

Parameters

- **scope** (*int*) – The scope of the leaf.
- **categories** (*Optional[Union[List, ndarray]]*) – The possible categories.
- **probabilities** (*Optional[Union[List, ndarray]]*) – The probabilities associated to each category.

`LEAF_TYPE = 1`

`fit`(*data*, *domain*, *alpha=0.1*, ***kwargs*)

Fit the distribution parameters given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*list*) – The domain of the distribution leaf.
- **alpha** (*float*) – The Laplace smoothing factor.
- **kwargs** – Optional parameters.

Raises
ValueError – If a parameter is out of domain.

em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

Raises
ValueError – If the categories are not initialized.

em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

mpe(*x*)

Compute the maximum at posteriori values.

Parameters
x (*ndarray*) – The inputs.

Returns
The distribution's maximum at posteriori values.

Return type
ndarray

sample(*x*)

Sample from the leaf distribution.

Parameters
x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

moment(*k=1*)

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
float

params_count()

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

Return type
int

params_dict()

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

Return type
dict

class deeprob.spn.structure.leaf.**Isotonic**(*scope*, *densities=None*, *breaks=None*)

Bases: *Leaf*

Initialize a histogram-Isotonic leaf node given its scope.

Parameters

- **scope** (*int*) – The scope of the leaf.
- **densities** (*Optional[Union[List[float], ndarray]]*) – The densities. They must sum up to one.
- **breaks** (*Optional[Union[List[float], ndarray]]*) – The breaks values, such that len(breaks) == len(densities) + 1.

Raises
ValueError – If a parameter is out of domain.

LEAF_TYPE = 2

fit(*data*, *domain*, *alpha=0.1*, ***kwargs*)

Fit the distribution parameters given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*tuple*) – The domain of the distribution leaf.
- **alpha** (*float*) – The Laplace smoothing factor.
- **kwargs** – Optional parameters.

Raises
ValueError – If a parameter is out of domain.

em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

mpe(*x*)

Compute the maximum at posteriori values.

Parameters
x (*ndarray*) – The inputs.

Returns
The distribution's maximum at posteriori values.

Return type
ndarray

sample(*x*)

Sample from the leaf distribution.

Parameters
x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

`moment(k=1)`

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
ndarray

`params_count()`

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

Return type
int

`params_dict()`

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

Return type
dict

`class deeprob.spn.structure.leaf.Uniform(scope, start=0.0, width=1.0)`

Bases: *Leaf*

Initialize an Uniform leaf node given its scope.

Parameters

- **scope** (*int*) – The scope of the leaf.
- **start** (*float*) – The start of the uniform distribution.
- **width** (*float*) – The width of the uniform distribution.

`LEAF_TYPE = 2`

`fit(data, domain, **kwargs)`

Fit the distribution parameters given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*tuple*) – The domain of the distribution leaf.
- **kwargs** – Optional parameters.

Raises
ValueError – If a parameter is out of domain.

em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

mpe(*x*)

Compute the maximum at posteriori values.

Parameters
x (*ndarray*) – The inputs.

Returns
The distribution's maximum at posteriori values.

Return type
array

sample(*x*)

Sample from the leaf distribution.

Parameters
x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

moment(*k=1*)

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
float

params_count()

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

Return type
int

params_dict()

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

Return type
dict

class deeprob.spn.structure.leaf.**Gaussian**(*scope*, *mean=0.0*, *stddev=1.0*)

Bases: *Leaf*

Initialize a Gaussian leaf node given its scope.

Parameters

- **scope** (*int*) – The scope of the leaf.
- **mean** (*float*) – The mean parameter.
- **stddev** (*float*) – The standard deviation parameter.

Raises
ValueError – If a parameter is out of domain.

LEAF_TYPE = 2

fit(*data*, *domain*, ***kwargs*)

Fit the distribution parameters given the domain and some training data.

Parameters

- **data** (*ndarray*) – The training data.
- **domain** (*tuple*) – The domain of the distribution leaf.
- **kwargs** – Optional parameters.

Raises
ValueError – If a parameter is out of domain.

em_init(*random_state*)

Random initialize the leaf's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

em_step(*stats*, *data*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **data** (*ndarray*) – The data regarding random variables of the leaf.
- **step_size** (*float*) – The step size of update.

likelihood(*x*)

Compute the likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

log_likelihood(*x*)

Compute the logarithmic likelihood of the distribution leaf given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

mpe(*x*)

Compute the maximum at posteriori values.

Parameters
x (*ndarray*) – The inputs.

Returns
The distribution's maximum at posteriori values.

Return type
ndarray

sample(*x*)

Sample from the leaf distribution.

Parameters
x (*ndarray*) – The samples with possible NaN values.

Returns
The completed samples.

Return type
ndarray

moment(*k=1*)

Compute the moment of a given order.

Parameters
k (*int*) – The order of the moment.

Returns
The moment of order k.

Return type
float

params_count()

Get the number of parameters of the distribution leaf.

Returns
The number of parameters.

Return type
int

params_dict()

Get a dictionary representation of the distribution parameters.

Returns
A dictionary containing the distribution parameters.

Return type
dict

deeprob.spn.structure.node module

class deeprob.spn.structure.node.**Node**(*scope*, *children=None*)

Bases: ABC

Initialize a SPN node given the children list and its scope.

Parameters

- **scope** (*List[int]*) – The scope.
- **children** (*Optional[List[Node]]*) – A list of nodes. If None, children are initialized as an empty list.

Raises

- **ValueError** – If the scope is empty.
- **ValueError** – If the scope contains duplicates.

abstract likelihood(*x*)

Compute the likelihood of the node given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

abstract log_likelihood(*x*)

Compute the logarithmic likelihood of the node given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

class deeprob.spn.structure.node.**Sum**(*scope=None*, *children=None*, *weights=None*)

Bases: *Node*

Initialize a SPN sum node given a list of children and their weights and a scope.

Parameters

- **scope** (*Optional[List[int]]*) – The scope. If None, the scope is initialized based on children scopes.
- **children** (*Optional[List[Node]]*) – A list of nodes. If None, children are initialized as an empty list.
- **weights** (*Optional[Union[List[float], np.ndarray]]*) – The weights associated to each children node. It can be None.

Raises

- **ValueError** – If both scope and children are None.
- **ValueError** – If children nodes have different scopes.
- **ValueError** – If the length of weights and children are different.
- **ValueError** – If weights don't sum up to 1.

em_init(*random_state*)

Random initialize the node's parameters for Expectation-Maximization (EM).

Parameters
random_state (*RandomState*) – The random state.

em_step(*stats*, *step_size*)

Compute a batch Expectation-Maximization (EM) step.

Parameters

- **stats** (*ndarray*) – The sufficient statistics of each sample.
- **step_size** (*float*) – The step size of update.

likelihood(*x*)

Compute the likelihood of the node given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

`log_likelihood`(*x*)

Compute the logarithmic likelihood of the node given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

`class deeprob.spn.structure.node.Product`(*scope=None*, *children=None*)

Bases: *Node*

Initialize a product node given a list of children and its scope.

Parameters

- **scope** (*Optional[List[int]]*) – The scope. If None, the scope is initialized based on children scopes.
- **children** (*Optional[List[Node]]*) – A list of nodes. If None, children are initialized as an empty list.

Raises

- **ValueError** – If both scope and children are None.
- **ValueError** – If children nodes don't have disjointed scopes.

`likelihood`(*x*)

Compute the likelihood of the node given some input.

Parameters
x (*ndarray*) – The inputs.

Returns
The resulting likelihoods.

Return type
ndarray

`log_likelihood`(*x*)

Compute the logarithmic likelihood of the node given some input.

Parameters
x (*append*) – The inputs.

Returns
The resulting log-likelihoods.

Return type
ndarray

`deeprob.spn.structure.node.assign_ids`(*root*)

Assign the ids to the nodes of a SPN.

Parameters
root (Node) – The root of the SPN.

Returns
The same SPN with each node having modified ids.

Raises
ValueError – If the SPN structure is not a DAG.

Return type
Node

deeprob.spn.structure.node.**bfs**(*root*)

Compute the Breadth First Search (BFS) ordering for a SPN.

Parameters
root (Node) – The root of the SPN.

Returns
The BFS nodes iterator.

Return type
Iterator[Node]

deeprob.spn.structure.node.**dfs_post_order**(*root*)

Compute Depth First Search (DFS) Post-Order ordering for a SPN.

Parameters
root (Node) – The root of the SPN.

Returns
The DFS Post-Order nodes iterator.

Return type
Iterator[Node]

deeprob.spn.structure.node.**topological_order**(*root*)

Compute the Topological Ordering for a SPN, using the Kahn's Algorithm.

Parameters
root (Node) – The root of the SPN.

Returns
A list of nodes that form a topological ordering. If the SPN graph is not acyclic, it returns None.

Return type
Optional[*List*[Node]]

deeprob.spn.structure.node.**topological_order_layered**(*root*)

Compute the Topological Ordering Layered for a SPN, using the Kahn's Algorithm.

Parameters
root (Node) – The root of the SPN.

Returns
A list of layers that form a topological ordering. If the SPN graph is not acyclic, it returns None.

Return type
Optional[*List*[*List*[Node]]]

Module contents

deeprob.spn.utils package

Submodules

deeprob.spn.utils.filter module

deeprob.spn.utils.filter.**collect_nodes**(*root*)

Get all the nodes in a SPN.

Parameters
root (Node) – The root of the SPN.

Returns
A list of nodes.

Return type
List[Node]

deeprob.spn.utils.filter.**filter_nodes_by_type**(*root*, *ntype=<class 'deeprob.spn.structure.node.Node'>*)

Get the nodes of some specified types in a SPN.

Parameters

- **root** (Node) – The root of the SPN.
- **ntype** (*Union[Type[Node], Tuple[Type[Node], ...]]*) – The node type. Multiple node types can be specified as a tuple.

Returns
A list of nodes of some specific types.

Return type
List[*Union*[Node, Leaf, Sum, Product]]

deeprob.spn.utils.partitioning module

class deeprob.spn.utils.partitioning.**Partition**(*row_ids*, *col_ids*, *uncond_vars*, *parent_partition=None*, *is_naive=False*, *is_conj=False*)

Bases: object

Create a partition, i.e. an object modeling a data slice (and some of its properties) by keeping track of its indices (i.e. row_ids and col_ids).

Parameters

- **row_ids** (*list*) – The row indices of the modeled slice.
- **col_ids** (*list*) – The column indices of the modeled slice.
- **uncond_vars** (*list*) – Ordered list of variables from which the conjunction variables will be extracted to horizontally split the current partition.
- **parent_partition** (*Optional[Partition]*) – The optional parent partition
- **is_naive** (*Optional[bool]*) – If True and determinism is not required, a naive factorization will be learnt over the data slice modeled by the current partition; otherwise, if True

and determinism is required, a disjunction will be learnt over the data slice modeled by the current partition.

- **is_conj** (*Optional[bool]*) – True if the modeled slice is associated to a conjunction, i.e. every row in the slice is equal to the others.

set_parent_partition(*parent_partition*)

Set the parent partition and update its sub_partitions attribute.

Parameters
parent_partition (Partition) – The parent partition.

is_partitioned()

Returns
True if the partition is partitioned, False otherwise.

is_horizontally_partitioned()

Returns
True if the partition is horizontally partitioned, False otherwise.

get_slice(*data*)

Slice the input data matrix according to self.

Parameters
data (*ndarray*) – The data to be sliced.

Returns
The data slice.

Return type
ndarray

get_vertical_split()

If possible, split vertically the current partition.

Return type
list[np.ndarray, np.ndarray]

get_conj_row_ids(*data*, *conj*, *min_part_inst*)

Return the row ids of the instances satisfying the given conjunction. The row ids must be found within the slice modeled by the self partition.

Parameters

- **data** (*ndarray*) – The input data.
- **conj** (*list*) – Conjunction modeled as a list of two lists: the first contains the IDs of the variables, the second the related assignment. For example, [[8,3],[1,0]] models the conjunction X8=1 and X3=0.
- **min_part_inst** (*int*) – the minimum number of instances allowed to return.

Returns
The row ids of the instances satisfying the given conjunction iff the number of such instances is greater or equal than the minimum number of instances allowed to return; otherwise, an empty array.

Return type
ndarray

get_horizontal_split(*data*, *min_part_inst*, *conj_len*, *arity*, *sd*, *random_state*)

If possible, split horizontally the current partition.

Parameters

- **data** (*ndarray*) – The input data matrix.
- **min_part_inst** (*int*) – The minimum number of instances allowed per partition.
- **conj_len** (*int*) – The conjunction length.
- **arity** (*int*) – The maximum number of subpartitions for an horizontal partitioned partition.
- **sd** (*bool*) – True if the generated tree will be used to model a SD PC, False otherwise.
- **random_state** (*RandomState*) – The random state.

Return type

Tuple[list, *ndarray*, list]

deeprob.spn.utils.partitioning.**generate_random_partitioning**(*data*, *min_part_inst*, *n_max_parts*, *conj_len*, *arity*, *sd*, *uncond_vars*, *random_state*)

Create a random partition tree.

Parameters

- **data** (*ndarray*) – The input data matrix.
- **min_part_inst** (*int*) – The minimum number of instances allowed per partition.
- **n_max_parts** (*int*) – The maximum number of partitions in the tree.
- **conj_len** (*int*) – The conjunction length.
- **arity** (*int*) – The maximum number of subpartitions for an horizontal partitioned partition.
- **sd** (*bool*) – True if the generated tree will be used to model a SD PC, False otherwise.
- **uncond_vars** (*list*) – Ordered list of variables from which the first *conj_len* ones are extracted as conjunction variables to partition the root partition.
- **random_state** (*RandomState*) – The random state.

Return partition_root

The partition root of the tree.

Return cl_parts_l

List containing the leaf partitions over which a CLTree will be learnt.

Return conj_vars_l

List of lists. Every sublist contains the variables of a conjunction (e.g. [[3, 5]]). If a sublist occurs before another, then the former has been used first. There are no duplicates.

Return n_partitions

The number of partitions in the generated tree.

deeprob.spn.utils.statistics module

deeprob.spn.utils.statistics.**compute_statistics**(*root*)

Compute some statistics of a SPN given its root. The computed statistics are the following:

- n_nodes, the number of nodes
- n_sum, the number of sum nodes
- n_prod, the number of product nodes
- n_leaves, the number of leaves
- n_edges, the number of edges
- n_params, the number of parameters
- depth, the depth of the network

Parameters
root (Node) – The root of the SPN.

Returns
A dictionary containing the statistics.

Return type
dict

deeprob.spn.utils.statistics.**compute_edges_count**(*root*)

Get the number of edges of a SPN given its root.

Parameters
root (Node) – The root of the SPN.

Returns
The number of edges.

Return type
int

deeprob.spn.utils.statistics.**compute_parameters_count**(*root*)

Get the number of parameters of a SPN given its root.

Parameters
root (Node) – The root of the SPN.

Returns
The number of parameters.

Return type
int

deeprob.spn.utils.statistics.**compute_depth**(*root*)

Get the depth of the SPN given its root.

Parameters
root (Node) – The root of the SPN.

Returns
The depth of the network.

Return type
int

deeprob.spn.utils.validity module

deeprob.spn.utils.validity.**check_spn**(*root*, *labeled=True*, *smooth=False*, *decomposable=False*, *structured_decomposable=False*)

Check a SPN have certain properties. Defaults to checking only 'labeled'. This function combines several checks over a SPN, hence reducing the computational effort used to retrieve the nodes from the SPN.

Parameters

- **root** (Node) – The root node of the SPN.
- **labeled** (*bool*) – Whether to check if the SPN is correctly labeled.
- **smooth** (*bool*) – Whether to check if the SPN is smooth.
- **decomposable** (*bool*) – Whether to check if the SPN is decomposable.
- **structured_decomposable** (*bool*) – Whether to check if the SPN is structured decomposable.

Raises

ValueError – If the SPN doesn't have a certain property.

deeprob.spn.utils.validity.**is_labeled**(*root*, *nodes=None*)

Check if the SPN is labeled correctly. It checks that the initial id is zero and each id is consecutive.

Parameters

- **root** (Node) – The root of the SPN.
- **nodes** (*Optional[List[Node]]*) – The list of nodes. If None, it will be retrieved starting from the root node.

Returns

None if the SPN is labeled correctly, a reason otherwise.

Return type

Optional[str]

deeprob.spn.utils.validity.**is_smooth**(*root*, *nodes=None*)

Check if the SPN is smooth (or complete). It checks that each child of a sum node has the same scope.

Parameters

- **root** (Node) – The root of the SPN.
- **nodes** (*Optional[List[Node]]*) – The list of nodes. If None, it will be retrieved starting from the root node.

Returns

None if the SPN is smooth, a reason otherwise.

Return type

Optional[str]

deeprob.spn.utils.validity.**is_decomposable**(*root*, *nodes=None*)

Check if the SPN is decomposable (or consistent). It checks that each child of a product node has disjointed scopes.

Parameters

- **root** (Node) – The root of the SPN.

- **nodes** (*Optional[List[Node]]*) – The list of nodes. If None, it will be retrieved starting from the root node.

Returns
None if the SPN is decomposable, a reason otherwise.

Return type
Optional[str]

deeprob.spn.utils.validity.**is_structured_decomposable**(*root*, *nodes=None*)

Check if the PC is structured decomposable. It checks that product nodes follow a vtree. Note that if a PC is structured decomposable then it's also decomposable.

Parameters

- **root** (Node) – The root of the PC.
- **nodes** (*Optional[List[Node]]*) – The list of nodes. If None, it will be retrieved starting from the root node.

Returns
None if the PC is structured decomposable, a reason otherwise.

Return type
Optional[str]

deeprob.spn.utils.validity.**are_compatible**(*root_a*, *root_b*, *nodes_a=None*, *nodes_b=None*)

Check if two PCs are compatible.

Parameters

- **root_a** (Node) – The root of the first PC.
- **root_b** (Node) – The root of the second PC.
- **nodes_a** (*Optional[List[Node]]*) – The list of nodes of the first PC. If None, it will be retrieved starting from the root node.
- **nodes_b** (*Optional[List[Node]]*) – The list of nodes of the second PC. If None, it will be retrieved starting from the root node.

Returns
None if the two PCs are compatible, a reason otherwise.

Return type
Optional[str]

deeprob.spn.utils.validity.**collect_scopes**(*nodes*)

Collect the scopes of each node.

Parameters
nodes (*List[Node]*) – The list of nodes.

Returns
A list of scopes.

Return type
List[*Tuple*[int]]

Module contents

Module contents

deeprob.torch package

Submodules

deeprob.torch.base module

class deeprob.torch.base.**ProbabilisticModel**(**args*, ***kwargs*)

Bases: ABC, Module

Initializes internal Module state, shared by both nn.Module and ScriptModule.

has_rsample = False

log_prob(*x*)

Compute the log-likelihood of a batched sample. Note that the nn.Module.forward method of sub-classes must implement log-likelihood evaluation.

Parameters
x (*Tensor*) – The batched sample.

Returns
The batched log-likelihoods.

Return type
Tensor

abstract sample(*n_samples*, *y=None*)

Sample some values from the modeled distribution.

Parameters

- **n_samples** (*int*) – The number of samples.
- **y** (*Optional[Tensor]*) – The samples labels. It can be None.

Returns
The samples.

Return type
Tensor

abstract loss(*x*, *y=None*)

Compute the loss of the model.

Parameters

- **x** (*Tensor*) – The outputs of the model.
- **y** (*Optional[Tensor]*) – The ground-truth. It can be None.

Returns
The loss.

Return type
Tensor

apply_constraints()

Apply the constraints specified by the model.

training: bool

deeprob.torch.base.**DensityEstimator**

A density estimator is either a DeeProb-kit probabilistic model or a Torch distribution.

alias of Union[*ProbabilisticModel*, Distribution]

deeprob.torch.callbacks module

class deeprob.torch.callbacks.**EarlyStopping**(*model*, *patience=1*, *filepath='checkpoint.pt'*, *delta=0.001*)

Bases: object

Early stops the training if validation loss doesn't improve after a given number of consecutive epochs.

Parameters

- **model** (*Module*) – The model to monitor.
- **patience** (*int*) – The number of consecutive epochs to wait.
- **filepath** (*Union[PathLike, str]*) – The checkpoint filepath where to save the model state dictionary.
- **delta** (*float*) – The minimum change of the monitored quantity.

Raises

ValueError – If the patience or delta values are out of domain.

property should_stop: bool

Check if the training process should stop.

get_best_state()

Get the best model's state dictionary.

Return type

OrderedDict

__call__(*loss*, *epoch*)

Update the state of early stopping.

Parameters

- **loss** (*float*) – The validation loss measured.
- **epoch** (*int*) – The current epoch.

deeprob.torch.constraints module

class deeprob.torch.constraints.**ScaleClipper**(*eps=1e-05*)

Bases: Module

Constraints the scale to be positive.

Parameters

eps (*float*) – The epsilon minimum value threshold.

Raises
ValueError – If the epsilon value is out of domain.

forward(*module*)

Call the constraint.

Parameters
module (*Module*) – The module.

training: bool

deeprob.torch.datasets module

class deeprob.torch.datasets.UnsupervisedDataset(*dataset*, *transform=None*)

Bases: Dataset

Initialize an unsupervised dataset.

Parameters

- dataset – The dataset.
- transform – An optional transformation to apply.

property features_shape: Union[int, tuple]

Get the dataset features shape.

class deeprob.torch.datasets.SupervisedDataset(*dataset*, *targets*, *transform=None*)

Bases: Dataset

Initialize a supervised dataset.

Parameters

- dataset – The dataset.
- targets – The targets.
- transform – An optional transformation to apply.

property features_shape: Union[int, tuple]

Get the dataset features shape.

property num_classes: int

Get the number of classes.

class deeprob.torch.datasets.WrappedDataset(*dataset*, *unsupervised=True*, *classes=None*, *transform=None*)

Bases: Dataset

Initialize a wrapped dataset (either unsupervised or supervised).

Parameters

- dataset – The dataset (assumed to be supervised).
- unsupervised – Whether to treat the dataset as unsupervised.
- classes – The class domain. It can be None if unsupervised is True.
- transform – An optional transformation to apply.

property features_shape: Union[int, tuple]

Get the dataset features shape.

property num_classes: int

Get the number of classes.

deeprob.torch.initializers module

deeprob.torch.initializers.**dirichlet_**(*tensor*, *alpha=1.0*, *log_space=True*, *dim=- 1*)

Initialize a tensor using the symmetric Dirichlet distribution.

Parameters

- **tensor** (*Tensor*) – The tensor to initialize.
- **alpha** (*float*) – The concentration parameter.
- **log_space** (*bool*) – Whether to initialize the tensor in the logarithmic space.
- **dim** (*int*) – The dimension over which to sample.

deeprob.torch.metrics module

class deeprob.torch.metrics.**RunningAverageMetric**

Bases: object

Initialize a running average metric object.

__call__(*metric*, *num_samples*)

Accumulate a metric value.

Parameters

- **metric** (*float*) – The metric value.
- **num_samples** (*int*) – The number of samples from which the metric is estimated.

Raises

ValueError – If the number of samples is not positive.

reset()

Reset the running average metric accumulator.

average()

Get the metric average.

Returns

The metric average.

Return type

float

deeprob.torch.metrics.**fid_score**(*dataset1*, *dataset2*, *model=None*, *transform=None*, *batch_size=100*, *num_workers=0*, *device=None*, *verbose=True*)

Compute the Frechet Inception Distance (FID) between two data samples. This implementation has been readapted from https://github.com/mseitzer/pytorch-fid. IMPORTANT NOTE: The computed FID score is not comparable with other FID scores based on Tensorflow's InceptionV3.

Parameters

- **dataset1** (*Union[Dataset, Tensor]*) – The first samples data set.
- **dataset2** (*Union[Dataset, Tensor]*) – The second samples data set.
- **model** (*Optional[Module]*) – The model to use to extract the features. If None the Torchvision's InceptionV3 model pretrained on ImageNet will be used.
- **transform** (*Optional[Any]*) – An optional transformation to apply to every sample. If transform and model are both None, then the transformation resizes to 3x299x299 and normalizes values from (0, 1) to (-1, 1).
- **batch_size** (*int*) – The batch size to use when extracting features.
- **num_workers** (*int*) – The number of workers used for the data loaders.
- **device** (*Optional[device]*) – The device used to run the model. If it's None 'cuda' will be used, if available.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The FID score.

Return type
float

deeprob.torch.metrics.**extract_features**(*model*, *dataset*, *transform=None*, *device=None*, *verbose=True*, ***kwargs*)

Extract the features produced by a model using a data set.

Parameters

- **model** (*Module*) – The model to use to extract the features.
- **dataset** (*Union[Dataset, Tensor]*) – The data set.
- **transform** (*Optional[Any]*) – An optional transformation to apply to every sample.
- **device** (*Optional[device]*) – The device used to run the model. If it's None 'cuda' will be used, if available.
- **verbose** (*bool*) – Whether to enable verbose mode.
- **kwargs** – Additional parameters to pass to the data loader.

Returns
The extracted features for each data sample.

Return type
Tensor

deeprob.torch.routines module

deeprob.torch.routines.**train_model**(*model*, *data_train*, *data_valid*, *setting*, *lr=0.001*, *batch_size=100*, *epochs=1000*, *optimizer='adam'*, *optimizer_kwargs=None*, *patience=20*, *checkpoint='checkpoint.pt'*, *train_base=True*, *drop_last=True*, *num_workers=0*, *device=None*, *verbose=True*)

Train a Torch model.

Parameters

- **model** (ProbabilisticModel) – The model to train.

- **data_train** (*Union[ndarray, Dataset]*) – The train dataset.
- **data_valid** (*Union[ndarray, Dataset]*) – The validation dataset.
- **setting** (*str*) – The train setting. It can be either 'generative' or 'discriminative'.
- **lr** (*float*) – The learning rate to use.
- **batch_size** (*int*) – The batch size for both train and validation.
- **epochs** (*int*) – The number of epochs.
- **optimizer** (*str*) – The optimizer to use.
- **optimizer_kwargs** (*Optional[dict]*) – A dictionary containing additional optimizer parameters.
- **patience** (*int*) – The epochs patience for early stopping.
- **checkpoint** (*Union[PathLike, str]*) – The checkpoint filepath used for early stopping.
- **train_base** (*bool*) – Whether to train the input base module. Only applicable for normalizing flows.
- **drop_last** (*bool*) – Whether to drop the last train data batch having size less than the specified batch size.
- **num_workers** (*int*) – The number of workers for data loading.
- **device** (*Optional[device]*) – The device used for training. If it's None 'cuda' will be used, if available.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The train history.

Raises
ValueError – If a parameter is out of domain.

Return type
Union[*Dict*[str, list], *Dict*[str, *Dict*[str, list]]]

deeprob.torch.routines.**train_generative**(*model*, *train_loader*, *valid_loader*, *optimizer*, *device*, *early_stopping*, *epochs=1000*, *train_base=True*, *verbose=True*)

Train a Torch model in generative setting.

Parameters

- **model** (ProbabilisticModel) – The model.
- **train_loader** (*DataLoader*) – The train data loader.
- **valid_loader** (*DataLoader*) – The validation data loader.
- **optimizer** (*Optimizer*) – The optimize to use.
- **device** (*device*) – The device to use for training.
- **epochs** (*int*) – The number of epochs.
- **early_stopping** (EarlyStopping) – The early stopping callback object.
- **train_base** (*bool*) – Whether to train the input base module. Only applicable for normalizing flows.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The train history with keys 'train' and 'validation'.

Raises
`ValueError` – If a parameter is out of domain.

Return type
Dict[str, list]

deeprob.torch.routines.**train_discriminative**(*model*, *train_loader*, *valid_loader*, *optimizer*, *device*, *early_stopping*, *epochs=1000*, *train_base=True*, *verbose=True*)

Train a Torch model in discriminative setting.

Parameters

- **model** (`ProbabilisticModel`) – The model.
- **train_loader** (*DataLoader*) – The train data loader.
- **valid_loader** (*DataLoader*) – The validation data loader.
- **optimizer** (*Optimizer*) – The optimize to use.
- **device** (*device*) – The device to use for training.
- **early_stopping** (`EarlyStopping`) – The early stopping callback object.
- **epochs** (*int*) – The number of epochs.
- **train_base** (*bool*) – Whether to train the input base module. Only applicable for normalizing flows.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The train history with keys 'train' and 'validation' and for both keys 'loss' and 'accuracy'.

Raises
`ValueError` – If a parameter is out of domain.

Return type
Dict[str, *Dict*[str, list]]

deeprob.torch.routines.**test_model**(*model*, *data_test*, *setting*, *batch_size=100*, *num_workers=0*, *device=None*, *verbose=True*)

Test a Torch model.

Parameters

- **model** (`ProbabilisticModel`) – The model to test.
- **data_test** (*Union[ndarray, Dataset]*) – The test dataset.
- **setting** (*str*) – The test setting. It can be either 'generative' or 'discriminative'.
- **batch_size** (*int*) – The batch size for testing.
- **num_workers** (*int*) – The number of workers for data loading.
- **device** (*Optional[device]*) – The device used for training. If it's None 'cuda' will be used, if available.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The mean log-likelihood and two standard deviations if setting='generative'. The negative log-likelihood and classification metrics if setting='discriminative'.

Raises
`ValueError` – If a parameter is out of domain.

Return type
Union[*Tuple*[float, float], *Tuple*[float, dict]]

deeprob.torch.routines.**test_generative**(*model*, *test_loader*, *device*, *verbose=True*)

Test a Torch model in generative setting.

Parameters

- **model** (`ProbabilisticModel`) – The model to test.
- **test_loader** (*DataLoader*) – The test data loader.
- **device** (*device*) – The device used for testing.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The mean log-likelihood and two standard deviations.

Return type
Tuple[float, float]

deeprob.torch.routines.**test_discriminative**(*model*, *test_loader*, *device*, *verbose=True*)

Test a Torch model in discriminative setting.

Parameters

- **model** (`ProbabilisticModel`) – The model to test.
- **test_loader** (*DataLoader*) – The test data loader.
- **device** (*device*) – The device used for testing.
- **verbose** (*bool*) – Whether to enable verbose mode.

Returns
The negative log-likelihood and classification report dictionary.

Return type
Tuple[float, dict]

deeprob.torch.transforms module

class deeprob.torch.transforms.**Transform**

Bases: `ABC`

Generic data transform function.

__call__(*x*)

Evaluate in forward mode the transformation. Equivalent to forward(x).

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

abstract forward(*x*)

Evaluate in forward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

abstract backward(*x*)

Evaluate in backward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

class deeprob.torch.transforms.**TransformList**(*iterable=()*, */*)

Bases: *Transform*, list

A list of transformations.

forward(*x*)

Evaluate in forward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

backward(*x*)

Evaluate in backward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

class deeprob.torch.transforms.**Normalize**(*mean*, *std*, *eps=1e-07*)

Bases: *Transform*

Initialize a normalization transformation. This transformation computes the following equations:

y = (x - mean) / (std + eps)

x = y * (std + eps) + mean

Parameters

- **mean** (*Union[float, Tensor]*) – The mean values. One for each channel.
- **std** (*Union[float, Tensor]*) – The standard deviation values.
- **eps** (*float*) – The epsilon value (used to avoid divisions by zero).

Raises

ValueError – If the epsilon value is out of domain.

forward(*x*)

Evaluate in forward mode the transformation.

Parameters

x (*Tensor*) – The inputs.

Returns

The outputs.

Return type

Tensor

backward(*x*)

Evaluate in backward mode the transformation.

Parameters

x (*Tensor*) – The inputs.

Returns

The outputs.

Return type

Tensor

class deeprob.torch.transforms.**Quantize**(*n_bits=8*)

Bases: *Transform*

Initialize a quantization transformation. This transformation computes the following equations:

y = clamp(floor(x * 2 ** n_bits), 0, 2 ** n_bits - 1) / (2 ** n_bits - 1)
x = ((x * (2 ** n_bits - 1)) + u) / (2 ** n_bits)
with u ~ Uniform(0, 1)

Parameters

n_bits (*int*) – The number of bits.

Raises

ValueError – If the number of bits is not positive.

forward(*x*)

Evaluate in forward mode the transformation.

Parameters

x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

backward(*x*)

Evaluate in backward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

class deeprob.torch.transforms.**Flatten**(*shape=None*)

Bases: *Transform*

Initialize a flatten transformation.

Parameters
shape (*Optional[Union[Size, List[int], Tuple[int, ...]]]*) – The original tensor shape. It can be None to enable only forward transformation.

forward(*x*)

Evaluate in forward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

backward(*x*)

Evaluate in backward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

class deeprob.torch.transforms.**Reshape**(*target_shape*, *shape=None*)

Bases: *Transform*

Initialize a reshape transformation.

Parameters

- **target_shape** (*Union[Size, List[int], Tuple[int, ...]]*) – The target tensor shape.
- **shape** (*Union[Size, List[int], Tuple[int, ...]]*) – The input tensor shape.

forward(*x*)

Evaluate in forward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

backward(*x*)

Evaluate in backward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

class deeprob.torch.transforms.**RandomHorizontalFlip**(*p=0.5*)

Bases: *Transform*

Initialize a random horizontal flip transformation.

Parameters
p (*float*) – The probability of flipping.

Raises
ValueError – If the probability of flipping is out of domain.

forward(*x*)

Evaluate in forward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

backward(*x*)

Evaluate in backward mode the transformation.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs.

Return type
Tensor

deeprob.torch.utils module

deeprob.torch.utils.**get_activation_class**(*name*)

Get the activation function class by its name.

Parameters
name (*str*) – The activation function's name. It can be one of: 'relu', 'leaky-relu', 'softplus', 'tanh', 'sigmoid'.

Returns
The activation function class.

Raises
ValueError – If the activation function's name is not known.

deeprob.torch.utils.**get_optimizer_class**(*name*)

Get the optimizer class by its name.

Parameters
name (*str*) – The optimizer's name. It can be 'sgd', 'rmsprop', 'adagrad', 'adam'.

Returns
The optimizer class.

Raises
ValueError – If the optimizer's name is not known.

class deeprob.torch.utils.**ScaledTanh**(*weight_size=1*)

Bases: Module

Build the module.

Parameters
weight_size (*Union[int, tuple, list]*) – The size of the weight parameter.

forward(*x*)

Apply the scaled tanh function.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs of the module.

Return type
Tensor

training: bool

class deeprob.torch.utils.**MaskedLinear**(*in_features*, *out_features*, *mask*)

Bases: Linear

Build a masked linear layer.

Parameters

- **in_features** (*int*) – The number of input features.
- **out_features** (*int*) – The number of output features.
- **mask** (*ndarray*) – The mask to apply to the weights of the layer.

Raises
ValueError – If the mask parameter is not consistent with the number of input and output features.

forward(*x*)

Evaluate the layer given some inputs.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs of the module.

Return type
Tensor

in_features: int

out_features: int

weight: Tensor

class deeprob.torch.utils.**WeightNormConv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *bias=True*)

Bases: Module

Initialize a Conv2d layer with weight normalization.

Parameters
- **in_channels** (*int*) – The number of input channels.
- **out_channels** (*int*) – The number of output channels.
- **kernel_size** (*Union[int, Tuple[int, int]]*) – The convolving kernel size.
- **stride** (*Union[int, Tuple[int, int]]*) – The stride of convolution.
- **padding** (*Union[int, Tuple[int, int]]*) – The padding to apply.
- **bias** (*bool*) – Whether to use bias parameters.

forward(*x*)

Evaluate the weight-normalized convolutional layer.

Parameters
x (*Tensor*) – The inputs.

Returns
The outputs of the module.

Return type
Tensor

training: bool

Module contents

deeprob.utils package

Submodules

deeprob.utils.data module

class deeprob.utils.data.**DataTransform**

Bases: ABC

Abstract data transformation.

abstract fit(*data*)

Fit the data transform with some data.

Parameters
data (*ndarray*) – The data for fitting.

abstract forward(*data*)

Apply the data transform to some data.

Parameters
data (*ndarray*) – The data to transform.

Returns
The transformed data.

Return type
ndarray

abstract backward(*data*)

Apply the backward data transform to some data.

Parameters
data (*ndarray*) – The data to transform.

Returns
The transformed data.

Return type
ndarray

class deeprob.utils.data.**DataFlatten**

Bases: *DataTransform*

Build the data flatten transformation.

fit(*data*)

Fit the data transform with some data.

Parameters
data (*ndarray*) – The data for fitting.

forward(*data*)

Apply the data transform to some data.

Parameters
data (*ndarray*) – The data to transform.

Returns
The transformed data.

Return type
ndarray

backward(*data*)

Apply the backward data transform to some data.

Parameters
data (*ndarray*) – The data to transform.

Returns
The transformed data.

Return type
ndarray

class deeprob.utils.data.**DataNormalizer**(*interval=None*, *clip=False*, *dtype=<class 'numpy.float32'>*)

Bases: *DataTransform*

Build the data normalizer transformation.

Parameters

- **interval** (*Optional[Tuple[float, float]]*) – The normalizing interval. If None data will be normalized in [0, 1].
- **clip** (*bool*) – Whether to clip data if out of interval.
- **dtype** – The type for type conversion.

Raises
ValueError – If the normalizing interval is out of domain.

fit(*data*)

Fit the data transform with some data.

Parameters
data (*ndarray*) – The data for fitting.

forward(*data*)

Apply the data transform to some data.

Parameters
data (*ndarray*) – The data to transform.

Returns
The transformed data.

Return type
ndarray

backward(*data*)

Apply the backward data transform to some data.

Parameters
data (*ndarray*) – The data to transform.

Returns
The transformed data.

Return type
ndarray

class deeprob.utils.data.**DataStandardizer**(*sample_wise=True*, *eps=1e-07*, *dtype=<class 'numpy.float32'>*)

Bases: *DataTransform*

Build the data standardizer transformation.

Parameters

- **sample_wise** (*bool*) – Whether to apply sample wise standardization.
- **eps** (*float*) – The epsilon value for standardization.
- **dtype** – The type for type conversion.

Raises

ValueError – If the epsilon value is out of domain.

fit(*data*)

Fit the data transform with some data.

Parameters

data (*ndarray*) – The data for fitting.

forward(*data*)

Apply the data transform to some data.

Parameters

data (*ndarray*) – The data to transform.

Returns

The transformed data.

Return type

ndarray

backward(*data*)

Apply the backward data transform to some data.

Parameters

data (*ndarray*) – The data to transform.

Returns

The transformed data.

Return type

ndarray

deeprob.utils.data.**ohe_data**(*data*, *domain*)

One-Hot-Encoding function.

Parameters

- **data** (*ndarray*) – The 1D data to encode.
- **domain** (*Union[List[int], ndarray]*) – The domain to use.

Returns

The One Hot encoded data.

Return type

ndarray

deeprob.utils.data.**mixed_ohe_data**(*data*, *domains*)

One-Hot-Encoding function, applied on mixed data (both continuous and non-binary discrete). Note that One-Hot-Encoding is applied only on categorical random variables having more than two values.

Parameters

- **data** (*ndarray*) – The data matrix to encode.
- **domains** (*List[Union[list, tuple]]*) – The domains to use.

Returns
The One Hot encoded data.

Raises
ValueError – If there are inconsistencies between the data and domains.

Return type
ndarray

deeprob.utils.data.**ecdf_data**(*data*)

Empirical Cumulative Distribution Function (ECDF).

Parameters
data (*ndarray*) – The data.

Returns
The result of the ECDF on data.

Return type
ndarray

deeprob.utils.data.**check_data_dtype**(*data*, *dtype=<class 'numpy.float32'>*)

Check whether the data is compatible with a given dtype (defaults to np.float32). If the data dtype is not compatible, then cast it.

Parameters

- **data** (*ndarray*) – The data.
- **dtype** (*Type[dtype]*) – The desidered dtype compatibility (defaults to np.float32).

Returns
The casted data if necessary, otherwise returns data itself.

deeprob.utils.graph module

class deeprob.utils.graph.**TreeNode**(*node_id*, *parent=None*)

Bases: object

Initialize a binary CLT.

Parameters

- **node_id** (*int*) – The ID of the node.
- **parent** (TreeNode) – The parent node.

get_id()

Get the ID of the node.

Returns
The ID of the node.

Return type
int

`get_parent()`

Get the parent node.

Returns
The parent node, None if the node has no parent.

Return type
TreeNode

`get_children()`

Get the children list of the node.

Returns
The children list of the node.

Return type
List[TreeNode]

`set_parent(parent)`

Set the parent node and update its children list.

Parameters
parent (TreeNode) – The parent node.

`is_leaf()`

Check whether the node is leaf.

Returns
True if the node is leaf, False otherwise.

Return type
bool

`get_n_nodes()`

Get the number of the nodes of the tree rooted at self.

Returns
The number of nodes of the tree rooted at self.

Return type
int

`get_tree_scope()`

Return the list of predecessors and the related scope of the tree rooted at self. Note that tree[root] must be -1, as it doesn't have a predecessor.

Return tree
List of predecessors.

Return scope
The related scope list.

Return type
Tuple[list, list]

`deeprob.utils.graph.build_tree_structure(tree, scope=None)`

Build a Tree node recursive data structure given a tree structure encoded as a list of predecessors. Note that tree[root] must be -1, as it doesn't have a predecessor. Optionally, a scope can be used to specify the tree node ids.

Parameters

- **tree** (*Union[List[int], ndarray]*) – The tree structure, as a sequence of predecessors.
- **scope** (*Optional[List[int]]*) – An optional scope, as a list of ids.

Returns
The Tree node structure's root.

Raises

- **ValueError** – If the tree structure is not compatible with the root node.
- **ValueError** – If the scope contains duplicates.
- **ValueError** – If the scope is incompatible with the tree structure.

Return type
TreeNode

deeprob.utils.graph.**compute_bfs_ordering**(*tree*)

Compute the breadth-first-search variable ordering given a tree structure. Note that tree[root] must be -1, as it doesn't have a predecessor.

Parameters
tree (*Union[List[int], ndarray]*) – The tree structure, as a sequence of predecessors.

Returns
The BFS variable ordering as a Numpy array.

Return type
Union[*List*[int], *ndarray*]

deeprob.utils.graph.**maximum_spanning_tree**(*root*, *adj_matrix*)

Compute the maximum spanning tree of a graph starting from a given root node.

Parameters

- **root** (*int*) – The root node index.
- **adj_matrix** (*ndarray*) – The graph's adjacency matrix.

Returns
The breadth first traversal ordering and the maximum spanning tree. The maximum spanning tree is given as a list of predecessors.

Return type
Tuple[*ndarray*, *ndarray*]

deeprob.utils.random module

deeprob.utils.random.**RandomState**

A random state type is either an integer seed value or a Numpy RandomState instance.

alias of Union[int, RandomState]

deeprob.utils.random.**check_random_state**(*random_state=None*)

Check a possible input random state and return it as a Numpy's RandomState object.

Parameters
random_state (*Optional[Union[int, RandomState]]*) – The random state to check. If None a new Numpy RandomState will be returned. If not None, it can be either a seed integer or a np.random.RandomState instance. In the latter case, itself will be returned.

Returns
A Numpy's RandomState object.

Raises
`ValueError` – If the random state is not None or a seed integer or a Numpy RandomState object.

Return type
RandomState

deeprob.utils.region module

class deeprob.utils.region.**RegionGraph**(*n_features*, *depth*, *random_state=None*)

Bases: `object`

Initialize a region graph.

A region graph is defined w.r.t. a set of indices of random variable in a SPN. A *region* R is defined as a non-empty subset of the indices, and represented as sorted tuples with unique entries. A *partition* P of a region R is defined as a collection of non-empty sets, which are non-overlapping, and whose union is R. R is also called *parent* region of P. Any region C such that C is in partition P is called *child region* of P. So, a region is represented as a sorted tuple of integers (unique elements) and a partition is represented as a sorted tuple of regions (non-overlapping, not-empty, at least 2). A *region graph* is an acyclic, directed, bi-partite graph over regions and partitions. So, any child of a region R is a partition of R, and any child of a partition is a child region of the partition. The root of the region graph is a sorted tuple composed of all the elements. The leaves of the region graph must also be regions. They are called input regions, or leaf regions. Given a region graph, we can easily construct a corresponding SPN: 1) Associate I distributions to each input region. 2) Associate K sum nodes to each other (non-input) region. 3) For each partition P in the region graph, take all cross-products (as product nodes) of distributions/sum nodes associated with the child regions. Connect these products as children of all sum nodes in the parent region of P. In the end, this procedure will always deliver a complete and decomposable SPN.

Parameters

- **n_features** (*int*) – The number of features.
- **depth** (*int*) – The maximum depth.
- **random_state** (*Optional[Union[int, RandomState]]*) – The random state. It can be either None, a seed integer or a Numpy RandomState.

Raises
`ValueError` – If a parameter is out of domain.

random_layers()

Generate a list of layers randomly over a single repetition of features.

Returns
A list of layers, alternating between regions and partitions.

Return type
List[*List*[tuple]]

make_layers(*n_repetitions=1*)

Generate a random graph's layers over multiple repetitions of features.

Parameters
n_repetitions (*int*) – The number of repetitions.

Returns
A list of layers, alternating between regions and partitions.

Raises
ValueError – If a parameter is out of domain.

Return type
List[*List*[tuple]]

deeprob.utils.statistics module

deeprob.utils.statistics.**compute_mean_quantiles**(*data*, *n_quantiles*)

Compute the mean quantiles of a dataset (Poon-Domingos).

Parameters

- **data** (*ndarray*) – The data.
- **n_quantiles** (*int*) – The number of quantiles.

Returns
The mean quantiles.

Raises
ValueError – If the number of quantiles is not valid.

Return type
ndarray

deeprob.utils.statistics.**compute_mutual_information**(*priors*, *joints*)

Compute the mutual information between each features, given priors and joints distributions.

Parameters

- **priors** (*ndarray*) – The priors probability distributions, as a (N, D) Numpy array having priors[i, k] = P(X_i=k).
- **joints** (*ndarray*) – The joints probability distributions, as a (N, N, D, D) Numpy array having joints[i, j, k, l] = P(X_i=k, X_j=l).

Returns
The mutual information between each pair of features, as a (N, N) Numpy symmetric matrix.

Raises

- ValueError – If there are inconsistencies between priors and joints arrays.
- ValueError – If joints array is not symmetric.
- ValueError – If priors or joints arrays don't encode valid probability distributions.

Return type
ndarray

deeprob.utils.statistics.**estimate_priors_joints**(*data*, *alpha=0.1*)

Estimate both priors and joints probability distributions from binary data.

This function returns both the prior distributions and the joint distributions. Note that priors[i, k] = P(X_i=k) and joints[i, j, k, l] = P(X_i=k, X_j=l).

Parameters

- **data** (*ndarray*) – The binary data matrix.
- **alpha** (*float*) – The Laplace smoothing factor.

Returns
A pair of priors and joints distributions.

Raises
ValueError – If the Laplace smoothing factor is out of domain.

Return type
Tuple[*ndarray*, *ndarray*]

deeprob.utils.statistics.**compute_gini**(*probs*)

Computes the Gini index given some probabilities.

Parameters
probs (*ndarray*) – The probabilities.

Returns
The Gini index.

Raises
ValueError – If the probabilities doesn't sum up to one.

Return type
float

deeprob.utils.statistics.**compute_bpp**(*avg_ll*, *shape*)

Compute the average number of bits per pixel (BPP).

Parameters

- **avg_ll** (*float*) – The average log-likelihood, expressed in nats.
- **shape** (*Union[int, tuple, list]*) – The number of dimensions or, alternatively, a sequence of dimensions.

Returns
The average number of bits per pixel.

deeprob.utils.statistics.**compute_fid**(*mean1*, *cov1*, *mean2*, *cov2*, *blocksize=64*, *eps=1e-06*)

Computes the Frechet Inception Distance (FID) between two multivariate Gaussian distributions. This implementation has been readapted from https://github.com/mseitzer/pytorch-fid.

Parameters

- **mean1** (*ndarray*) – The mean of the first multivariate Gaussian.
- **cov1** (*ndarray*) – The covariance of the first multivariate Gaussian.
- **mean2** (*ndarray*) – The mean of the second multivariate Gaussian.
- **cov2** (*ndarray*) – The covariance of the second multivariate Gaussian.
- **blocksize** (*int*) – The block size used by the matrix square root algorithm.
- **eps** (*float*) – Epsilon value used to avoid singular matrices.

Returns
The FID score.

Raises
ValueError – If there is a shape mismatch between input arrays.

Return type
float

deeprob.utils.statistics.**compute_prior_counts**(*data*)

Compute the counts of the values of an RV given the data.

Parameters
data (*ndarray*) – The binary data matrix.

Returns
The counts.

deeprob.utils.statistics.**compute_joint_counts**(*data*)

Compute the counts of the configurations of an RV and its parent given the data.

Parameters
data (*ndarray*) – The binary data matrix.

Returns
The counts.

Module contents

6.1.2 Submodules

6.1.3 deeprob.context module

deeprob.context.**is_check_dtype_enabled**()

Returns whether the context flag 'check_dtype' is enabled.

Return type
bool

deeprob.context.**is_check_spn_enabled**()

Returns whether the context flag 'check_spn' is enabled.

Return type
bool

class deeprob.context.**ContextState**(***kwargs*)

Bases: ContextDecorator

Thread-safe Context State that disables some flags during execution.

Current supported flags are the following: - check_dtype: bool = True, Whether to check (and cast when needed) Numpy arrays data types. - check_spn: bool = True, Whether to check the SPNs structure properties.

6.1.4 Module contents

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W