LibSPN: A Library for Learning and Inference with Sum-Product Networks and TensorFlow

Andrzej Pronobis [1 2] **Avinash Ranganath** [2] **Rajesh P. N. Rao** [1]

Abstract

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. Here, we present a new general-purpose Python library called LIBSPN, 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, a framework already used by a large community of researchers and developers in multiple domains. We describe the design and benefits of LIBSPN, give several use-case examples, and demonstrate the applicability of the library to real-world problems on the example of spatial understanding in mobile robotics.

1. Introduction

Over the last decade, advancements in deep learning have enabled unprecedented performance in a wide range of applications. Recently, Sum-Product Networks (SPNs) have emerged as a new promising architecture which combines the advantages of deep learning and probabilistic modeling. SPNs learn tractable probabilistic models directly from high-dimensional, noisy data. They are based on solid theoretical foundations and have been shown to provide state-of-the-art results in several different domains (Gens & Domingos, 2012; Peharz et al., 2014; Amer & Todorovic, 2016).

[1]Paul G. Allen School of Computer Science Engineering, University of Washington, Seattle, WA, USA. [2]Robotics, Perception and Learning Lab, KTH Royal Institute of Technology, Stockholm, Sweden. Correspondence to: Andrzej Pronobis <pronobis@cs.washington.edu>.

Presented at the ICML 2017 Workshop on Principled Approaches to Deep Learning, Sydney, Australia, 2017. Copyright 2017 by the author(s).

One of the primary factors hindering research on Sum-Product Networks and limiting their application to new domains is the lack of accessible, actively maintained software implementing general SPNs. It is the availability of software libraries such as Caffe (Jia et al., 2014), Theano (Theano Development Team, 2016), and TensorFlow (Abadi et al., 2016) that greatly accelerated the research on deep neural networks. Those libraries provide a framework for implementing new approaches with minimum effort, comparing and benchmarking existing algorithms, and finally, facilitate deployment of models to a wide range of datasets and scenarios. They help in bringing together researchers and developers actively contributing new techniques and code improvements.

In this paper, we present LIBSPN[1], a general-purpose Python library with the objective of serving similar purposes for the community of researchers and developers centered around Sum-Product Networks. LIBSPN is designed to accommodate various SPN architectures and offers a rich set of features, with additional features being actively developed. It offers a clean, object-oriented interface enabling quick prototyping and powerful customization. LIBSPN is tightly coupled with *TensorFlow* to achieve an efficient and scalable solution capable of utilizing multiple CPU and GPU devices. It is designed to be intuitive and familiar to existing *TensorFlow* users, and integrate seamlessly with the large repository of other machine learning techniques already implemented within *TensorFlow*.

LIBSPN was created out of the need to apply new SPN architectures to real-world problems in the domain of robotics (Pronobis & Rao, 2017). While several other software packages related to SPNs exist, they are often unmaintained, limited to a particular domain, data type or network architecture, or implemented as research code demonstrating a specific algorithm (Poon & Domingos, 2011; Gens & Domingos, 2013; Lowd & Rooshenas, 2015). Authors of (Zhao et al., 2016a) provide a C++ implementation of several batch and online parameter learning algorithms, but the code lacks GPU support and the ability to generate or learn SPN structure. A notable exception is Tachyon (Kalra, 2017), which also utilizes *TensorFlow*

[1]`http://www.libspn.org/`

for GPU computations and implements several different SPN learning algorithms. In comparison, LIBSPN offers a cleaner, and more extensible interface; realizes a broader set of network architectures and inferences; and provides a large toolbox for data processing and introspection. Moreover, LIBSPN ships with custom *TensorFlow* operations implemented directly in C++ and CUDA to address specific requirements of SPN architectures that cannot be fulfilled efficiently with native *TensorFlow* operations.

We begin by giving a short primer on SPNs and introducing a simple SPN model which serves as a running example throughout the paper. Then, we present the design and features of LIBSPN and give a simple usage example. Finally, we apply LIBSPN in practice to two problems: a toy problem illustrating the probabilistic nature of SPNs and a real-world problem in robotics demonstrating the ability of the library to perform learning and inference with noisy data.

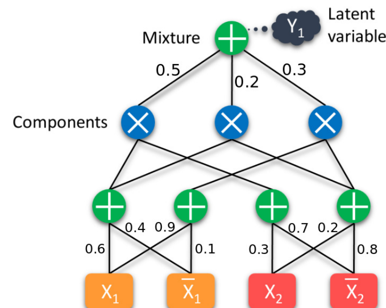


Figure 1. An SPN implementing a naive Bayes mixture model $P(X_1, X_2)$, with three components over two binary variables. The bottom layer consists of indicators for X_1 and X_2. Weighted sum nodes, with weights attached to inputs, are marked with $+$, while product nodes are marked with $\times$. Y_1 represents a latent variable (can be made explicit) marginalized out by the root sum.

2. Sum-Product Networks

SPNs are a recently proposed probabilistic deep architecture with several appealing properties and solid theoretical foundations (Peharz et al., 2015; Poon & Domingos, 2011; Gens & Domingos, 2012). One of the primary limitations of probabilistic graphical models is the complexity of their partition function, often requiring complex approximate inference in the presence of non-convex likelihood functions. In contrast, SPNs represent probability distributions with partition functions that are guaranteed to be tractable, and involve a polynomial number of sum and product operations, permitting exact inference. While not all probability distributions can be encoded by polynomial-sized SPNs, recent experiments in several domains show that the class of distributions modeled by SPNs is sufficient for many real-world problems, including speech (Peharz et al., 2014) and language modeling (Cheng et al., 2014), human activity recognition (Amer & Todorovic, 2016), image classification (Gens & Domingos, 2012), image completion (Poon & Domingos, 2011), and robotics (Pronobis & Rao, 2017). SPNs model joint or conditional distributions and can be learned generatively (Poon & Domingos, 2011) or discriminatively (Gens & Domingos, 2012) using Expectation Maximization (EM) or gradient descent. Additionally, several algorithms were proposed for simultaneous learning of network parameters and structure (Hsu et al., 2017; Gens & Domingos, 2013; Peharz et al., 2013). SPNs are a deep, hierarchical representation, capable of representing context-specific independence and performing fast, tractable inference on high-treewidth models.

As shown in Fig. 1, on a simple example of a naive Bayes mixture model, an SPN is a generalized directed acyclic graph composed of weighted sum and product operations. The sums can be seen as mixture models over subsets of variables, with weights representing mixture priors. The latent variables of such mixtures can be made explicit and their values inferred. Products can be viewed as features or mixture components. Not all possible architectures consisting of sums and products result in valid probability distributions and certain constraints (completeness and decomposability (Poon & Domingos, 2011; Peharz et al., 2015)) must be followed to guarantee validity. SPNs can be defined for both continuous and discrete variables, with evidence for categorical variables often specified in terms of binary indicators. Inference in SPNs is accomplished by an upwards pass which calculates the probability of the evidence and a downwards pass which obtains gradients for calculating marginals or MPE state of the missing evidence (in selective SPNs, see (Peharz et al., 2017)). The latter can be obtained by replacing sum operations with weighted max operations (the resulting network is sometimes referred to as Max-Product Network, MPN (Gens & Domingos, 2012)).

3. Concepts and Design

Let us now introduce the fundamental concepts and design principles of LIBSPN. LIBSPN is integrated with *TensorFlow* (Abadi et al., 2016), and is using it as a backend for distributing computations over multiple CPU and GPU devices. *TensorFlow* is an open-source library for numerical computation developed by the Google Brain team. In *TensorFlow*, a generic computation can be expressed as a static directed data flow graph, where each node represents an operation (a unit of computation) and the graph edge represents flow of a multi-dimensional array (tensor) from one node to another. Most operations in *TensorFlow* have both CPU and GPU implementations, and the library manages their deployment to the appropriate device. This enables execution of TensorFlow graphs on a wide variety of systems, ranging from mobile phones to clusters of ma-

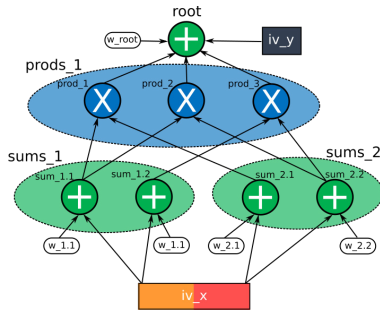


Figure 2. The *SPN graph* in LIBSPN corresponding to the SPN model in Fig. 1. The bottom layer consists of a single variable node `iv_x` representing multiple indicator variables. Above, the sum and product operations can be added to the *SPN graph* either as single-operation nodes (`prod_*` and `sum_*` nodes) or as blocks (layers) of multiple operations to increase compactness and efficiency (`prods_*` and `sums_*`). The variable node `iv_y` attached to the root node represents an explicit latent variable. Finally, parameter nodes `w_*` with sum weights are attached to each sum node.

chines with multiple GPUs. Initially, *TensorFlow* has been designed to support deep neural network models. As a result, several SPN-specific mathematical operations could not be realized efficiently with the library. To address this problem, and still benefit from the general infrastructure of *TensorFlow*, LIBSPN contributes C++ CUDA implementations of several custom *TensorFlow* operations.

The workflow of *TensorFlow* assumes two independent steps. First, a static graph of *TensorFlow* operations is created, which serves as an implementation of the computation. Then, the graph is executed inside a *session*, which manages the deployment of specific operations to specific devices. To address the limitations of the static *TensorFlow graph* and allow for higher-level, SPN-specific abstractions, LIBSPN introduces a dynamic *SPN graph*, a directed acyclic graph used to specify the structure of an SPN model. The dynamic nature of the *SPN graph* enables automatic generation, learning and pruning of the SPN structure. The *SPN graph* consists of three types of nodes corresponding to single or multiple instances of: random variables, SPN operations (e.g. sums, products, or nonlinearities), and parameters (e.g. sum weights). Representing parameters as part of the *SPN graph* permits weight sharing between multiple operations in an SPN. As is the case for the *TensorFlow graph*, edges in the *SPN graph* represent flow of tensors. Specifically, the first dimension of each tensor corresponds to a batch of data samples, while the remaining dimensions represent multiple values passed between the nodes. Such design makes it possible to feed results of any computation directly into the *SPN graph* (e.g. from data processing or other ML models). Fig. 2 shows a visualization of an *SPN graph* for the naive Bayes mixture model in Fig. 1.

In addition to the graph representing the model structure, LIBSPN provides algorithm classes, which implement various inference, learning and data processing algorithms. The role of the algorithm classes is to combine the model structure together with specific computations defined by the algorithms to generate a *TensorFlow graph* which is later executed in a *session*. This conversion process includes optimizations that result in a smaller and more efficient *TensorFlow graph*. Additionally, a set of high-level classes and a command-line interface facilitate creation of standard models for typical datasets. The high-level interface combines model structure generation, data (e.g. image) processing, execution of learning and inference algorithms as well as model loading and saving. LibSPN is a Python library following the OOP principles. Consequently, both the nodes of the graph and the algorithms are implemented using interfaces and classes that can be easily customized or extended.

4. Key Features of LibSPN

The goal of LIBSPN is to make it straightforward and effortless to apply SPNs to a wide range of applications and become the platform of choice for new algorithms and developments contributed by the machine learning community. To this end, LIBSPN offers the following advantages:
Generality: It is designed to be a general-purpose, domain-independent library applicable to many data types.
Simplicity: It offers a clean, simple, and well-documented Python interface. To facilitate prototyping, it is integrated with the Jupyter Notebook.
Familiarity: It relies on concepts and constructs familiar to the large community of existing users of TensorFlow.
Usability: It offers a set of high-level classes and a command-line interface for standard models and datasets.
Expressiveness: It aspires to accommodate a wide range of SPN architectures and inference/learning algorithms.
Efficiency: It leverages TensorFlow to efficiently perform parallel computations on (multiple) GPUs. It ships with custom TensorFlow operations implementing SPN-specific functionality directly in C++ and CUDA.
Scalability: It scales to datasets larger than the memory.
Introspectability: It complements the TensorFlow introspection tools with SPN-specific introspection and data visualization features integrated with Jupyter.
Flexibility: It integrates well with other software packages and the quickly growing repository of machine learning techniques implemented with TensorFlow.
Extensibility: The object-oriented interface of LIBSPN makes it easy to implement custom extensions to model architectures, algorithms, and datasets.
Reliability: The library comes with an extensive suite of tests and is actively maintained.

LIBSPN implements a wide range of features and additional features are being actively developed:

- Building custom networks
  - Building SPNs from single operations or blocks (layers) of operations
  - Support for categorical variables and continuous variables *[Partial]*
  - Realizing joint and conditional distributions
  - Realizing MPNs and SPNs
  - Creating custom operation nodes (e.g. nonlinearities)
  - Adding explicit latent variables
- Automatic SPN validity checking and scope calculation
- SPN structure generation
  - Generating dense random SPNs of varying complexity
  - Network pruning (Poon & Domingos, 2011)
  - Structure learning algorithms (Gens & Domingos, 2013; Vergari et al., 2015; Hsu et al., 2017) *[Under development]*
  - Hybrid models with convolutional neural networks *[Under development]*
- Inference
  - Marginal and MPE inference (for selective SPNs (Peharz et al., 2017))
  - Inferring values of explicit latent variables (often used for classification)
  - Sampling *[Under development]*
- Learning
  - Batch and online learning
  - Expectation-maximization learning (hard (Poon & Domingos, 2011) and soft (Peharz et al., 2017) *[Under development]*)
  - Gradient descent learning using TensorFlow optimization (Gens & Domingos, 2012) *[Under development]*
  - Weight sharing
- Saving and loading of trained SPNs
- SPN network visualization
- Helper classes and a command-line interface for building standard models for data classification and generation
- Helper classes for data handling
  - Data loading and saving (data can be larger than the memory) for standard datasets (images and CSV files)
  - Data batching and shuffling
  - Compatibility with the TensorFlow input pipeline
  - Generation of toy datasets
  - Basic data visualizations

We hope that contributions from the machine learning community will bring additional features, including other learning algorithms (Zhao et al., 2016b; Rashwan et al., 2016)).

5. Use Case Examples

We now illustrate the concepts behind LIBSPN, its interface, and several of its fundamental features in practice. To this end, we demonstrate how LIBSPN can be used to build a simple SPN model, learn its parameters from a dataset and perform inference.

Building the SPN Graph We begin be assembling a custom *SPN graph* on the example of the graph shown in Fig. 2, initially using nodes representing single sum and product operations:

```
1  import libspn as spn
2
3  iv_x = spn.IVs(num_vars=2, num_vals=2)
4  sum_11 = spn.Sum((iv_x, [0,1]))
5  sum_12 = spn.Sum((iv_x, [0,1]))
6  sum_21 = spn.Sum((iv_x, [2,3]))
7  sum_22 = spn.Sum((iv_x, [2,3]))
8  prod_1 = spn.Product(sum_11, sum_21)
9  prod_2 = spn.Product(sum_11, sum_22)
10 prod_3 = spn.Product(sum_12, sum_22)
11 root = spn.Sum(prod_1, prod_2, prod_3)
12 iv_y = root.generate_ivs()
13 spn.generate_weights(root,
     ↪  init_value=spn.ValueType.RANDOM_UNIFORM(0, 1))
```

Once the library is imported, we can simply create nodes in the graph by instantiating the classes representing node types. First, a single node `iv_x` representing four indicators for two binary variables X_1 and X_2 is added. Then, three layers of sum and product operations are assembled. Each operation node takes multiple values as input. If a child node outputs more than a single value, the parent node can select which values to use as input (as e.g. in line 4). Finally, we use two helper methods `generate_ivs` and `generate_weights` to add a variable node representing the explicit latent variable `iv_y` of the root sum, and the parameter nodes holding the weights of all sums (initialized with random values). Alternatively, these variable and parameter nodes can be created manually, and simply attached to the sum nodes. This can be used to create a graph in which weights are shared between multiple sums.

The same SPN model can also be realized using nodes implementing groups of sum or product operations:

```
1  iv_x = spn.IVs(num_vars=2, num_vals=2)
2  sums_1 = spn.ParSums((iv_x, [0,1]), num_sums=2)
3  sums_2 = spn.ParSums((iv_x, [2,3]), num_sums=2)
4  prods_1 = spn.PermProducts(sums_1, sums_2)
5  root = spn.Sum(prod_1, prod_2, prod_3)
```

Note that all nodes inherit from a common class `Node` which defines their generic interface. This interface can be used to introduce custom types of operations, variables or parameters.

Finally, we could simply employ a dense SPN generator to create the network:

```
1  gen = spn.DenseSPNGenerator(num_decomps=1,
     ↪  num_subsets=2, num_mixtures=2)
2  iv_x = spn.IVs(num_vars=2, num_vals=2)
3  root = gen.generate(iv_x)
```

The generator is aware of the scopes of its inputs and can be used to build very complex SPNs on top of any input.

Inspecting the Graph Once an *SPN graph* is created or generated, it can be inspected or visualized:

```
print(root.get_num_nodes()) # 15 (includes params)
print(root.get_scope())     # {iv_x:0, iv_x:1, iv_y:0}
assert root.is_valid()
spn.display_spn_graph(root)
```

Here, we print the number of nodes in the graph as well as its scope, i.e. the set of variables that appear in the graph (the scope in the example includes variables X_1, X_2, and Y_1). Scope can be calculated for any node in the *SPN graph* and is used internally to verify if the structure of the graph follows the principles of completeness and decomposability (see line 3). The function `display_spn_graph` can be used to visualize the *SPN graph* identified by its root.

Learning Given the initial graph structure, we can use one of the learning algorithm classes to generate *TensorFlow* operations performing learning of model parameters:

```
learner = spn.EMLearner(root, <learning_params>)
init_weights = spn.initialize_weights(root)
init_learning = learner.initialize()
learn = learner.learn()
likelihood = tf.reduce_mean(learner.root_value)
```

Once the *TensorFlow* operations are generated, they can be run inside a *session* that will distribute the computation over all available CPU and GPU devices:

```
# Training set
iv_x_arr = [[0,1], [1,1], ...]
iv_y_arr = [[-1]] * len(iv_x_arr)  # No evidence

# Learning
with spn.session() as (sess, _):
  sess.run(init_weights)
  sess.run(init_learning)
  for epoch in range(num_epochs):
    likelihood_arr, _ = sess.run([likelihood, learn],
 ↪   feed_dict={iv_x:iv_x_arr, iv_y:iv_y_arr})
    print("Avg. likelihood: %s" % (likelihood_arr))
```

In this example, the dataset is specified using simple arrays. This includes pairs of values[2] of the variables X_1 and X_2, as well as corresponding values of the variable Y_1 (value -1 indicates lack of evidence).

Using Large Datasets The *SPN graph* can use any tensor as a source of data for variable nodes. This enables integration with external models implemented with *TensorFlow*, as well as the use of the standard *TensorFlow* input pipeline. This mechanism is exploited by the dataset classes in LIBSPN, available for data types such as images, CSV files and certain custom datasets. These classes automatically load and shuffle batches of data during learning and permit the use of datasets larger than the available memory. The previous example can be modified to automatically load batches of data from a file:

[2] The `IVs` node converts an integer value of a variable to a set of indicators internally.

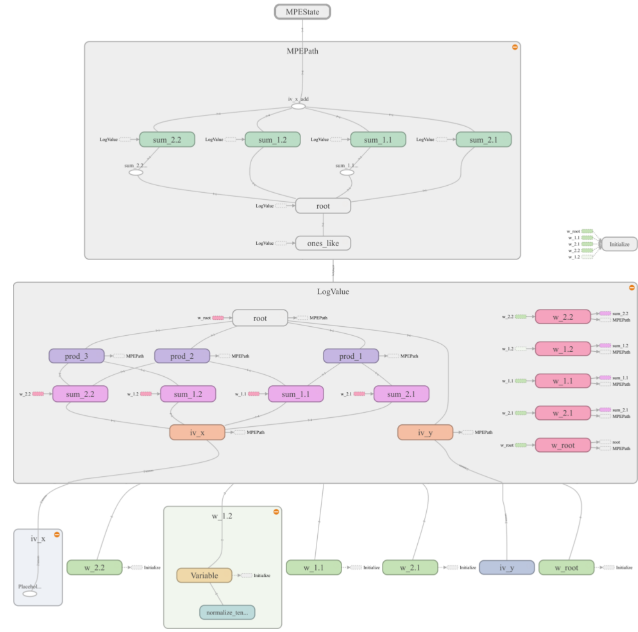


Figure 3. Visualization generated by TensorBoard, the *TensorFlow* introspection tool, illustrating the structure of the *TensorFlow graph* calculating the MPE state of variables in the *SPN graph* in Fig. 2. The *TensorFlow* operations are grouped to avoid clutter, with some groups expanded to illustrate their content. The bottom layer of the graph consists of nodes representing SPN parameters and inputs. The `LogValue` group contains the operations computing the log value of the SPN (upward pass). The `MPEPath` group calculates hard gradients (downward pass), with the `MPEState` group extracting the MPE state of SPN variables from the hard gradients.

```
dataset = spn.CSVFileDataset('data.csv',
 ↪   num_epochs=10, batch_size=10, shuffle=True, ...)
samples, labels = dataset.get_data()
iv_x.attach_feed(samples)
iv_y.attach_feed(labels)

with spn.session() as (sess, run):
  sess.run(init_weights)
  sess.run(init_learning)
  try:
    while run():
      sess.run(learn)
  except tf.errors.OutOfRangeError:
    print("Done!")
```

Making Inferences Finally, we can make a wide range of inferences using the learned model. First, we can simply calculate the value of the SPN or MPN for (partial) evidence. This corresponds to calculating the joint/marginal probability or probability of the MPE state. As in the case of learning, we begin by creating *TensorFlow* operations performing the calculation:

```
marginal_val = root.get_value(
 ↪   inference_type=spn.InferenceType.MARGINAL)
mpe_val = root.get_value(
 ↪   inference_type=spn.InferenceType.MPE)
```

Here, the parameter `inference_type` is used to specify whether sum (SPN) or max (MPN) operations are used during the upward pass through the network. The parameter is useful also during learning to control the way latent variables are inferred. The resulting *TensorFlow graph* will be similar to the upward pass *TensorFlow* graph shown in Fig. 3. Then, we can run the inference operations in a *session* for specific values of the variables:

```
1 iv_x_arr = [[0, 1], [0, -1], [-1, -1]]
2 iv_y_arr = [[0],    [-1],    [-1]]
3
4 with spn.session() as (sess, _):
5     marginal_val_arr = sess.run(marginal_val,
  ↪   feed_dict={iv_x: iv_x_arr, iv_y: iv_y_arr})
6     mpe_val_arr = sess.run(mpe_val, feed_dict={iv_x:
  ↪   iv_x_arr, iv_y: iv_y_arr})
7
8 print(marginal_val_arr) # 0.06, 0.31, 1.0
9 print(mpe_val_arr)      # 0.06, 0.14, 0.216
```

Specifying -1 as the value of a variable marginalizes the variable out in case of marginal inference. In particular, the last sample in the three-sample batch in the above example[3] marginalizes out all the variables, effectively calculating the value of the partition function. The resulting *TensorFlow graph* is illustrated in Fig. 3.

We can now calculate the MPE state for the missing evidence:

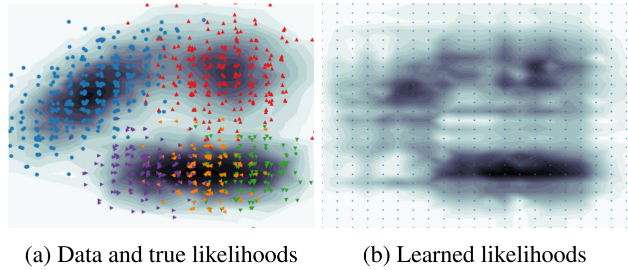
```
1 mpe_state = spn.MPEState()
2 iv_x_mpe, iv_y_mpe = mpe_state.get_state(
  ↪   root, iv_x, iv_y)
3
4 with spn.session() as (sess, _):
5     iv_x_mpe_arr, iv_y_mpe_arr = sess.run(
  ↪   [iv_x_mpe, iv_y_mpe],
  ↪   feed_dict={iv_x: iv_x_arr, iv_y: iv_y_arr})
6
7 print(iv_x_mpe_arr)  # [[0, 1], [0, 0], [1, 0]]
8 print(iv_y_mpe_arr)  # [[0],     [0],     [2]]
```

For the same variable assignment as in the previous example, we get the MPE state of X_2 and Y_1 for $X_1 = 0$, as well as the MPE state of all variables. Latent variables, such as Y_1, are often used to represent the class label in classification models. In such case, MPE inference corresponds to classification.

Command-line Interface Besides the object-oriented interface, the library provides a set of command-line tools. These tools are designed to facilitate data processing, learning and inference for typical datasets and use cases. In particular, the `spn-model` script offers a set of commands, e.g. `load` for loading a model, `build` for building the initial SPN structure, `train` for learning a model, and `test` for making inferences, including classification and data sample generation. Each command accepts a series of command-line options. For example, `train` requires specifying the dataset type, the type of the learning algorithm and the learning parameters. The commands can be chained with their options loaded from a YAML file, e.g.:

```
1 spn-model -c "data_params.yaml" -c "model_params.yaml"
  ↪   build train save "model.spn"
```

[3]The first dimension of `iv_x_arr` and `iv_y_arr` corresponds to batch samples.



(a) Data and true likelihoods (b) Learned likelihoods

Figure 4. (a) Training data samples together with their true likelihoods. The samples have discrete values, and are jittered on the plot to illustrate their concentration. (b) Likelihoods for all values of the discrete variables X_1 and X_2 obtained from a learned model.

6. Applications and Evaluation

In this section, we apply LibSPN to two problems. First, we use an illustrative toy example to demonstrate the ability of LibSPN to learn a probability distribution from data samples. Then, we apply LibSPN to a real-world problem in robotics, which requires a custom SPN architecture and multiple types of real-time inferences based on noisy sensory data captured by a mobile robot.

6.1. Learning Distributions

We begin by tasking LibSPN with the problem of learning a probability distribution over two discrete variables from a dataset of samples. The samples were originally drawn from a Gaussian mixture distribution with 5 components, and then discretized to take 30 values. The samples as well as their true likelihoods are illustrated in Fig. 4a.

To generate the initial structure of the model, we used the dense SPN generator included with the library. We set the parameters of the generator to obtain a naive Bayes mixture SPN, similar to the example in Fig. 1, but with a larger number of sum nodes over the indicator variables for the variables X_1 and X_2. We used the same learning procedure as in the following experiment with real-world data. We began with an over-complete model (including 40 sums per variable) and proceeded with hard EM learning from the data samples[4]. We terminated learning after 50 epochs,

[4]When inferring latent variables, we used sums for the upward pass and maxes for the downward pass. As a result, the value of each latent variable is computed conditioning on the MPE values of variables above it and marginalizing out the variables below it.

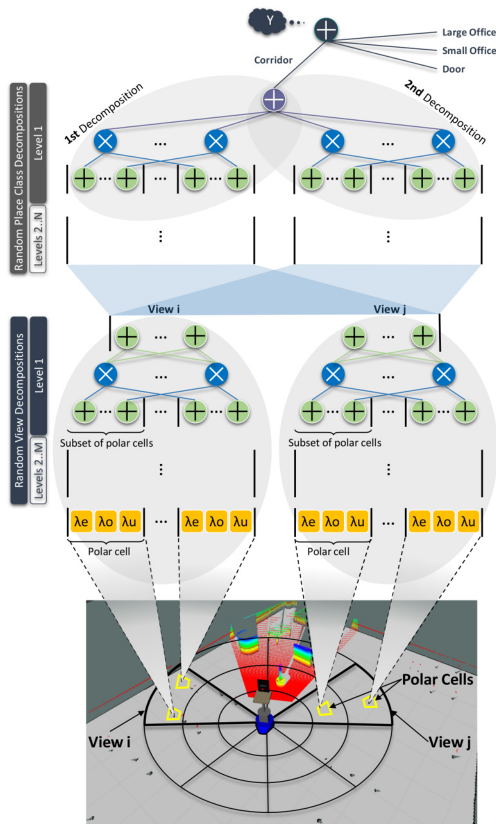


Figure 5. The structure of the SPN implementing the model used for spatial understanding. The bottom images illustrate a robot in an environment and a robocentric polar grid formed around the robot. The SPN is built on top of indicator variables (orange) representing the occupancy in each cell of the polar grid (one for empty, occupied and unknown space). Explicit latent variable Y capturing the place class is shown in the gray callout.

and pruned the network to eliminate edges associated with weights close to zero. The resulting network included only 31 out of the initial 80 sum nodes defined over the input indicators. The pruning can be seen as a simple form of structure learning.

The likelihoods generated by the trained model for each value of the discrete input variables are shown in Fig. 4b. It is evident that the model learns a distribution resembling the one from which the training samples were drawn. This demonstrates the ability of the library to build probabilistic models capturing distributions with complex shapes.

6.2. Spatial Understanding in Robotics

In our next experiment, we aimed at validating the applicability of the library to realistic problems involving high-dimensional noisy data. Our domain of choice was mobile robotics and the problem was defined as one of modeling the space around a robot navigating through a large office building. Below, we briefly summarize the experimental setup, the architecture of the model and the obtained results. For additional details, the reader is referred to (Pronobis & Rao, 2017).

As training data, we used sensory observations captured by the robot using a laser-range finder. The observations were first integrated using a local occupancy-grid mapping algorithm. The algorithm performed spatio-temporal integration of the sensed distance to obstacles and represented the information in terms of grid cells that are either empty, occupied or unknown. The resulting local maps describe the geometry of the environment surrounding the robot. These Cartesian grid maps were then converted to a polar representation (examples of such polar occupancy grids are shown in Fig. 7). The resulting grids contain higher-resolution details closer to the robot and lower-resolution information further away, a useful property for both spatial understanding and action planning in this domain.

The architecture of the SPN model we built with LIBSPN is shown in Fig.5. It represents a probability distribution over the occupancy information in the polar grid and the semantic category of the place at which the robot was positioned when the data was captured. The structure of the model is partially static and partially generated randomly. The resulting model is a single SPN, which is assembled from three levels of sub-SPNs. We begin by splitting the polar grid equally into 8 *views* (45 degrees each). For each view, we use a random SPN generator to recursively build a hierarchy of distributions for subsets of polar cells. Then, on top of all the sub-SPNs representing the views, we randomly generate an SPN representing complete place geometries for each place class. The sub-SPNs for place classes are combined by a sum node forming the root of the network. The latent variable associated with the root node is made explicit and is set to the appropriate class label during learning.

The model was trained using hard EM from polar grids captured on several floors of an office building and tested on samples collected on another floor. We tasked the learned model with several inferences. As the first inference problem, we chose classification, where the goal was to infer the semantic place class given an observed polar grid. As a baseline, we used a well-established model based on SVM and geometric features extracted from laser scans (Pronobis et al., 2010). To ensure the best SVM result, we used the RBF kernel and selected the kernel and learning parameters directly on the test set. The polar grids used for training were collected in places belonging to four different classes: a large office, a small office, a corridor and a doorway. The classification rate averaged over all classes (giving equal importance to each class) was $85.9\% \pm 5.4$ for SVM and $92.7\% \pm 6.2$ for SPN.

Next, to evaluate the quality of the learned likelihood val-

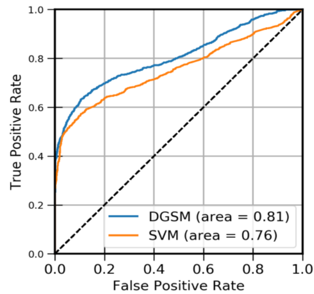
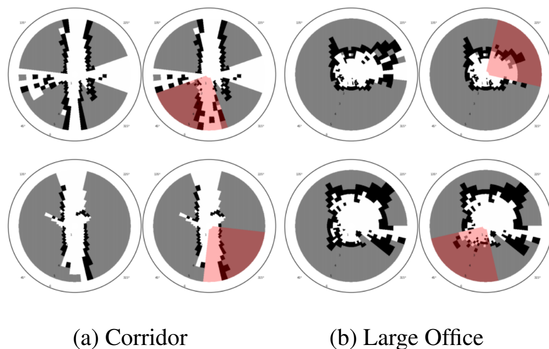


Figure 6. ROC curves for novelty detection. Inliers are considered positive, while novel samples are negative.

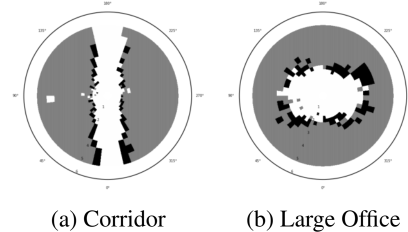


(a) Corridor (b) Large Office

Figure 7. Examples of completed polar grids with masked data for two out of the four place categories. The left image shows original data, and the right image shows reconstructed data within the shaded area.

ues, we evaluated the model for the task of novelty detection. We used the same trained model as for classification, and used the likelihood of a polar grid (the value of the root node of the SPN) to determine whether or not the grid was captured in a place belonging to one of the four categories available during training. The model was tested on previously unseen examples from the known classes as well as examples captured in places belonging to novel classes. As a baseline, we used a 1-class SVM trained as in the previous experiment. The cumulative ROC curve for the detection task is shown in Fig. 6. Here, again, SPN performed significantly better, with AUC of 0.81 compared to 0.76 for SVM.

The next two inferences evaluated the generative abilities of the model. First, we used the trained SPN to generate missing observations in partially masked polar grids. We masked a random 90-degree view in each test polar grid (25% of the grid cells) and inferred the masked values. All indicators for the masked polar cells as well as the class latent variable were set to 1 to indicate missing evidence, and MPE inference followed. Fig. 7 shows examples of completed polar grids. Overall, when averaged over all test examples and data splits, SPN correctly reconstructed $77.14\% \pm 1.04$ of masked cells. This time as a baseline we used Generative Adversarial Networks, which for a similar setup recovered $75.84\% \pm 1.51$ of masked cells.



(a) Corridor (b) Large Office

Figure 8. Prototypical polar grids inferred based on the semantic place class.

Finally, we tasked the network with generating complete, prototypical representations of places based only on the semantic category. This time, the value of the latent class variable was set, and the MPE state of the variables representing occupancy was inferred. The generated polar occupancy grids are shown in Fig. 8. Comparing it with the true data samples shown in Fig. 7), we can see that each prototype is very characteristic of the class from which it was generated.

7. Conclusions

This papers presents LIBSPN, a general-purpose library for inference and learning with Sum-Product Networks, designed to facilitate application of various SPN architectures to large-scale datasets and problems. Through experiments, we have demonstrated the potential of LIBSPN in real-world applications. Compared to models such as SVMs and Generative Adversarial Networks, the SPN implemented with LIBSPN offered superior performance for the tasks of classification and novelty detection, and comparable generative potential. Importantly, for robotic applications, all inferences performed using LIBSPN were significantly faster than required for real-time operation.

The library is currently in active development, and its developers are continuously extending the set of available features. The features in active development include complete support for gradient descent learning, generation and learning of several new SPN architectures as well as integration with convolutional models.

We hope that the efficiency, scalability and universal design of the library will make it the platform of choice for implementing new SPN algorithms contributed by the machine learning community, and will open doors for many new applications of Sum-Product Networks.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) grant no. N00014-13-1-0817 and the Swedish Research Council (VR) project 2012-04907 SKAEENet.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *preprint arXiv:1603.04467*, 2016.

Amer, Mohamed and Todorovic, Sinisa. Sum-Product Networks for activity recognition. *Transactions on Pattern Analysis and Machine Intelligence*, 38(4):800–813, 2016.

Cheng, Wei-Chen, Kok, Stanley, Pham, Hoai Vu, Chieu, Hai Leong, and Chai, Kian Ming A. Language modeling with Sum-Product Networks. In *Proceedings of Interspeech*, 2014.

Gens, Robert and Domingos, Pedro. Discriminative learning of Sum-Product Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

Gens, Robert and Domingos, Pedro. Learning the structure of Sum-Product Networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2013. Code available at http://spn.cs.washington.edu/learnspn/.

Hsu, Wilson, Kalra, Agastya, and Poupart, Pascal. Online structure learning for Sum-Product Networks with Gaussian leaves. *preprint arXiv:1701.05265*, 2017.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. *preprint arXiv:1408.5093*, 2014.

Kalra, Agastya. Tachyon, 2017. Code available at https://github.com/KalraA/Tachyon.

Lowd, Daniel and Rooshenas, Amirmohammad. The libra toolkit for probabilistic models. *Journal of Machine Learning Research*, 16:2459–2463, 2015. Code available at http://libra.cs.uoregon.edu/.

Peharz, Robert, Geiger, Bernhard C, and Pernkopf, Franz. Greedy Part-Wise learning of Sum-Product Networks. In *Machine Learning and Knowledge Discovery in Databases*, Lecture Notes in Computer Science, pp. 612–627. Springer Berlin Heidelberg, 2013.

Peharz, Robert, Robert, Peharz, Georg, Kapeller, Pejman, Mowlaee, and Franz, Pernkopf. Modeling speech with Sum-product Networks: Application to bandwidth extension. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

Peharz, Robert, Tschiatschek, Sebastian, Pernkopf, Franz, and Domingos, Pedro. On theoretical properties of Sum-Product Networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2015.

Peharz, Robert, Gens, Robert, Pernkopf, Franz, and Domingos, Pedro. On the latent variable interpretation in Sum-Product Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

Poon, Hoifung and Domingos, Pedro. Sum-Product Networks: A new deep architecture. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011. Code available at http://spn.cs.washington.edu/spn/.

Pronobis, Andrzej and Rao, Rajesh P. N. Learning deep generative spatial models for mobile robots. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2017.

Pronobis, Andrzej, Mozos, Oscar M., Caputo, Barbara, and Jensfelt, Patric. Multi-modal semantic place classification. *International Journal of Robotics Research*, 29(2-3), February 2010.

Rashwan, Abdullah, Zhao, Han, and Poupart, Pascal. Online and distributed Bayesian moment matching for parameter learning in Sum-Product Networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *preprint arXiv:1605.02688*, 2016.

Vergari, Antonio, Di Mauro, Nicola, and Esposito, Floriana. Simplifying, regularizing and strengthening Sum-Product Network structure learning. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2015.

Zhao, Han, Adel, Tameem, Gordon, Geoff, and Amos, Brandon. Collapsed variational inference for Sum-Product Networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016a. Code available at http://www.cs.cmu.edu/~hzhao1/papers/ICML2016/spn_release.zip.

Zhao, Han, Poupart, Pascal, and Gordon, Geoff. A unified approach for learning the parameters of Sum-Product Networks. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, 2016b.