# Understanding Sum Product Networks: A Theoretical-Based Guide with Practical Implementation


## Abstract

Sum-Product Networks (SPNs) reduce complexity of computation in various aspects of machine learning. They can not only be used as an enhancement of the partition function in many graph models, but they can also be used to represent complex probability distributions and perform inference and learning compactly. Partition functions are often a bottleneck in graphical model inference. Hence, SPNs can be used as a solution to reduce the complexity of normalization and computation of marginals and likelihoods. The purpose of this report is to analyze and demonstrate the architecture of SPNs and to show their implementation through image datasets.


## 1. Introduction

### 1.1. What is a Sum-Product Network?

Sum-Product Networks (SPN) are generative models that are very different from your average neural network. It is a directed acyclic graph with leaves of the network representing variables, and the nodes in the hidden layer representing Sum or Product functions.

These Sum and Product nodes almost act as activation functions. We log our result from one Sum node and use exponent to convert to the Product node alternatively. Mathematically, Having a bunch of sums and products is the same as having a neural network where you alternate between logarithm and exponential function layers.

There exist a lot of techniques for training SPNs other than simply gradient descent. The Expectation maximization (EM) algorithm can be utilized and possibly cause a super linear convergence better than gradient descent.


. Correspondence to: Anonymous Author <anon.email@domain.com>.



Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.


### 1.2. Importance and Applications

Algorithms exist to estimate optimal SPN graphical structures. Like Bayesian and Markov Networks, SPNs represent joint distributions over a set of variables and encode correlations based on node connections, proving successful for modeling. However, traditional inference can be intractable, falling into the Sharp P class of complexity. NP hardness refers to the fact that there exists problem instances that might take at least exponential time to be solved. Sharp P is even harder than NP, since, in NP, we are solving for a satisfiable solution, whereas in Sharp P, we are counting all the number of satisfiable solutions that exist for a given problem. When it comes to inference in PGAs (Probabilistic Graphical Models), we are counting all the joint assignments with their weights. The probabilities then represent the sum of the probabilities of all of the satisfying assignments. Clearly, this method involves complex counting, therefore approximation is necessary. The beauty of SPNs lies in their tractable inference, which can be achieved in linear time, making them a powerful tool in machine learning. Experiments show that inference and learning with SPNs can be both faster and more accurate than with standard deep networks. For example, SPNs perform image completion better than state-of-the-art deep networks for this task. SPNs also have intriguing potential connections to the architecture of the cortex (Poon & Domingos, 2011).

An additional advantage of SPNs is their ability to prevent under/overflow. By logging small decimal numbers, which are converted to large negative values, the model avoids interpreting these numbers as zero, thus maintaining stability. While not its primary function, this feature improves the SPN's quality.

Furthermore, SPNs can be:

- Converted to Bayesian Networks (BN) or Markov Networks (MN) without exponential blow-up. However, converting from BN/MN to SPN can cause an exponential increase in complexity.
- Great for parallel computing as they can compute all requested queries simultaneously in both directions. This means they can handle large datasets efficiently.
- Utilized by collecting all the data and then calculating

similarities, such as cosine similarity, which measures the angle between two vectors, unlike many machine learning models that require the data to follow a normal distribution.

## 2. Background and Theory

### 2.1. Architecture & Mathematical Principles

As stated previously, SPNs are a special type of deep learning model. Mathematically, an SPN is a rooted directed acyclic graph, where leaf nodes represent input variables, and internal nodes represent sums and products.

Simply put:

- **Sum Nodes:**
  - Calculate the weighted sum of their child nodes. The weights represent the probability of each child node.
  - A sum node S in an SPN with child nodes

    $$C_1, C_2, ..., C_n$$

    and corresponding weights

    $$w_1, w_2, ..., w_n$$

    calculates its value as follows:

    $$S = w_1C_1 + w_2C_2 + \ldots + w_nC_n$$

    Note that the weights must be **normalized** so that they sum up to 1.
- **Product Nodes:**
  - Calculate the product of their child nodes, representing the joint probability of their child nodes.
  - A product node P with child nodes

    $$C_1, C_2, ..., C_n$$

    calculates its value as follows:

    $$P = C_1 * C_2 * ... * C_n$$

An SPN represents a joint probability distribution where the value of the root node gives the probability of the input data under this distribution. Due to this structure, calculating the marginal probability is performed in linear time through the bottom-up pass starting from the leaves to the root.

In Figure 1, the SPN is implementing a Naive Bayes Mixture Model. For simplicity, we assume that the input variables are Boolean (i.e., they can take one of two values, 0 or 1) and that the sum and product nodes are arranged in alternating order.

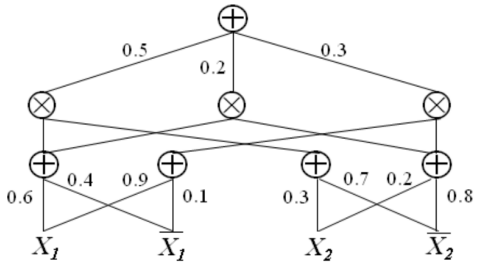


*Figure 1.* SPN implementing a naive Bayes mixture model (three components, two variables) (Poon & Domingos, 2011)

The **network polynomial** is the function represented by the SPN at the root of the tree, which is a combination of the sums and products of the nodes, making it multi-linear.

The network polynomial of an unnormalized probability distribution $\Phi(x)$ is given by: $\sum_x \Phi(x) \cdot \Pi(x)$.

This means that for each possible state x, we calculate $\Phi(x) \cdot \Pi(x)$, where $\Phi(x)$ is the unnormalized probability of that state and $\Pi(x)$ is the product of the indicator variables that have value 1 in that state. The indicator variables represent the state of the input variables. In the example shown in Figure 1., the indicator variables are demonstrated in the context of boolean variables.

Then, the total network polynomial is calculated through the summation of these products for all possible states of x, ending up with a single polynomial that represents the entire SPN and can be used to calculate probabilities for any given evidence.

When it comes to boolean indicator variables, evidence refers to an incomplete or complete assignment of these variables. If certain variables are not assigned a value, this means that they are not part of the evidence and will be assigned a default value. For example, $\Phi(X_1 = 1, X_3 = 0)$ is the value of the network polynomial when $X_1$ and $X_3$ are set to 0 and the remaining indicators are set to 1 throughout (Poon & Domingos, 2011). This is very typical in SPNs with boolean inputs.

Finally, the "partition function" is the total sum of all possible joint assignments of values to the variables of the network, which consists of setting all indicators to 1. This is used for the process of normalization, where all the probabilities are able to be summed up to 1. Hence, one can infer that the need for a separate partition function is not needed any more, as this calculation is built into the network structure in a very efficient manner.

### 2.2. Learning Process

In many traditional neural networks, weights are usually adjusted using gradient descent. However, in SPNs, the Expectation Maximization (EM) algorithm is typically used for weight adjustment and learning.

The EM algorithm is very useful, because it can handle missing data and hidden variables well. It works by iteratively estimating the probabilities (the Expectation step) and then optimizing the parameters to maximize the likelihood of the data given these estimated probabilities (the Maximization step).

Note that SPNs are not just limited to the EM algorithm. Gradient descent can also be utilized for weight adjustment in SPNs, especially when dealing with discriminative learning tasks. The choice between both learning algorithms depends on the specific task and structure at hand.

**Algorithm 1** LearnSPN

**Input:** Set $D$ of instances over variables $X$.
**Output:** An SPN with learned structure and parameters.
$S \leftarrow$ GenerateDenseSPN($X$)
InitializeWeights($S$)
**repeat**
  **for all** $d \in D$ **do**
    UpdateWeights($S$, Inference($S, d$))
  **end for**
**until** convergence
$S \leftarrow$ PruneZeroWeights($S$)
**return** $S$

*Figure 2.* A general learning scheme with online learning on weights (Poon & Domingos, 2011)

This pseudocode shown in Figure 2. shows the LearnSPN algorithm, which utilizes the Expectation Maximization (EM) algorithm to learn the structure and parameters of an SPN.

Here is a breakdown of the code:

- GenerateDenseSPN(X):
  - This function generates an initial dense SPN structure over the variables, X, as a generic structure. A dense SPN is a network where each node is connected to every other node.
- InitializeWeights(S):
  - This function initializes the weights of the SPN, S. This could be done randomly or by using some other methods.
- Repeat until convergence:
  - This loop repeats until the weights stop changing significantly, indicating that the learning process has converged.
- UpdateWeights(S, Inference(S,d)):
  - For each instance, d, in the dataset, D, this function updates the weights of the SPN, S, based on the result of the Inference function. The Inference function makes a forward pass through the SPN, calculating the probabilities of the evidence in instance, d.
- PruneZeroWeights(S):
  - This function removes connections in the SPN, S, that have a weight of 0, simplifying the network.

Finally, the learned SPN, S, is returned.

The Inference function corresponds to the E-step of EM, and the UpdateWeights function corresponds to the M-step of the EM algorithm.

### 2.3. Pros & Cons

*Table 1.* Pros & Cons of SPNs

| Pros | Cons |
|---|---|
| Efficient Inference | Complex for Large Datasets |
| Handles Missing Data | Overfitting |
| Versatile (used for both generative & discriminative learning) | Inference for Continuous variables can be more challenging |

## 3. Implementation

This report showcases the application of SPNs through image classification tasks. Two main libraries were used in the implementation of the SPN: TensorFlow and the SPN library, libspn-keras.

### 3.1. TensorFlow Framework

TensorFlow is popular open source machine learning framework developed by Google. It is mainly used for building and training models for machine learning, like neural networks.

One of its strengths lies in its ability for automatic differentiation and optimization, making it very suitable for developing complex structures like SPNs.

### 3.2. libspn-keras

libspn-keras is a library that provides higher level abstractions when it comes to building Sum-Product-Networks

(SPNs) using Keras and TensorFlow. It is a tool designed specifically for building and training SPNs. This library helped advance the process of development, as many of the details of the sum and product nodes were handled by this library. This increased the focus on more higher level designs of the network. Without this library, the sum and product nodes, as well as the majority of the SPN's architecture, would require to be developed manually.

### 3.3. The Dataset

In this implementation, the "mnist" dataset is loaded from the TensorFlow database.

The MNIST dataset is a large database of handwritten digits that is commonly used for training various image processing systems. It contains 60,000 training images and 10,000 testing images, where each is a 28x28 grayscale image of a digit between 0 and 9.

### 3.4. SPN Implementation

Here are the steps of the implementation:

1. Installation: Install the required libraries, sch as libspn-keras and TensorFlow using pip.
2. Set the default operations: this defines the default behaviour of the SPN's nodes during the forward and backward pass during learning, allowing for consistency across the SPN structure.
3. Data Preparation: Load and normalize the training data from the mnist dataset which is loaded from tensorflowdatasets.
4. Defining the SPN model: Build the model using a Sequential model from Keras. Create the:
    - NormalLeaf layers: the distributions over the leaf input features.
    - Conv2DProduct Layers: Captures the spatial relationships between the features in the image input data through the combination and product of features.
    - Local2DSum Layers: perform summation in local regions of the output feature map. By doing this for each region of the image, the layer creates a new grid consisting of numbers that are the sums of the local regions. This helps in capturing patterns in different parts of the input data.
    - Strides: control the convolutional kernel movements across the input data. Larger strides reduce the size of the output, but smaller strides cover more information in the input.
    - Dilations: aids the model in viewing different parts of the input data with varying levels of detail.
    - Padding: adds borders around the pictures, ensuring that the output size remains the same as the input size after convolutional operations are complete.
5. Data Shuffling: Load and shuffle the MNIST data into batches for both training and testing.
6. Compilation: Compile the SPN model.
7. Training: Train the SPN model on the defined training dataset for 15 epochs.
8. Testing: Test and evaluate the SPN model on the testing dataset.
9. Save and Evaluate: Save the weights and check accuracy levels.

## 4. Experimentation

### 4.1. Hypothesis

#### 4.1.1. Hypothesis 1:

Effectiveness of SPNs for MNIST Classification: My hypothesis suggests that SPNs, with their unique architecture, can effectively and accurately classify handwritten digits from the MNIST dataset.

### 4.2. Results

#### 4.2.1. Results of Hypothesis 1:

The results of my experiment conclude that the SPN was not sufficient enough to accurately predict the classifications of the numbered image data from MNIST dataset. This can be seen through the results of the training and testing functions, where the accuracy is very low. The model's accuracy percentage of the predictions for the testing dataset was about 23.5%, and 23.9% for the training dataset. This means that the model correctly classified only 23.54% of the images in the testing set.

## 5. Analysis

Evidently, the accuracy levels resulting from the experiment are not very good accuracy's. A good accuracy is typically around 95%. These inaccurate predictions can be due to a number of reasons and factors:

1. Low number of epochs: A low number of epochs may result in under-training the model. For the sake of time and due to GPU limitations, only 15 epochs were used in the training phase of this experiment/implementation. However, a good rule of thumb is to train a model for at least 100 epochs. You

can then monitor the training and validation loss and stop training when the validation loss starts to increase. A number of of research papers have suggested for 100 epochs to be a good starting point for training deep learning models, particularly in the paper "Deep Residual Learning for Image Recognition" by He et al (2015)., where

2. Low number of Dilations: Due to GPU limitations, the Dilations in this experiment were also set to [1,1]. It is possible that reducing the Dilations to [1,1] could be reducing the performance of the SPN. To explain further, a kernel is a matrix of weights that acts like a filter that slides over the input data to perform some convolution operation when working with images. As stated in the SPN Implementation section, Dilations refer to the spacing between the values in a kernel when it is applied to the input. A Dilation of [1,1] would mean that the kernel is applied to the input data without skipping anything. Therefore, the receptive field of the layers would the same size as the kernel and each neuron in the layer will only see a small and localized region of the input space, limiting the ability of the SPN model to learn more complex patterns. Access to more powerful GPU and training the SPN with larger Dilations could potentially improve the performance of the model's predictions.

## 6. Insights

Through the implementation and experiment of the SPN, it can be inferred that increasing the number of epochs to give your model more training time, and increasing the Dilations for larger scope of the input data, can possibly lead to much greater accuracy, not only in image classification, but also across a wide range of applications, when utilizing and training an SPN.

### 6.1. SPNs and Other Models

Generally, SPNs are more compact and efficient than both, Junction trees and Mixture models. SPNs excel even when the Junction trees are formed from Bayesian networks with context-specific independence in the form of decision trees at the nodes. This is because decision trees perform too many replications resulting in an exponential growth, which is much larger than the DAG (Directed Acyclic Graph) representation of the same graph in an SPN. Hence, SPNs can represent the probability functions much more compactly than Junctions trees. Furthermore, Mixture models also perform inference in exponential time. In Figure 3., the structure of the Mixture model represents the probability distribution as a weighted sum of multiple component distributions which means that the time it takes to make a prediction grows exponentially with the number of variables. In contrast, as seen in Figure 3., SPNs consist of both sum and product nodes, and therefore perform inference in linear time, which means that the time it takes to make a prediction grows linearly with the number of variables.

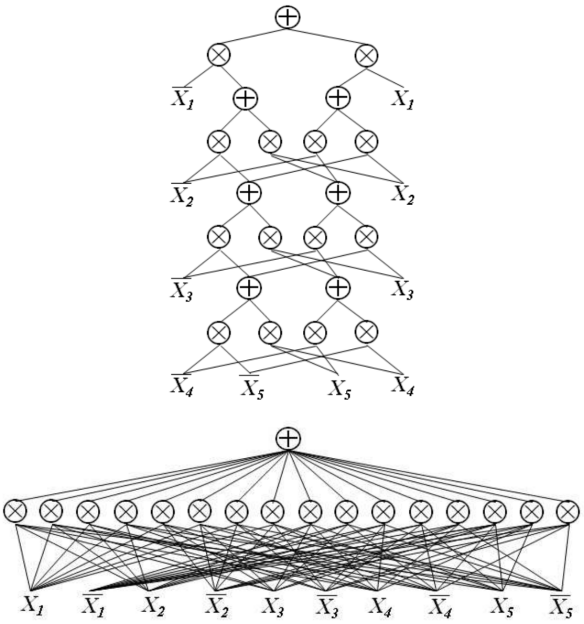


*Figure 3.* The figure demonstrates how SPNs can be used to represent complex probability distributions in a more compact and efficient way than Mixture models. (Poon & Domingos, 2011)

## 7. Conclusion

In conclusion, Sum-Product-Networks are powerful types of probabilistic graphical models that consist of a number of advantages over other models, including linear time inference as well as compactness when it comes to partition functions. Furthermore, there are large libraries such as TensorFlow and libspn-keras that can be utilized to create SPNs with ease. Overall, SPNs can be used for a variety of purposes, such as image classification, as demonstrated in this report. It is essential to consider various parameters during the training of the model for more optimized results and higher accuracy rates.

### 7.1. Future Work

Moving into the future, one thought is to develop new methods for SPNs to efficiently learn from large datasets as well as perform efficient inference when it comes to continuous variables.

## References

Poon, H. and Domingos, P. Sum-product networks: A new deep architecture. pp. 1–10, 2011.