
LibSPN Keras

Release 0.6.0

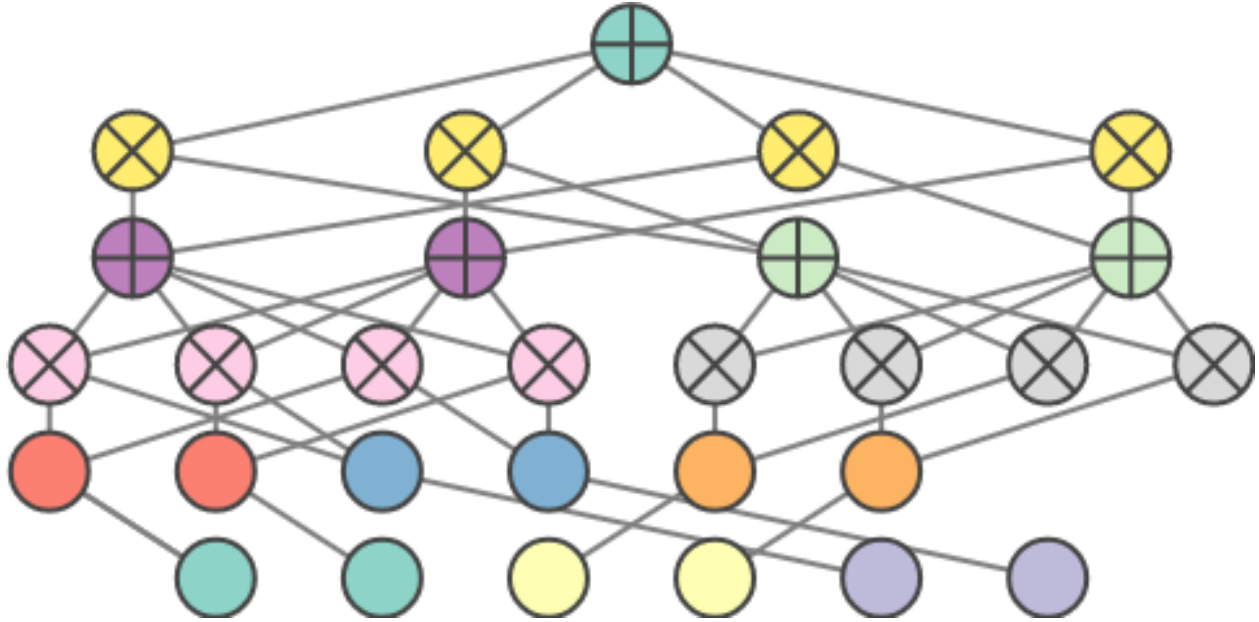
Jos van de Wolfshaar, Andrzej Pronobis

May 20, 2021

CONTENTS

1	Contents	3
2	Documentation	21
3	What are SPNs?	23
4	Dependencies	25
5	Installation	27
6	Note on stability of the repo	29
7	Feature Overview	31
8	Examples / Tutorials	33
9	TODOs	37
	Python Module Index	39
	Index	41

LibSPN Keras is a library for constructing and training Sum-Product Networks. By leveraging the Keras framework with a TensorFlow backend, it offers both ease-of-use and scalability. Whereas the previously available `libspn` focused on scalability, `libspn-keras` offers scalability **and** a straightforward Keras-compatible interface.



CONTENTS

1.1 API Reference

1.1.1 Layers

This submodule contains Keras layers for building SPNs. Besides leaf layers and regularization layers, there are two main groups of layers:

- Region layers for arbitrary decompositions of variables. Must be preceded with a *FlatToRegions* - *BaseLeaf* - *PermuteAndPadScopes* block. Regions are arbitrary sets of variables. A region graph describes how these sets of variables hierarchically define a probability distribution.
- Spatial layers for [Deep Generalized Convolutional Sum Product Networks](#)

All layers propagate **log probabilities** in the forward pass. So in case you want to know about the ‘raw’ probability in linear space, you simply pass the output of a layer through `exp`.

Leaf layers

Leaf layers transform raw observations to probabilities.

- *NormalLeaf*, *CauchyLeaf* and *LaplaceLeaf* can be used for continuous inputs.
- *IndicatorLeaf* should be used for discrete inputs.

If a variable is not part of the evidence, that means that variable should be marginalized out. This can be done by replacing the output of the corresponding components with 0 since that corresponds with 1 in log-space.

Continuous leaf layers

```
class libspn_keras.layers.NormalLeaf(num_components, location_initializer=None, location_trainable=True, scale_initializer=None, scale_trainable=False, accumulator_initializer=None, use_accumulators=False, scale_constraint=None, **kwargs)
```

Computes the log probability of multiple components per variable along the final axis.

Each component is modelled as a normal distribution with a diagonal covariance matrix.

Parameters

- **num_components** (int) – Number of components per variable

- **location_initializer** (Optional[Initializer]) – Initializer for location variable
- **location_trainable** (bool) – Boolean that indicates whether location is trainable
- **scale_initializer** (Optional[Initializer]) – Initializer for scale variable
- **scale_trainable** (bool) – Boolean that indicates whether scale is trainable
- ****kwargs** – kwargs to pass on to the keras.Layer super class

```
class libspn_keras.layers.CauchyLeaf (num_components, location_initializer=None, location_trainable=True, scale_initializer=None, accumulator_initializer=None, use_accumulators=False, scale_trainable=False, **kwargs)
```

Computes the log probability of multiple components per variable along the final axis.

Each component is modelled as a Cauchy distribution with a diagonal location matrix.

Parameters

- **num_components** (int) – Number of components per variable
- **location_initializer** (Optional[Initializer]) – Initializer for location variable
- **location_trainable** (bool) – Boolean that indicates whether location is trainable
- **scale_initializer** (Optional[Initializer]) – Initializer for scale variable
- ****kwargs** – kwargs to pass on to the keras.Layer super class

```
class libspn_keras.layers.LaplaceLeaf (num_components, location_initializer=None, location_trainable=True, scale_initializer=None, accumulator_initializer=None, use_accumulators=False, **kwargs)
```

Computes the log probability of multiple components per variable along the final axis.

Each component is modelled as a Laplace distribution with a diagonal location matrix.

Parameters

- **num_components** (int) – Number of components per variable
- **location_initializer** (Optional[Initializer]) – Initializer for location variable
- **location_trainable** (bool) – Boolean that indicates whether location is trainable
- **scale_initializer** (Optional[Initializer]) – Initializer for scale variable
- ****kwargs** – kwargs to pass on to the keras.Layer super class

Discrete leaf layers

```
class libspn_keras.layers.IndicatorLeaf (num_components, dtype=tf.int32, **kwargs)
```

Indicator leaf distribution taking integer inputs and producing a discrete indicator representation.

This effectively comes down to computing a one-hot representation along the final axis.

Parameters

- **num_components** (int) – Number of components, or indicators in this context.
- **dtype** (DType) – Dtype of input

- ****kwargs** – Kwargs to pass onto the `keras.layers.Layer` superclass.

Region layers

Region layers assume the tensors that are passed between them are of the shape `[num_scopes, num_decomps, num_batch, num_nodes]`. One region is given by the scope index + the decomposition (so it is indexed on the first two axes). This shape is chosen so that `matmul` operations done in `DenseSum` layers don't always require transposing first.

class `libspn_keras.layers.FlatToRegions` (*num_decomps*, ***kwargs*)

Flat representation to a dense representation.

Reshapes a flat input of shape `[batch, num_vars[, var_dimensionality]]` to `[batch, num_vars == scopes, decomp, var_dimensionality]`

If `var_dimensionality` is 1, the shape can also be `[batch, num_vars]`.

Parameters ****kwargs** – Keyword arguments to pass on the `keras.Layer` super class

Permutation layers

These layers permute variables so that this only has to be done once at the bottom of the network .. `autoclass:: libspn_keras.layers.PermuteAndPadScopes` .. `autoclass:: libspn_keras.layers.PermuteAndPadScopesRandom`

Sum Layers

class `libspn_keras.layers.DenseSum` (*num_sums*, *logspace_accumulators=None*, *accumulator_initializer=None*, *sum_op=None*, *accumulator_regularizer=None*, *logspace_accumulator_constraint=None*, *linear_accumulator_constraint=None*, ***kwargs*)

Computes densely connected sums per scope and decomposition.

Expects incoming `Tensor` to be of shape `[num_scopes, num_decomps, num_batch, num_nodes]`. If your input is passed through a `FlatToRegions` layer this is already taken care of.

Parameters

- **num_sums** (`int`) – Number of sums per scope
- **logspace_accumulators** (`Optional[bool]`) – If `True`, accumulators will be represented in log-space which is typically used with [SumOpGradBackprop](#). If `False`, accumulators will be represented in linear space. Weights are computed by normalizing the accumulators per sum, so that we always end up with a normalized SPN. If `None` (default) it will be set to `True` for [SumOpGradBackprop](#) and `False` otherwise.
- **accumulator_initializer** (`Optional[Initializer]`) – Initializer for accumulator. Will automatically be converted to log-space values if `logspace_accumulators` is enabled.
- **accumulator_regularizer** (`Optional[Regularizer]`) – Regularizer for accumulator (experimental)
- **linear_accumulator_constraint** (`Optional[Constraint]`) – Constraint for accumulator defaults to constraint that ensures small positive constant at minimum. Will be ignored if `logspace_accumulators` is set to `True`.

- **sum_op** (*SumOpBase*) – SumOpBase instance which determines how to compute the forward and backward pass of the weighted sums
- ****kwargs** – kwargs to pass on to keras.Layer super class

```
class libspn_keras.layers.RootSum(return_weighted_child_logits=True,  
                                logspace_accumulators=None,           accumulator_initializer=None,           accumulator_regularizer=None,           linear_accumulator_constraint=None,           sum_op=None,  
                                **kwargs)
```

Final sum of an SPN. Expects input to be in log-space and produces log-space output.

Parameters

- **return_weighted_child_logits** (*bool*) – If True, returns a weighted child log probability, which can be used for e.g. (Sparse)CategoricalCrossEntropy losses. If False, computes the weighted sum of the input, which effectively is the log probability of the distribution defined by the SPN.
- **logspace_accumulators** (*Optional[bool]*) – If True, accumulators will be represented in log-space which is typically used with [SumOpGradBackprop](#). If False, accumulators will be represented in linear space. Weights are computed by normalizing the accumulators per sum, so that we always end up with a normalized SPN. If None (default) it will be set to True for [SumOpGradBackprop](#) and False otherwise.
- **accumulator_initializer** (*Optional[Initializer]*) – Initializer for accumulator. If None, defaults to `initializers.Constant(1.0)`
- **accumulator_regularizer** (*Optional[Regularizer]*) – Regularizer for accumulator.
- **linear_accumulator_constraint** (*Optional[Constraint]*) – Constraint for linear accumulators. Defaults to a constraint that ensures a minimum of a small positive constant. If `logspace_accumulators` is set to True, this constraint will be ignored
- **sum_op** (*SumOpBase*) – SumOpBase instance which determines how to compute the forward and backward pass of the weighted sums
- ****kwargs** – kwargs to pass on to the keras.Layer super class

Product Layers

```
class libspn_keras.layers.DenseProduct(num_factors, **kwargs)
```

Computes products per decomposition and scope by an ‘n-order’ outer product.

Assumes the incoming tensor is of shape `[num_scopes, num_decomps, num_batch, num_nodes]` and produces an output of `[num_scopes // num_factors, num_decomps, num_batch, num_nodes ** num_factors]`. It can be considered a *dense* product as it computes all possible products given the scopes it has to merge.

Parameters

- **num_factors** (*int*) – Number of factors per product
- ****kwargs** – kwargs to pass on to the keras.Layer super class

```
class libspn_keras.layers.ReduceProduct(num_factors, **kwargs)
```

Computes products per decomposition and scope by reduction.

Assumes the incoming tensor is of shape `[num_batch, num_scopes, num_decomps, num_nodes]` and produces an output of `[num_batch, num_scopes // num_factors, num_decomps, num_nodes]`.

Parameters

- **num_factors** (int) – Number of factors per product
- ****kwargs** – kwargs to pass on to the `keras.Layer` super class.

Spatial layers

Spatial layers are layers needed for building **DGC-SPNs**. The final layer of an SPN should still be a `RootSum`. Use `SpatialToRegions` to convert the output from a spatial SPN to a region SPN.

```
class libspn_keras.layers.Local2DSum(num_sums,      logspace_accumulators=None,      ac-
                                     cumulator_initializer=None,      sum_op=None,
                                     accumulator_regularizer=None,
                                     logspace_accumulator_constraint=None,      lin-
                                     ear_accumulator_constraint=None, **kwargs)
```

Computes a spatial local sum, i.e. all cells will have unique weights.

In other words, there is no weight sharing across the spatial axes.

Parameters

- **num_sums** (int) – Number of sums per spatial cell. Corresponds to the number of channels in the output
- **logspace_accumulators** (Optional[bool]) – If `True`, accumulators will be represented in log-space which is typically used with `SumOpGradBackprop`. If `False`, accumulators will be represented in linear space. Weights are computed by normalizing the accumulators per sum, so that we always end up with a normalized SPN. If `None` (default) it will be set to `True` for `SumOpGradBackprop` and `False` otherwise.
- **accumulator_initializer** (Optional[Initializer]) – Initializer for accumulator
- **accumulator_regularizer** (Optional[Regularizer]) – Regularizer for accumulators
- **linear_accumulator_constraint** (Optional[Constraint]) – Constraint for accumulators (only applied if `log_space_accumulators==False`)
- **sum_op** (`SumOpBase`) – `SumOpBase` instance which determines how to compute the forward and backward pass of the weighted sums
- ****kwargs** – kwargs to pass on to the `keras.Layer` super class

```
class libspn_keras.layers.Conv2DSum(num_sums,      logspace_accumulators=None,      ac-
                                     cumulator_initializer=None,      sum_op=None,
                                     accumulator_regularizer=None,
                                     logspace_accumulator_constraint=None,      lin-
                                     ear_accumulator_constraint=None, **kwargs)
```

Computes a convolutional sum, i.e. weights are shared across the spatial axes.

Parameters

- **num_sums** (int) – Number of sums per spatial cell. Corresponds to the number of channels in the output

- **logspace_accumulators** (Optional[bool]) – If True, accumulators will be represented in log-space which is typically used with [SumOpGradBackprop](#). If False, accumulators will be represented in linear space. Weights are computed by normalizing the accumulators per sum, so that we always end up with a normalized SPN. If None (default) it will be set to True for [SumOpGradBackprop](#) and False otherwise.
- **accumulator_initializer** (Optional[Initializer]) – Initializer for accumulator
- **sum_op** ([SumOpBase](#)) – SumOpBase instance which determines how to compute the forward and backward pass of the weighted sums
- **accumulator_regularizer** (Optional[Regularizer]) – Regularizer for accumulators
- **linear_accumulator_constraint** (Optional[Constraint]) – Constraint for accumulators (only applied if log_space_accumulators==False)
- ****kwargs** – kwargs to pass on to the keras.Layer super class

```
class libspn_keras.layers.Conv2DProduct (strides, dilations, kernel_size,
                                         num_channels=None, padding='valid', depthwise=False, **kwargs)
```

Convolutional product as described in (Van de Wolfshaar and Pronobis, 2019).

Expects log-space inputs and produces log-space outputs.

Parameters

- **strides** (List[int]) – A tuple or list of strides
- **dilations** (List[int]) – A tuple or list of dilations
- **kernel_size** (List[int]) – A tuple or list of kernel sizes
- **num_channels** (Optional[int]) – Number of channels. If None, will be set to num_in_channels ** prod(kernel_sizes). This can be source of OOM problems quickly.
- **padding** (str) – Can be either ‘full’, ‘valid’ or ‘final’. Use ‘final’ for the top ConvProduct of a DGC-SPN. The other choices have the standard interpretation. Valid padding usually requires non-overlapping patches, whilst full padding is used with overlapping patches and exponentially increasing dilation rates, see also [Van de Wolfshaar, Pronobis (2019)].
- **depthwise** (bool) – Whether to use depthwise convolutions. If True, the value of num_channels will be ignored
- ****kwargs** – Keyword arguments to pass on to the keras.Layer superclass.

References

Deep Generalized Convolutional Sum-Product Networks for Probabilistic Image Representations, [Van de Wolfshaar, Pronobis \(2019\)](#)

```
class libspn_keras.layers.SpatialToRegions (*args, **kwargs)
```

Reshapes spatial SPN layer to a dense layer.

The dense output has leading dimensions for scopes and decomp (which will be [1, 1]).

Dynamic SPN layers

For reusing SPN structures along the temporal dimension one can implement dynamic SPNs. These rely on *template SPNs*, *top SPNs* and an *interface*. The interface of the previous timestep and the template at the current timestep can be combined through TemporalDenseProduct.

class libspn_keras.layers.TemporalDenseProduct (*args, **kwargs)

Computes ‘temporal’ dense products.

This is used to connect an interface stack at $t - 1$ of a dynamic SPN with a template SPN at t . Computes a product of all possible combinations of nodes along the last axis of the two incoming layers.

Parameters ****kwargs** – kwargs to pass on to the keras.Layer super class.

Regularization layers

class libspn_keras.layers.LogDropout (rate, noise_shape=None, seed=None, axis_at_least_one=None, **kwargs)

Log dropout layer.

Applies dropout in log-space. Should not precede product layers in an SPN, since their scope probability then potentially becomes -inf, resulting in NaN-values during training.

Parameters

- **rate** (float) – Rate at which to randomly dropout inputs.
- **noise_shape** (Optional[Tuple[int, ...]]) – Shape of dropout noise tensor
- **seed** (Optional[int]) – Random seed
- ****kwargs** – kwargs to pass on to the keras.Layer super class

Normalization

Normalize axes

class libspn_keras.layers.NormalizeAxes (value)

Enum for normalization axes.

Enumerates possible choices for normalization axes. `SAMPLE` orresponds to normalizing each sample. `VARIABLE_WISE` is for normalizing the values for each variable using statistics across all samples, while `GLOBAL` corresponds to statistics gathered from all input values (no specific axes excluded from reduction).

SAMPLE_WISE = 'sample-wise'

Normalize each sample

VARIABLE_WISE = 'variable-wise'

Normalize each variable

GLOBAL = 'global'

Normalize using all variables across all samples

Normalize layers

```
class libspn_keras.layers.NormalizeStandardScore (normalization_epsilon=1e-08,  
                                                axes=<NormalizeAxes.SAMPLE_WISE:  
                                                'sample-wise'>, **kwargs)
```

Normalizes samples to a standard score.

In other words, the output is the input minus its mean and divided by the standard deviation. This can be used to achieve the same kind of normalization as used in (Poon and Domingos, 2011).

Parameters

- **normalization_epsilon** (float) – Small positive constant to prevent division by zero, but could also be used a ‘smoothing’ factor.
- **axes** (NormalizeAxes) – If `NormalizeAxes.SAMPLE_WISE`, will compute z-scores where statistics are computed sample-wise, otherwise, computes z-scores through computing cross-sample statistics. To use cross-sample statistics, call `NormalizeStandardScore.adapt(train_ds)` where `train_ds` is an instance of `tf.data.Dataset`.
- ****kwargs** – kwargs to pass on to the `keras.Layer` super class

References

Sum-Product Networks, a New Deep Architecture [Poon and Domingos, 2011](#)

adapt (*ds*)

Compute cross-sample statistics. Assumes that `ds` is a batched dataset.

Parameters **ds** (`DatasetV2`) – Instance of `tf.data.Dataset` containing train data to compute statistics from.

Raises **RuntimeError** – Raised when axes are set to `SAMPLE_WISE`, in which case there is no point in calling `adapt`.

Return type `None`

1.1.2 Models

This submodule provides some out-of-the box model analogues of `Model`. They can be used to train SPNs for e.g. generative scenarios, where there is no label for an input. There’s also a [*DynamicSumProductNetwork*](#) that can be used for

Feedforward models

```
class libspn_keras.models.SumProductNetwork (*args, unsupervised=True, **kwargs)  
    An SPN analogue of tensorflow.keras.Model that can be trained generatively.
```

It does not expect labels `y` when calling `.fit()` if `unsupervised == True`.

Parameters **unsupervised** (*bool*) – If `True` (default) the model does not expect label inputs in `.fit()` or `.evaluate()`. Also, losses and metrics should not expect a target output, just a `y_hat`.

```
class libspn_keras.models.SequentialSumProductNetwork (layers, infer_no_evidence=False, infer_weighted_sum=True, unsupervised=None, **kwargs)
```

An analogue of `Sequential` that can be trained in an unsupervised way.

It does not expect labels `y` when calling `.fit()` if `unsupervised == True`. Inherits from `Sequential`, so layers are passed to it as a list.

Parameters

- **layers** (*list of Layer*) – List of `tf.keras.layers.Layer` instance
- **infer_no_evidence** (*bool*) – If `True`, the model expects an evidence mask defined as a boolean tensor which is used to mask out variables that are not part of the evidence.
- **unsupervised** (*bool*) – If `True` the model does not expect label inputs in `.fit()` or `.evaluate()`. Also, losses and metrics should not expect a target output, just a `y_hat`. By default, it will be inferred from `infer_no_evidence` and otherwise defaults to `True`.
- **infer_weighted_sum** (*bool*) – If `True` gives weighted sum of leaf representations, where coefficients are given by backprop signals. Otherwise chooses argmax of those coefficients.

zero_evidence_inference (*size*)

Do inference when no evidence at all is provided.

This means that internally, all evidence booleans are set to `False`.

Parameters **size** (*Tensor*) – Size of batch.

Return type *Tensor*

Returns Representation of batch size in absence of evidence.

Temporal models

```
class libspn_keras.models.DynamicSumProductNetwork (template_network, interface_network_t0, interface_network_t_minus_1, top_network, return_last_step=True, unsupervised=True, **kwargs)
```

SPN that re-uses its nodes at each time step.

The input is expected to be pre-padded sequences with a full tensor shape of `[num_batch, max_sequence_len, num_variables]`.

Parameters

- **template_network** (*Model*) – Template network that is applied to the leaves and ends with nodes that cover all variables for each timestep.
- **interface_network_t0** (*Model*) – Interface network for `t = t0`, applied on top of the template network's output at the current timestep.
- **interface_network_t_minus_1** (*Model*) – Interface network for `t = t0 - 1`, applied to the output of the interfaced output of the previous timestep

- **top_network** (`Model`) – Network on top of the interfaced network at the current timestep (covers all variables of the current timestep, including those of previous timesteps). This network must end with a root sum layer.
- **return_last_step** (`bool`) – Whether to return only the roots at the last step with shape `[num_batch, root_num_out]` or whether to `[num_batch, max_sequence_len, root_num_out]`
- **unsupervised** (`bool`) –

1.1.3 Sum Operations

LibSPN-Keras offers a convenient way to control the backward pass for sum operations used in the SPNs you build. Internally, LibSPN-Keras defines a couple of sum operations with different backward passes, for gradient as well as EM learning. All of these operations inherit from the `SumOpBase`.

NOTE

By default, LibSPN-Keras uses `SumOpGradBackprop`.

Getting And Setting a Sum Op

These methods allow for setting and getting the current default `SumOpBase`. By setting a default all sum layers (`DenseSum`, `Conv2DSum`, `Local2DSum` and `RootSum`) will use that sum op, unless you explicitly provide a `SumOpBase` instance to any of those classes when initializing them.

```
libspn_keras.set_default_sum_op(op)
```

Set default sum op to conveniently use it throughout an SPN architecture.

Parameters `op` (`SumOpBase`) – Implementation of sum op with corresponding backward pass definitions

Return type `None`

```
libspn_keras.get_default_sum_op()
```

Obtain default sum op.

Return type `SumOpBase`

Returns The default sum op.

Sum Operation With Gradients In Backward Pass

```
class libspn_keras.SumOpGradBackprop(logspace_accumulators=None)
```

Sum op primitive with gradient in backpropagation when computed through TensorFlow's autograd engine.

Internally, weighted sums are computed with default gradients for all ops being used.

Parameters `logspace_accumulators` (`Optional[bool]`) – If provided overrides default log-space choice. For a `SumOpGradBackprop` the default is `True`

Sum Operations With EM Signals In Backward Pass

class libspn_keras.SumOpEMBackprop

Sum op primitive with EM signals in backpropagation.

These are dense EM signals as opposed to the other EM based instances of SumOpBase

class libspn_keras.SumOpHardEMBackprop (*sample_prob=None*)

Sum op with hard EM signals in backpropagation when computed through TensorFlow's autograd engine.

Parameters **sample_prob** (Union[float, Tensor, None]) – Sampling probability in the range of [0, 1]. Sampling logits are taken from the normalized log probability of the children of each sum.

class libspn_keras.SumOpUnweightedHardEMBackprop (*sample_prob=None*)

Sum op with hard EM signals in backpropagation when computed through TensorFlow's autograd engine.

Instead of using weighted sum inputs to select the maximum child, it relies on unweighted child inputs, which has the advantage of alleviating a self-amplifying chain of hard EM signals in deep SPNs.

Parameters **sample_prob** (Union[float, Tensor, None]) – Sampling probability in the range of [0, 1]. Sampling logits are taken from the normalized log probability of the children of each sum.

Sum Operations With Sample Signals In Backward Pass

class libspn_keras.SumOpSampleBackprop

Sum op with hard EM signals in backpropagation when computed through TensorFlow's autograd engine.

Parameters **sample_prob** – Sampling probability in the range of [0, 1]. Sampling logits are taken from the normalized log probability of the children of each sum.

1.1.4 Initializers

In addition to initializers in `tensorflow.keras.initializers`, `libspn-keras` implements a few more useful initialization schemes for both leaf layers as well as sum weights.

Setting Defaults

Since accumulator initializers are often the same for all layers in an SPN, `libspn-keras` provides the following functions to get and set default accumulator initializers. These can still be overridden by providing the initializers explicitly at initialization of a layer.

libspn_keras.set_default_accumulator_initializer (*initializer*)

Configure the default accumulator that will be used for sum accumulators.

Parameters **initializer** (Initializer) – The initializer which will be used by default for sum accumulators.

Return type None

libspn_keras.get_default_accumulator_initializer ()

Obtain default accumulator initializer.

Return type Initializer

Returns The default accumulator initializer that will be use in sum accumulators, unless specified explicitly at initialization.

Location initializers

For a leaf distribution of the location scale family, the following initializers can be used for initializing the location parameters

class libspn_keras.initializers.**PoonDomingosMeanOfQuantileSplit** (*data=None*)
Initializes the data according to the algorithm described in (Poon and Domingos, 2011).

The data is divided over K quantiles where K is the number of nodes along the last axis of the tensor to be initialized. The quantiles are computed over all samples in the provided *data*. Then, the mean per quantile is taken as the value for initialization.

Parameters *data* (*numpy.ndarray*) – Data to compute quantiles over

References

Sum-Product Networks, a New Deep Architecture [Poon and Domingos, 2011](#)

class libspn_keras.initializers.**KMeans** (*data=None*, *samplewise_normalization=True*,
data_fraction=0.2, *normalization_epsilon=0.01*, *stop_epsilon=0.0001*,
num_iters=100, *group_centroids=True*,
max_num_clusters=8, *jitter_factor=0.05*, *centroid_initialization='kmeans++'*, *downsample=None*, *use_groups=False*)

Initializer learned through K-means from data.

The centroids learned from K-means are used to initialize the location parameters of a location-scale leaf, such as a *NormalLeaf*. This is particularly useful for variables with dimensionality of greater than 1.

Notes

Currently only works for spatial SPNs.

Parameters

- **data** (*numpy.ndarray*) – Data on which to perform K-means.
- **samplewise_normalization** (*bool*) – Whether to normalize data before learning centroids.
- **data_fraction** (*float*) – Fraction of the data to use for K-means (chosen randomly)
- **normalization_epsilon** (*float*) – Normalization constant (only used when *sample_normalization* is True).
- **stop_epsilon** (*float*) – Non-zero constant for difference in MSE on which to stop K-means fitting.
- **num_iters** (*int*) – Maximum number of iterations.
- **group_centroids** (*bool*) – If True, performs another round of K-means to group the centroids along the scope axes.
- **max_num_clusters** (*int*) – Maximum number of clusters (use this to limit the memory needed)
- **jitter_factor** (*float*) – If the number of clusters is larger than allowed according to *max_num_clusters*, the learned *max_num_clusters* centroids are repeated and

then jittered with noise generated from a truncated normal distribution with a standard deviation of `jitter_factor`

- **centroid_initialization** (*str*) – Centroid initialization algorithm. If "kmeans++", will iteratively initialize clusters far apart from each other. Otherwise, the centroids will be initialized from the data randomly.

class libspn_keras.initializers.**Equidistant** (*minval=0.0, maxval=1.0*)

Initializer that generates tensors where the last axis is initialized with 'equidistant' values.

Parameters

- **minval** (*float*) – A python scalar or a scalar tensor. Lower bound of the range of random values to generate.
- **maxval** (*float*) – A python scalar or a scalar tensor. Upper bound of the range of random values to generate. Defaults to 1 for float types.

Scale initializers

For a leaf distribution of the location scale family, the following initializers can be used for initializing the scale parameters

class libspn_keras.initializers.**PoonDomingosStddevOfQuantileSplit** (*data=None*)

Initializes the data according to the algorithm described in (Poon and Domingos, 2011).

The data is divided over K quantiles where K is the number of nodes along the last axis of the tensor to be initialized. The quantiles are computed over all samples in the provided `data`. Then, the stddev per quantile is taken as the value for initialization.

Parameters **data** (*numpy.ndarray*) – Data to compute quantiles over

References

Sum-Product Networks, a New Deep Architecture [Poon and Domingos, 2011](#)

Weight initializers

class libspn_keras.initializers.**Dirichlet** (*axis=-2, alpha=0.1*)

Initializes all values in a tensor with $Dir(\alpha)$.

Parameters

- **axis** (*int*) – The axis over which to sample from a $Dir(\alpha)$.
- **alpha** (*float*) – The α parameter of the Dirichlet distribution. If a scalar, this is broadcast along the given axis.

Note Initializer for discrete EM ([SumOpHardEMBackprop](#) and [SumOpUnweightedHardEMBackprop](#)).

class libspn_keras.initializers.**EpsilonInverseFanIn** (*axis=-2, epsilon=0.0001*)

Initializes all values in a tensor with ϵK^{-1} .

Where K is the dimension at `axis`.

This is particularly useful for (unweighted) hard EM learning and should generally be avoided otherwise.

Parameters

- **axis** (int) – The axis for input nodes so that K^{-1} is the inverse fan in. Usually, this is -2 .
- **epsilon** (float) – A small non-zero constant

1.1.5 Losses

Supervised generative learning

The following loss can be used when trying to maximize $p(X, Y)$.

```
class libspn_keras.losses.NegativeLogJoint (reduction='auto', name=None)
    Compute  $-\log(p(X, Y))$ .
```

Assumes that its input is $\log(p(X|Y))$ where Y is indexed on the second axis. This can be used for supervised generative learning with gradient-based optimizers or (hard) expectation maximization.

Unsupervised generative learning

The following loss can be used when trying to maximize $p(X)$.

```
class libspn_keras.losses.NegativeLogLikelihood (reduction='auto', name=None)
    Marginalize logits over last dimension so that it computes  $-\log(p(X))$ .
```

This can be used for unsupervised generative learning.

1.1.6 Optimizers

Apart from the one(s) below, any optimizer in `tensorflow.keras.optimizers` can be used.

```
class libspn_keras.optimizers.OnlineExpectationMaximization (learning_rate=0.01,
                                                             accumu-
                                                             late_batches=1,
                                                             name='OnlineEM',
                                                             **kwargs)
```

Online expectation maximization.

Requires sum layers to use any of the EM-based `SumOpBase` instances, such as `SumOpEMBackprop` `SumOpHardEMBackprop`.

Parameters

- **learning_rate** (float) – Learning rate for EM. If learning rate is η , then updates are given by: $w \leftarrow (1 - \eta)w + \eta\Delta w$
- **accumulate_batches** (int) – The number of batches to accumulate gradients before applying updates.
- **name** (str) – Name of the optimizer
- **kwargs** – Remaining kwargs to pass to `Optimizer` superclass

1.1.7 Metrics

class libspn_keras.metrics.**LogLikelihood** (*name='llh', **kwargs*)
 Compute log marginal $1/N \sum \log(p(X))$.

Assumes that the last layer of the SPN is a RootSum. It ignores the `y_true` argument, since a target for Y is absent in unsupervised learning.

1.1.8 Constraints

Setting Defaults

Since constraints are often the same for all layers in an SPN, libspn-keras provides the following functions to get and set default constraints. These can still be overridden by providing the initializers explicitly at initialization of a layer.

Linear accumulator constraints

By default, the logspace accumulator constraint is set to *GreaterEqualEpsilonNormalized*.

`libspn_keras.set_default_linear_accumulators_constraint (op)`
 Set default sum op to conveniently use it throughout an SPN architecture.

Parameters `op` (Constraint) – A constraint applied to linear accumulators after updates

Return type None

`libspn_keras.get_default_linear_accumulators_constraint ()`
 Get default linear accumulator constraint.

Return type Optional[Constraint]

Returns A Constraint instance that was set with `set_default_linear_accumulators_constraint`

Logspace accumulator constraints

By default, the logspace accumulator constraint is set to *LogNormalized*.

`libspn_keras.set_default_logspace_accumulators_constraint (op)`
 Set default sum op to conveniently use it throughout an SPN architecture.

Parameters `op` (Constraint) – A constraint applied to logspace accumulators after updates

Return type None

`libspn_keras.get_default_logspace_accumulators_constraint ()`
 Get default logspace accumulator constraint.

Return type Optional[Constraint]

Returns A Constraint instance that was set with `set_default_logspace_accumulators_constraint`

Linear Weight Constraints

These should be used for linear accumulators.

```
class libspn_keras.constraints.GreaterEqualEpsilonNormalized(epsilon=1e-10,  
                                                         axis=- 2)
```

Constraints the weight to be greater than or equal to epsilon and then normalizes.

Parameters **epsilon** (float) – Constant, usually small non-zero

```
class libspn_keras.constraints.GreaterEqualEpsilon(epsilon=1e-10)
```

Constraints the weight to be greater than or equal to epsilon.

Parameters **epsilon** (float) – Constant, usually small non-zero

Log Weight Constraints

These should be used for log accumulators.

```
class libspn_keras.constraints.LogNormalized(axis=- 2)
```

Normalizes log-space weights.

Parameters **axis** (int) – Axis along which to normalize

Scale Constraints

The following constraint is useful for ensuring stable scale parameters in location-scale leaf layers.

```
class libspn_keras.constraints.Clip(min, max=None)
```

Constraints the weights to be between min and max.

Parameters

- **min** (float) – Minimum clip value
- **max** (Optional[float]) – Maximum clip value

1.1.9 Region graphs

The region graph utilities can be made to create SPNs by explicitly defining the region structure. This can be used to e.g. conveniently express some learned structure.

For some examples on how to use region graphs check out [this tutorial](#).

```
class libspn_keras.RegionVariable(index)
```

Represents a region in the SPN.

Regions graphs are DAGs just like SPNs, but they do not represent sums and products. They are merely used for defining the scope structure of the SPN.

A RegionVariable is a leaf node without children, as opposed to a RegionNode which is a node with children.

Parameters **index** (int) – Index of the variable

```
class libspn_keras.RegionNode(children)
```

Represents a region in the SPN.

Regions graphs are DAGs just like SPNs, but they do not represent sums and products. They are merely used for defining the scope structure of the SPN.

A RegionNode is a node with children, as opposed to a RegionVariable, which is a leaf node.

Parameters `children` (*list of RegionNode or RegionVariable*) – children of this node. Scope of the resulting node is the union of the scopes of its children. The scopes of these children must not overlap (pairwise disjoint)

```
libspn_keras.region_graph_to_dense_spn(region_graph_root, leaf_node,
                                       num_sums_iterable, return_weighted_child_logits,
                                       logspace_accumulators=False, accumulator_initializer=None,
                                       linear_accumulator_constraint=None, product_first=True,
                                       num_classes=None, with_root=True, sum_op=None)
```

Convert a region graph (built from *RegionNode* and *RegionVar*) to a dense SPN.

Parameters

- **region_graph_root** (*RegionNode*) – Root of the region graph
- **leaf_node** (*BaseLeaf*) – Node to insert at the leaf of the SPN
- **num_sums_iterable** (*Iterator[int]*) – Number of sums for all but the last root sum layer from bottom to top
- **logspace_accumulators** (*bool*) – Whether to represent accumulators of weights in logspace or not
- **accumulator_initializer** (*Optional[Initializer]*) – Initializer for accumulators
- **linear_accumulator_constraint** (*Optional[Constraint]*) – Constraint for linear accumulator, default: *GreaterThanEpsilon*
- **product_first** (*bool*) – Whether to start with a product layer
- **num_classes** (*Optional[int]*) – Number of classes at output. If *None*, will not use ‘latent’ sums at the end but will instead directly connect the root to the final layer of the dense stack. This means if set to *None* the SPN cannot be used for classification.
- **with_root** (*bool*) – If *True*, sets a *RootSum* as the final layer.
- **return_weighted_child_logits** (*bool*) – Whether to return weighted child logits.
- **sum_op** (*Optional[SumOpBase]*) – *SumOpBase* instance to use for sum layers.

Return type *Sequential*

Returns A Sum-Product Network as a *tf.keras.Sequential* model.

1.1.10 Visualization

For educational purposes, the following module can be used to visualize dense SPNs.

```
libspn_keras.visualize.visualize_dense_spn(dense_spn, show_legend=False,
                                           show_padding=True, transparent=False,
                                           node_size=30)
```

Visualize dense SPN.

The dense SPN must consist of *DenseSum*, *DenseProduct*, *RootSum* and leaf layers.

Parameters

- **dense_spn** (*Model*) – An SPN of type *tensorflow.keras.Sequential*

- **show_legend** (bool) – Whether to show legend of scopes and layers on the right
- **show_padding** (bool) – Whether to show padded nodes
- **transparent** (bool) – If True, the background is transparent.
- **node_size** (int) – Size of the nodes drawn in the graph. Adjust to avoid clutter.

Return type Figure

Returns A `plotly.graph_objects.Figure` instance. Use `.show()` to render the visualization.

DOCUMENTATION

The documentation of the library is hosted on [ReadTheDocs](#).

WHAT ARE SPNS?

Sum-Product Networks (SPNs) are a probabilistic deep architecture with solid theoretical foundations, which demonstrated state-of-the-art performance in several domains. Yet, surprisingly, there are no mature, general-purpose SPN implementations that would serve as a platform for the community of machine learning researchers centered around SPNs. LibSPN Keras is a new general-purpose Python library, which aims to become such a platform. The library is designed to make it straightforward and effortless to apply various SPN architectures to large-scale datasets and problems. The library achieves scalability and efficiency, thanks to a tight coupling with TensorFlow and Keras, two frameworks already in use by a large community of researchers and developers in multiple domains.

DEPENDENCIES

Currently, LibSPN Keras is tested with `tensorflow>=2.0` and `tensorflow-probability>=0.8.0`.

INSTALLATION

```
pip install libspn-keras
```


NOTE ON STABILITY OF THE REPO

Currently, the repo is in an alpha state. Hence, one can expect some sporadic breaking changes.

FEATURE OVERVIEW

- Gradient based training for generative and discriminative problems
- Hard EM training for generative problems
- Hard EM training with unweighted weights for generative problems
- Soft EM training for generative problems
- [Deep Generalized Convolutional Sum-Product Networks](#)
- SPNs with arbitrary decompositions
- Fully compatible with Keras and TensorFlow 2.0
- Input dropout
- Sum child dropout
- Image completion
- Model saving
- Discrete inputs through an `IndicatorLeaf` node
- Continuous inputs through `NormalLeaf`, `CauchyLeaf` or `LaplaceLeaf`. Each of these distributions support both univariate as well as *multivariate* inputs.

EXAMPLES / TUTORIALS

1. **Benchmark:** libspn-keras and Einsum Networks.
2. **Image Classification:** A Deep Generalized Convolutional Sum-Product Network (DGC-SPN).
3. **Image Completion:** A Deep Generalized Convolutional Sum-Product Network (DGC-SPN).
4. **Understanding region SPNs**
5. **Samping with convolutional SPNs**
6. More to come, and if you would like to see a tutorial on anything in particular please raise an issue!

Check out the way we can build complex DGC-SPNs in a layer-wise fashion:

```
import libspn_keras as spnk
from tensorflow import keras

spnk.set_default_sum_op(spnk.SumOpGradBackprop())
spnk.set_default_accumulator_initializer(
    keras.initializers.TruncatedNormal(stddev=0.5, mean=1.0)
)

sum_product_network = keras.Sequential([
    spnk.layers.NormalizeStandardScore(input_shape=(28, 28, 1)),
    spnk.layers.NormalLeaf(
        num_components=16,
        location_trainable=True,
        location_initializer=keras.initializers.TruncatedNormal(
            stddev=1.0, mean=0.0)
    ),
    # Non-overlapping products
    spnk.layers.Conv2DProduct(
        depthwise=True,
        strides=[2, 2],
        dilations=[1, 1],
        kernel_size=[2, 2],
        padding='valid'
    ),
    spnk.layers.Local2DSum(num_sums=16),
    # Non-overlapping products
    spnk.layers.Conv2DProduct(
        depthwise=True,
        strides=[2, 2],
        dilations=[1, 1],
        kernel_size=[2, 2],
        padding='valid'
    )
])
```

(continues on next page)

(continued from previous page)

```

),
spnk.layers.Local2DSum(num_sums=32),
# Overlapping products, starting at dilations [1, 1]
spnk.layers.Conv2DProduct(
    depthwise=True,
    strides=[1, 1],
    dilations=[1, 1],
    kernel_size=[2, 2],
    padding='full'
),
spnk.layers.Local2DSum(num_sums=32),
# Overlapping products, with dilations [2, 2] and full padding
spnk.layers.Conv2DProduct(
    depthwise=True,
    strides=[1, 1],
    dilations=[2, 2],
    kernel_size=[2, 2],
    padding='full'
),
spnk.layers.Local2DSum(num_sums=64),
# Overlapping products, with dilations [2, 2] and full padding
spnk.layers.Conv2DProduct(
    depthwise=True,
    strides=[1, 1],
    dilations=[4, 4],
    kernel_size=[2, 2],
    padding='full'
),
spnk.layers.Local2DSum(num_sums=64),
# Overlapping products, with dilations [2, 2] and 'final' padding to combine
# all scopes
spnk.layers.Conv2DProduct(
    depthwise=True,
    strides=[1, 1],
    dilations=[8, 8],
    kernel_size=[2, 2],
    padding='final'
),
spnk.layers.SpatialToRegions(),
# Class roots
spnk.layers.DenseSum(num_sums=10),
spnk.layers.RootSum(return_weighted_child_logits=True)
])

sum_product_network.summary(line_length=100)

```

Which produces:

Model: "sequential"

Layer (type)	Output Shape	Param #
normal_leaf (NormalLeaf)	(None, 28, 28, 16)	25088
conv2d_product (Conv2DProduct)	(None, 14, 14, 16)	4
local2d_sum (Local2DSum)	(None, 14, 14, 16)	50176

(continues on next page)

(continued from previous page)

conv2d_product_1	(Conv2DProd (None, 7, 7, 16))	4
local2d_sum_1	(Local2DSum) (None, 7, 7, 32)	25088
conv2d_product_2	(Conv2DProd (None, 8, 8, 32))	4
local2d_sum_2	(Local2DSum) (None, 8, 8, 32)	65536
conv2d_product_3	(Conv2DProd (None, 10, 10, 32))	4
local2d_sum_3	(Local2DSum) (None, 10, 10, 64)	204800
conv2d_product_4	(Conv2DProd (None, 14, 14, 64))	4
local2d_sum_4	(Local2DSum) (None, 14, 14, 64)	802816
conv2d_product_5	(Conv2DProd (None, 8, 8, 64))	4
spatial_to_regions	(SpatialT (None, 1, 1, 4096))	0
dense_sum	(DenseSum) (None, 1, 1, 10)	40960
root_sum	(RootSum) (None, 10)	10
=====		
Total params: 1,214,498		
Trainable params: 1,201,930		
Non-trainable params: 12,568		

TODOS

- Structure learning
- Advanced regularization e.g. pruning or auxiliary losses on weight accumulators

PYTHON MODULE INDEX

I

`libspn_keras.visualize`, [19](#)

A

`adapt()` (*libspn_keras.layers.NormalizeStandardScore* method), 10

C

`CauchyLeaf` (*class in libspn_keras.layers*), 4
`Clip` (*class in libspn_keras.constraints*), 18
`Conv2DProduct` (*class in libspn_keras.layers*), 8
`Conv2DSum` (*class in libspn_keras.layers*), 7

D

`DenseProduct` (*class in libspn_keras.layers*), 6
`DenseSum` (*class in libspn_keras.layers*), 5
`Dirichlet` (*class in libspn_keras.initializers*), 15
`DynamicSumProductNetwork` (*class in libspn_keras.models*), 11

E

`EpsilonInverseFanIn` (*class in libspn_keras.initializers*), 15
`Equidistant` (*class in libspn_keras.initializers*), 15

F

`FlatToRegions` (*class in libspn_keras.layers*), 5

G

`get_default_accumulator_initializer()` (*in module libspn_keras*), 13
`get_default_linear_accumulators_constraint()` (*in module libspn_keras*), 17
`get_default_logspace_accumulators_constraint()` (*in module libspn_keras*), 17
`get_default_sum_op()` (*in module libspn_keras*), 12
`GLOBAL` (*libspn_keras.layers.NormalizeAxes* attribute), 9
`GreaterEqualEpsilon` (*class in libspn_keras.constraints*), 18
`GreaterEqualEpsilonNormalized` (*class in libspn_keras.constraints*), 18

I

`IndicatorLeaf` (*class in libspn_keras.layers*), 4

K

`KMeans` (*class in libspn_keras.initializers*), 14

L

`LaplaceLeaf` (*class in libspn_keras.layers*), 4
`libspn_keras.visualize` module, 19
`Local2DSum` (*class in libspn_keras.layers*), 7
`LogDropout` (*class in libspn_keras.layers*), 9
`LogLikelihood` (*class in libspn_keras.metrics*), 17
`LogNormalized` (*class in libspn_keras.constraints*), 18

M

module
`libspn_keras.visualize`, 19

N

`NegativeLogJoint` (*class in libspn_keras.losses*), 16
`NegativeLogLikelihood` (*class in libspn_keras.losses*), 16
`NormalizeAxes` (*class in libspn_keras.layers*), 9
`NormalizeStandardScore` (*class in libspn_keras.layers*), 10
`NormalLeaf` (*class in libspn_keras.layers*), 3

O

`OnlineExpectationMaximization` (*class in libspn_keras.optimizers*), 16

P

`PoonDomingosMeanOfQuantileSplit` (*class in libspn_keras.initializers*), 14
`PoonDomingosStddevOfQuantileSplit` (*class in libspn_keras.initializers*), 15

R

`ReduceProduct` (*class in libspn_keras.layers*), 6

`region_graph_to_dense_spn()` (in module `libspn_keras`), 19

`RegionNode` (class in `libspn_keras`), 18

`RegionVariable` (class in `libspn_keras`), 18

`RootSum` (class in `libspn_keras.layers`), 6

S

`SAMPLE_WISE` (`libspn_keras.layers.NormalizeAxes` attribute), 9

`SequentialSumProductNetwork` (class in `libspn_keras.models`), 10

`set_default_accumulator_initializer()` (in module `libspn_keras`), 13

`set_default_linear_accumulators_constraint()` (in module `libspn_keras`), 17

`set_default_logspace_accumulators_constraint()` (in module `libspn_keras`), 17

`set_default_sum_op()` (in module `libspn_keras`), 12

`SpatialToRegions` (class in `libspn_keras.layers`), 8

`SumOpEMBackprop` (class in `libspn_keras`), 13

`SumOpGradBackprop` (class in `libspn_keras`), 12

`SumOpHardEMBackprop` (class in `libspn_keras`), 13

`SumOpSampleBackprop` (class in `libspn_keras`), 13

`SumOpUnweightedHardEMBackprop` (class in `libspn_keras`), 13

`SumProductNetwork` (class in `libspn_keras.models`), 10

T

`TemporalDenseProduct` (class in `libspn_keras.layers`), 9

V

`VARIABLE_WISE` (`libspn_keras.layers.NormalizeAxes` attribute), 9

`visualize_dense_spn()` (in module `libspn_keras.visualize`), 19

Z

`zero_evidence_inference()` (`libspn_keras.models.SequentialSumProductNetwork` method), 11