# Dynamic Sum Product Networks for Tractable Inference on Sequence Data

**Mazen Melibari**[1] MMELIBAR@UWATERLOO.CA
**Pascal Poupart**[1] PPOUPART@UWATERLOO.CA
**Prashant Doshi**[2] PDOSHI@CS.UGA.EDU
**George Trimponias**[3] G.TRIMPONIAS@HUAWEI.COM
[1]*David R. Cheriton School of Computer Science, University of Waterloo Waterloo, Ontario, Canada*
[2]*Department of Computer Science, University of Georgia Athens, Georgia, USA*
[3]*Huawei Noah's Ark Lab, Hong Kong*

## Abstract

Sum-Product Networks (SPN) have recently emerged as a new class of tractable probabilistic models. Unlike Bayesian networks and Markov networks where inference may be exponential in the size of the network, inference in SPNs is in time linear in the size of the network. Since SPNs represent distributions over a fixed set of variables only, we propose dynamic sum product networks (DSPNs) as a generalization of SPNs for sequence data of varying length. A DSPN consists of a template network that is repeated as many times as needed to model data sequences of any length. We present a local search technique to learn the structure of the template network. In contrast to dynamic Bayesian networks for which inference is generally exponential in the number of variables per time slice, DSPNs inherit the linear inference complexity of SPNs. We demonstrate the advantages of DSPNs over DBNs and other models on several datasets of sequence data.

**Keywords:** Tractable probabilistic models; dynamic sum-product networks; sequence data.

## 1. Introduction

Probabilistic graphical models (Koller and Friedman, 2009) such as Bayesian networks (BNs) and Markov netwoks (MNs) provide a general framework to represent multivariate distributions while exploiting conditional independence. Over the years, many approaches have been proposed to learn the structure of those networks (Neapolitan, 2004). However, even if the resulting network is small, inference may be intractable (e.g., exponential in the size of the network) and practitioners must often resort to approximate inference techniques. Recent work has focused on the development of alternative probabilistic models such as arithmetic circuits (ACs) (Darwiche, 2003) and sum-product networks (SPNs) (Poon and Domingos, 2011) for which inference is guaranteed to be tractable (e.g., linear in the size of the network for SPNs and ACs). This means that the networks learned from data can be directly used for inference without any further approximation. So far, this work has focused on learning models for a fixed number of variables based on fixed-length data (Lowd and Domingos, 2012; Dennis and Ventura, 2012; Gens and Domingos, 2013; Peharz et al., 2013; Rooshenas and Lowd, 2014).

We present Dynamic Sum-Product Networks (DSPNs) as an extension to SPNs that model sequence data of varying length. Similar to Dynamic Bayesian networks (DBNs) (Dean and Kanazawa, 1989), DSPNs consist of a *template network* that repeats as many times as the length of a data sequence. We describe an invariance property for the template network that is sufficient to ensure that the resulting DSPN is valid (i.e., encodes a joint distribution) by being complete and decomposable.

Since existing structure learning algoritms for SPNs assume a fixed set of variables and fixed-length data, they cannot be used to learn the structure of a DSPN. We propose a general anytime search-and-score framework with a specific local search technique to learn the structure of the template network that defines a DSPN based on data sequences of varying length. We demonstrate the advantages of DSPNs over static SPNs, DBNs, hidden Markov models (HMMs) and recurrent neural networks (RNNs) with synthetic and real sequence data.

## 2. Background

**Definition 1 (Sum-Product Network (Poon and Domingos, 2011))** *A sum-product network (SPN) over $n$ binary variables $X_1, ..., X_n$ is a rooted directed acyclic graph whose leaves are the indicators $I_{x_1}, ..., I_{x_n}$ and $I_{\bar{x}_1}, ..., I_{\bar{x}_n}$, and whose internal nodes are sums and products. Each edge $(i, j)$ emanating from a sum node $i$ has a non-negative weight, $w_{ij}$. The value of a product node is the product of the values of its children. The value of a sum node is $\sum_{j \in Ch(i)} w_{ij} v_j$, where $Ch(i)$ is the set of children of $i$ and $v_j$ is the value of node $j$. The value of an SPN is the value of its root.*

The value of an SPN can be seen as the output of a network polynomial whose variables are the indicator variables and the coefficients are the weights (Darwiche, 2003). This polynomial represents a joint probability distribution over the variables if the SPN is *valid*. *Completeness* and *decomposability* (see below) are sufficient conditions for validity (Darwiche, 2003; Poon and Domingos, 2011) that impose some conditions on the *scope* of each node, which is the set of variables that appear in the sub-SPN rooted at that node.

**Definition 2 (Completeness)** *An SPN is complete iff all children of the same sum node have the same scope.*

**Definition 3 (Decomposability)** *An SPN is decomposable iff all children of the same product node have disjoint scopes.*

Several basic distributions can be encoded by simple SPNs. For instance, a univariate distribution can be encoded using an SPN whose root node is a sum that is linked to each indicator of a single variable $X$ (Fig. 1(a)). A factored distribution over a set of variables $X_1, ..., X_n$ is encoded by a root product node linked to univariate distributions for each variable $X_i$ (Fig. 1(b)). A naive Bayes model is encoded by a root sum node linked to a set of factored distributions (Fig. 1(c)) and a product of naive Bayes models is encoded by a root product node linked to a set of naive Bayes models (Fig. 1(d)).

Inference queries $\Pr(X = x | Y = y)$ can be answered by taking the ratio of the values obtained by two bottom up passes of an SPN. In the first pass, we initialize $I_x = 1$, $I_{\bar{x}} = 0$, $I_y = 1$, $I_{\bar{y}} = 0$ and set all remaining indicators to 1 in order to compute a value proportional to the desired query. In the second pass, we initialize $I_y = 1$, $I_{\bar{y}} = 0$ and set all remaining indicators to 1 in order to compute the normalization constant. The linear complexity of inference in SPNs is an appealing property given that inference for other models such as BNs is exponential in the size of the network in the worst case.

While SPNs are computational graphs based on which it is difficult to infer the relationships between the variables (e.g., conditional independence), Zhao et al. (2015) showed how to convert SPNs into equivalent bipartite Bayesian networks without any exponential blow up. In contrast, the

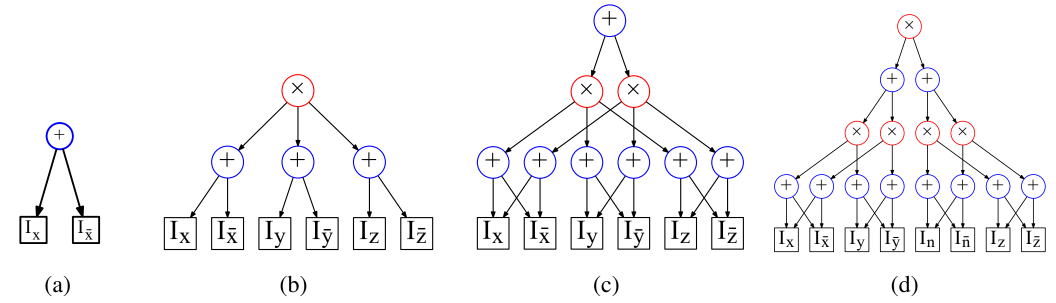

Figure 1: (a) Univariate distribution over a binary variable $x$. (b) Factored distribution over three binary variables $x$, $y$, and $z$. (c) naive Bayes model over three binary variables $x$, $y$, and $z$. (d) Product of naive Bayes models.

compilation of a Bayesian network into an equivalent SPN may yield an exponential blow up. SPNs are syntactically equivalent to arithmetic circuits (ACs) (Park and Darwiche, 2004) in the sense that they can be reduced to each other in linear time and space.

An extension to network polynomials for dynamic Bayesian networks (DBNs) was given in (Brandherm and Jameson, 2004). A procedure based on variable elimination is proposed to compile a DBN into a recursive network polynomial that can be represented by a special AC that we call dynamic AC. Since there is a risk that the compiled dynamic AC will be intractable, the authors use the Boyen-Koller (Boyen and Koller, 1998) method to approximate the output with a factored representation. Thus, compiling a DBN to a dynamic AC does not reduce the complexity of inference, but only makes it linear in the size of the compiled dynamic AC, which could be intractable. In contrast, we propose an approach to learn tractable models directly from sequence data.

## 3. Dynamic Sum-Product Networks

We propose dynamic SPNs (DSPNs) as a generalization of SPNs for modeling sequence data of varying length. While DSPNs are equivalent to dynamic ACs (i.e., reducible to each other without any blow up), we develop a structure learning algorithm that learns a tractable DSPN directly from sequence data (instead of learning a DBN from data and then compiling it into a potentially exponentially larger DSPN or dynamic AC). We also show sufficient conditions to ensure that estimated DSPNs are valid and therefore permit exact sequential inference in linear time.

Consider temporal sequence data that is generated by $n$ variables (or features) over $T$ time steps: $\left\langle \langle X_1, X_2, \ldots, X_n\rangle^1, \langle X_1, X_2, \ldots, X_n\rangle^2, \ldots, \langle X_1, X_2, \ldots, X_n\rangle^T \right\rangle$ where $X_i$, $i = 1 \ldots n$ is a random variable in one time slice and $T$ may vary with each sequence. Note that non-temporal sequence data such as sentences (sequence of words) can also be represented by sequences of repeated features. We will label the set of repeating variables as a *slice* and we will index slices by $t$ even if the sequence is not temporal, for uniformity.

A DSPN models sequences of varying length with a fixed number of parameters by using a template that is repeated at each slice. This is analogous to DBNs where the template corresponds to the network that connects two consecutive slices.

**Definition 4 (Template network)** *A template network for a slice of $n$ binary variables at time $t$, $\langle X_1, X_2, \ldots, X_n\rangle^t$, is a directed acyclic graph with $k$ roots and $k + 2n$ leaf nodes. The $2n$ leaf*

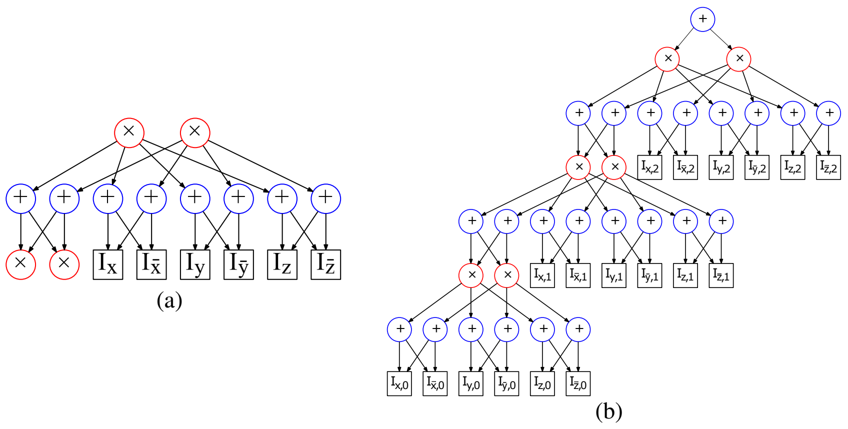

Figure 2: (a) An example of a generic template network. Notice the interface nodes in red. (b) A generic example of a complete DSPN unrolled over 3 time slices. Two template networks are stacked on the bottom network and capped by the top network.

*nodes are the indicator variables, $I_{x_1^t}, I_{x_2^t}, \ldots, I_{x_n^t}, I_{\bar{x}_1^t}, I_{\bar{x}_2^t}, \ldots, I_{\bar{x}_n^t}$. The remaining $k$ leaves and an equal number of roots are interface nodes to and from the template for the previous and next slices, respectively. The interface and interior nodes are either sum or product nodes. Each edge $(i, j)$ emanating from a sum node $i$ has a non-negative weight $w_{ij}$ as in a SPN. Furthermore, we define a bijective mapping $f$ between the input and output interface nodes.*

Fig. 2(a) shows a generic template network. In addition, we define two special networks.

**Definition 5 (Bottom network)** *A bottom network for the first slice of $n$ binary variables, $\langle X_1, X_2, \ldots, X_n \rangle^1$, is a directed acyclic graph with $k$ roots and $2n$ leaf nodes. The $2n$ leaf nodes are the indicator variables, $I_{x_1^1}, I_{x_2^1}, \ldots, I_{x_n^1}, I_{\bar{x}_1^1}, I_{\bar{x}_2^1}, ..., I_{\bar{x}_n^1}$. The $k$ roots are interface nodes to the template network for the next slice. The interface and interior nodes are either sum or product nodes. Each edge $(i, j)$ emanating from a sum node $i$ has a non-negative weight $w_{ij}$ as in a SPN.*

**Definition 6 (Top network)** *Define a top network as a rooted directed acyclic graph composed of sum and product nodes with $k$ leaves. The leaves of this network are interface nodes, which were introduced previously. Each edge $(i, j)$ emanating from a sum node $i$ has a non-negative weight $w_{ij}$ as in a SPN.*

Consider a data sequence of length $T$. A DSPN of $T$ slices is obtained by stacking $T - 1$ template networks of Def. 4 on top of a bottom network. This is capped by a top network. Two networks are stacked by merging the input interface nodes of the upper network with the output interface nodes of the lower network. Figure 2(b) shows an example with 3 slices of 2 variables each.

As we mentioned previously, completeness and decomposability are sufficient to ensure the validity of an SPN. While one could check that each sum node in the DSPN is complete and each product node is decomposable, we provide a simpler way to ensure that any DSPN is complete and decomposable. In particular, we describe an invariance property for the template network that can be verified directly in the template without unrolling the DSPN. This invariance property is sufficient to ensure that completeness and decomposability are satisfied in the DSPN for any number of slices.

**Definition 7 (Invariance)** *A template network over $\langle X_1, ..., X_n \rangle^t$ is invariant when the scope of each input interface node excludes variables $\{X_1^t, ..., X_n^t\}$ and for all pairs of input interface nodes, $i$ and $j$, the following properties hold:*

1. $scope(i) = scope(j) \vee scope(i) \cap scope(j) = \emptyset$
2. $scope(i) = scope(j) \iff scope(f(i)) = scope(f(j))$
3. $scope(i) \cap scope(j) = \emptyset \iff scope(f(i)) \cap scope(f(j)) = \emptyset$
4. *all interior and output sum nodes are complete*
5. *all interior and output product nodes are decomposable*

*Here $f$ is the bijective mapping that indicates which input nodes correspond to which output nodes in the interface.*

Intuitively, a template network is invariant if we can assign a scope to each input interface node such that each pair of input interface nodes has the same scope or disjoint scopes, and the same relation holds between the scopes of the corresponding output nodes. Scopes of pairs of corresponding interface nodes must be the same or disjoint because a product node is decomposable when its children have disjoint scopes and a sum node is complete when its children have identical scope. Hence, verifying the identity or disjoint relation of the scopes for every pair of input interface nodes helps us in verifying the completeness and decomposability of the remaining nodes in the template. Theorem 8 below shows that the invariance property of Def. 7 can be used to ensure that the corresponding DSPN is complete and decomposable.

**Theorem 8** *If (a) the bottom network is complete and decomposable, (b) the scopes of all pairs of output interface nodes of the bottom network are either identical or disjoint, (c) the scopes of the output interface nodes of the bottom network can be used to assign scopes to the input interface nodes of the template and top networks in such a way that the template network is invariant and the top network is complete and decomposable, then the corresponding DSPN is complete and decomposable.*

**Proof** We sketch a proof by induction (see the extended version for more details (Melibari et al., 2016)). For the base case, consider a single-slice DSPN (bottom network) capped with a top networks. The bottom network is complete and decomposable by assumption. Since the interface output nodes of the bottom network are merged with the input interface nodes of the top network, they are assigned the same scope, which ensures that the top network is also complete and decomposable. For the induction step, assume that a DSPN of $T$ slices is complete and decomposable. Consider a DSPN of $T + 1$ slices that shares the same bottom network and the same first $T - 1$ copies of the template network as the DSPN of $T$ slices. Hence the bottom network and the first $T - 1$ copies of the template network in the DSPN of $T + 1$ slices are complete and decomposable. Since the next copy of the template network is invariant when its input interface nodes are assigned the scopes with the same identity and disjoint relations as the scopes of the output interface nodes of the bottom network, it is also complete and decomposable. Similarly, the top network is complete and decomposable. ■

## 4. Structure Learning of DSPN

As a DSPN is an SPN, we could ignore the repeated structure and learn an SPN for the number of variables corresponding to the longest sequence. Shorter sequences could be treated as sequences with missing data for the unobserved slices. Unfortunately, this is intractable for very long sequences because the inability to model the repeated structure implies that the SPN will be very large and the learning computationally intensive. This approach may be feasible for datasets that contain only short sequences, nevertheless the amount of data needed may be prohibitively large because in the absence of a repeating structure the number of parameters is much higher. Furthermore, the SPN could be asked to perform inference on a sequence that is longer than any of the training sequences, and it is likely to perform poorly.

Alternately, it is tempting to apply existing algorithms to learn the repeated structure of the DSPN. Unfortunately, this is not possible. As existing algorithms assume a fixed set of variables, one could break data sequences into fixed-length segments corresponding to each slice. An SPN can be learned from this dataset of segments. However, it is not clear how to use the resulting SPN to construct a template network because a regular SPN has a single root while the template network has multiple roots and an equal number of input leaves that are not indicator variables. One would have to treat each segment as independent data instances and could not answer queries about the probability of some variables in one slice given the values of other variables in other slices.

We present an *anytime search-and-score* framework to learn the structure of the template SPN in a DSPN. It starts with an arbitrary structure and then generates several neighbouring structures. It ranks the neighbouring structures according to a scoring function and selects the best neighbour. These steps are repeated until a stopping criterion is met. This framework can be instantiated in multiple ways based on the choice for the initial structure, the neighbour-generation process, the scoring function and the stopping criterion. We proceed with the description of a specific instantiation below, although other instantiations are possible.

Without loss of generality, we propose to use product nodes as the interface nodes for both the input and output of the template network.[1] We also propose to use a bottom network that is identical to the template network after removing the nodes that do not have any indicator variable as descendent. This way we can design a single algorithm to learn the structure of the template network since the bottom network will be automatically determined from the learned template. We also propose to fix the top network to a root sum node directly linked to all the input product nodes. For the template network, we initialize the SPN rooted at each output product node to a factored model of univariate distributions. Figure 2(a) shows an example of this initial structure with two interface nodes and three variables. Each output product node has four children where each child is a sum node corresponding to a univariate distribution. Three of those children are univariate distributions linked to the indicators of the three variables, while the fourth sum node is a distribution over the interface input nodes. On merging the interface nodes for repeated instantiations of the template, we obtain a hierarchical mixture model. We begin with a single interface node and iteratively increase their number until the score stops improving. Alg. 1 summarizes the steps to compute the initial structure.

1. WLOG assume that the DSPN alternates between layers of sum and product nodes. Since a DSPN consists of a repeated structure, there is flexibility in choosing the interfaces of the template. We chose the interfaces to be at layers of product nodes, but the interfaces could be shifted by one level to layers of sum nodes or even traverse several layers to obtain a mixture of product and sum nodes. These boundaries are all equivalent subject to suitable adjustments to the bottom and top networks.

**Algorithm 1** Initial Structure

```
Input: $trainSet$, $validationSet$, $\langle X_1, ..., X_n \rangle$ (variables for a slice)
Output: $templNet$: Initial Template Network Structure
  $g \leftarrow factoredDistribution(\langle X_1, ..., X_n \rangle)$
  $newTempl \leftarrow train(g, trainSet)$
  repeat $templNet \leftarrow newTempl$; $newTempl \leftarrow train(templNet \cup \{g\}, trainSet)$
  until $likelihood(newTempl, validationSet) < likelihood(templNet, validationSet)$
```

A simple scoring function is to use the likelihood of the data since exact inference in DSPNs can be done quickly. If the goal is to produce a generative model of the data, then the likelihood of the data is a natural criterion. If the goal is to produce a discriminative model for classification, then the conditional likelihood of some class variables given the remaining variables is a suitable criterion. For a given structure, parameters can be estimated using various parameter learning algorithms including gradient ascent (Poon and Domingos, 2011) and expectation maximization (Poon and Domingos, 2011; Peharz, 2015).

Our neighbour generation process (Alg. 2) begins by sampling a product node uniformly and replacing the sub-SPN rooted at that product node by a new sub-SPN. Note that to satisfy the decomposability property, a product node must partition its scope into disjoint scopes for each of its children. Also note that different partitions of the scope can be seen as different conditional independencies between the variables (Gens and Domingos, 2013). Hence, the search space of a product node generally corresponds to the set of all partitions of its scope. We use the 'restricted growth string (RGS)' encoding of partitions to define a lexicographical order of the set of all possible partitions (Knuth, 2006). We can select the next partition according to the lexicographic ordering or by sampling from a distribution over all possible partitions. The distribution can be uniform in the absence of prior knowledge or an informed one otherwise.

Since the search space is exponential in the number of variables in the scope of the product node, we greedily split the scope into mutually independent subsets according to pairwise independence tests applied recursively similar to (Gens and Domingos, 2013). In case no independent subsets are found, we sample a partition at random when the number of variables is greater than some threshold and select the next partition according to the lexicographic ordering otherwise. Alg. 3 describes the process of finding the next partition based on which we construct a product of naive Bayes models (Fig. 1(d)) where each naive Bayes model has two children that encode factored distributions. This may increase or decrease the size of the template network depending on whether the new product of naive Bayes models replaces a bigger or smaller sub-SPN at the sampled product node.

Since constructing the new template, learning its parameters, and computing its score can be done in a time that is linear in the size of the template network and the dataset, each iteration of the anytime search-and-score algorithm scales linearly with the size of the template network and the amount of data.

**Theorem 9** *The network templates produced by Alg. 1 and 2 are invariant.*

**Proof** Let the scope of all input interface nodes be identical. The initial structure of the template network is a collection of factored distributions over all the variables. Hence the output interface nodes all have the same scope (which includes all the variables). Hence, Alg. 1 produces an initial template network that is invariant. Alg. 2 replaces the sub-SPN of a product node by a new

**Algorithm 2** Generate neighbour (improved template network)

```
Input: $trainSet, validationSet, templNet$
Output: $templNet$
  repeat
    $n \leftarrow$ sample product node uniformly from $templNet$
    $newPartition \leftarrow \text{GetPartition(n)}$
    $n' \leftarrow$ construct product of naiveBayes models based on $newPartition$
    $newTempl \leftarrow$ replace $n$ by $n'$ in $templNet$
  until $likelihood(newTempl, validationSet) < likelihood(templNet, validationSet)$
```

**Algorithm 3** GetPartition

```
Input: product node $n$
Output: $nextPartition$
  if $|\text{scope(n)}| > \text{threshold}$ then
    $\{s_1, ..., s_k\} \leftarrow$ partition $scope(n)$ into indep. subsets
    if $k > 1$ then return $\cup_{i=1}^{k} GetPartition(s_i)$
    else return random partition of $scope(n)$
  else return next lexicographic partition of $scope(n)$ according to the RGS encoding
```

sub-SPN, which does not change the scope of the product node. This follows from the fact that the new partition used to construct the new sub-SPN has the same variables as the original partition. Since the scope of the product node under which we change the sub-SPN does not change, all nodes above that product node, including the output interface nodes, preserve their scope. Hence Alg. 2 produces neighbour template networks that are invariant. ■

## 5. Experiments

We evaluate the performance of our anytime search-and-score method for DSPNs on several synthetic and real-world *sequence* datasets. In addition, we measure how well the DSPNs model the data by comparing the negative log-likelihoods with those of static SPNs learned using LearnSPN (Gens and Domingos, 2013), and with other dynamic models such as Hidden Markov Models (HMM), DBNs and recurrent neural networks (RNNs). The threshold in Alg. 3 was set to 6 in all experiments.

The synthetic datasets include three dynamic processes with different structures: sequences of observations sampled from $(i)$ an HMM with one hidden variable, $(ii)$ the well-known Water DBN (Jensen et al., 1989) and $(iii)$ the Bayesian automated taxi (BAT) DBN (Forbes et al., 1995). We also evaluate DSPNs with 5 real-world sequence datasets from the UCI repository (Lichman, 2013). They include applications such as online handwriting recognition (Alimoglu and Alpaydin, 1996) and speech recognition (Hammami and Sellam, 2009; Kudo et al., 1999).

We first compare DSPNs to the true model on the synthetic datasets. As LearnSPN cannot be used with data of variable length, we include it in the synthetic datasets experiment only, where we sample sequences of fixed length. Table 1 shows the negative log-likelihoods based on 10-fold cross

| Dataset<br>(#i, length, #oVars) | HMM-Samples<br>(100, 100, 1) | Water<br>(100, 100, 4) | BAT<br>(100, 100, 10) |
|---|---|---|---|
| True model | 62.2 ± 0.8 | 249.6 ± 1.0 | 628.2 ± 2.0 |
| LearnSPN | 65.4 ± 0.7 | 270.4 ± 0.9 | 684.4 ± 1.3 |
| DSPN | **62.5** ± 0.7 | **252.4** ± 0.9 | **641.6** ± 1.1 |

Table 1: Mean negative log-likelihood and standard error based on 10-fold cross validation for the synthetic datasets. (#i,length,#oVars) indicates the number of data instances, length of each sequence and number of observed variables. Lower likelihoods are better.

| Dataset<br>(#i,length,#oVars) | ozLevel<br>(2533,24,2) | PenDigits<br>(10992,16,7) | ArabicDigits<br>(8800,40,13) | JapanVowels<br>(640,16,12) | ViconPhysic<br>(200,3026,27) |
|---|---|---|---|---|---|
| HMM | 56.7 ± 1.1 | 74.2 ± 0.1 | 327.5 ± 0.4 | 94.3 ± 0.3 | 40862 ± 369 |
| HMM-SPN | 49.8 ± 0.9 | 67.7 ± 0.6 | 305.8 ± 1.8 | 89.8 ± 1.2 | 38410 ± 440 |
| RNN | **16.2** ± 0.7 | 68.7 ± 1.3 | 303.6 ± 6.4 | 78.8 ± 2.3 | 57217 ± 873 |
| Search-Score DBN | 40.2 ± 4.7 | 67.3 ± 2.3 | 263.7 ± 4.6 | 75.6 ± 2.5 | - |
| Reveal DBN | 52.4 ± 2.5 | 74.4 ± 0.2 | 260.2 ± 1.0 | 71.3 ± 1.2 | - |
| DSPN | 33.0 ± 1.0 | **63.5** ± 0.3 | **257.9** ± 0.5 | **68.8** ± 0.3 | **36385** ± 682 |

Table 2: Mean negative log-likelihood and standard error based on 10-fold cross validation for the real world datasets. (#i,length,#oVars) indicates the number of data instances, average length of the sequences and number of observed variables.

validation for the synthetic datasets. In all three synthetic datasets, DSPN learned generative models that exhibited likelihoods that are close to that of the true models. It also outperforms LearnSPN in all three cases.

Next, we compare DSPNs to classic HMMs with parameters learned by Baum-Welch (Baum et al., 1970), HMM-SPNs where each observation distribution is an SPN (Peharz et al., 2014), fully observable DBNs whose structure is learned by the Reveal algorithm (Liang et al., 1998) from the BayesNet Toolbox (Murphy, 2001), partially observable DBNs, whose structure and hidden variables are learned by search and score (Friedman et al., 1998), and RNNs with one input node, one hidden layer consisting of long short term memory (LSTM) units (Hochreiter and Schmidhuber, 1997) and one output sigmoid unit with a cross-entropy loss function. We select LSTM units due to their popularity and success in sequence learning (Sutskever et al., 2014). The input node corresponds to the value of the current observation and the output node to the predicted value of the next observation in the sequence. We train the network by backpropagation through time (bptt) truncated to 20 time steps (Williams and Peng, 1990) with a learning rate of 0.01. Our implementation is based on the Theano library (Theano Development Team, 2016) in Python.

Table 2 shows the results for the real datasets. DSPNs outperform the other approaches except for one dataset where the RNN achieved better results. DSPNs are more expressive than classic HMMs and HMM-SPNs since our search and score algorithm has the flexibility of learning a suitable structure with multiple interface nodes for the transition dynamics where as the structure of the transition dynamics is fixed with a single hidden variable in classic HMMs and HMM-SPNs. DSPNs are also more expressive than the fully observable DBNs found by Reveal since the sum nodes in the template networks implicitly denote hidden variables. DSPNs are as expressive as the partially observable DBNs found by search and score, but better results are achieved by DSPNs because their linear inference complexity allows us to explore a larger space of structures more quickly. DSPNs are less expressive than RNNs since DSPNs are restricted to sum and product nodes while RNNs use sum, product, max and sigmoid operators. Nevertheless, RNNs are notoriously difficult to train due to the non-convexity of their loss function and vanishing/exploding gradient issues that arise in backpropagation through time. This explains why RNNs did not outperform DSPNs on 4 of the 5 datasets.

Table 3 shows the time to learn and do inference with the DBN, RNN and DSPN models (the HMM models are omitted since they do not learn any structure for the transition dynamics and therefore are not as expressive). All models were trained till convergence or up to two days. We report the total time for learning with Reveal and the time per iteration for learning with the other algorithms since they are anytime algorithms. Learning DSPNs is generally faster than training RNNs and search-and-score DBNs. The time to do inference for all the sequences in each dataset when one variable is observed and the other variables are hidden is reported in the right hand side of the table. DSPNs and RNNs are fast since they allow exact inference in linear time with respect to the size of their network, while the DBNs obtained by Reveal and search-and-score are slow because inference may be exponential in the number of hidden variables if they all become correlated.

## 6. Conclusion

Existing methods for learning SPNs become inadequate when the task involves modeling sequence data such as time series data points. The specific challenge is that sequence data could be composed of instances of different lengths. Motivated by dynamic Bayesian networks, we presented a new model called dynamic SPN, which utilized a template network as a building block. We also defined a notion of invariance and showed that invariant template networks can be composed safely to ensure that the resulting DSPN is valid. We provided an anytime algorithm based on the framework of search-and-score for learning the structure of the template network from data. As our experiments demonstrated, a DSPN fits sequential data better than static SPNs (produced by LearnSPN). We also showed that the DSPNs found by our search-and-score algorithm achieve higher likelihood than competing HMMs, DBNs and RNNs on several temporal datasets. While approximate inference is typically used in DBNs to avoid an exponential blow up, inference can be done exactly in linear time with DSPNs.

| Dataset | Learning Time (Seconds) | | | | Inference Time (Seconds) | | | |
|---|---|---|---|---|---|---|---|---|
| | Reveal | Per Iteration | | | Reveal | RNN | SS DBN | DSPN |
| | | RNN | SS DBN | DSPN | | | | |
| ozLevel | 952 | 56 | 108 | 54 | 6.3 | 0.1 | 15.6 | 0.1 |
| PenDigits | 3,977 | 558 | 1,463 | 475 | 15.0 | 0.2 | 30.7 | 0.1 |
| ArabicDigits | 16,549 | 2572 | 14,911 | 2,909 | 53.6 | 2.5 | 465.8 | 2.9 |
| JapaneseVowls | 516 | 55 | 363 | 51 | 15.2 | 0.2 | 69.2 | 0.5 |
| ViconPhysical | - | 4705 | - | 6734 | - | 2274 | - | 1825 |

Table 3: Comparisons of the learning and inference times of the networks learned by Reveal, RNN, Search-Score DBN (SS DBN) and DSPN.

## Acknowledgments

This research was funded by a grant from Huawei Noah's Ark Lab in Hong Kong.

## References

F. Alimoglu and E. Alpaydin. Methods of combining multiple classifiers based on different representations for pen-based handwritten digit recognition. In *Turkish Artificial Intelligence and Artificial Neural Networks Symposium*, 1996.

L. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of mathematical statistics*, pages 164–171, 1970.

X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *UAI*, pages 33–42, 1998.

B. Brandherm and A. Jameson. An extension of the differential approach for Bayesian network inference to dynamic Bayesian networks. *International Journal of Intelligent Systems*, 19(8): 727–748, 2004.

A. Darwiche. A differential approach to inference in Bayesian networks. *JACM*, 50(3):280–305, 2003.

T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational intelligence*, 5(2):142–150, 1989.

A. Dennis and D. Ventura. Learning the architecture of sum-product networks using clustering on variables. In *NIPS*, 2012.

J. Forbes, T. Huang, K. Kanazawa, and S. Russell. The batmobile: Towards a Bayesian automated taxi. In *IJCAI*, pages 1878–1885, 1995.

N. Friedman, K. Murphy, and S. Russell. Learning the structure of dynamic probabilistic networks. In *UAI*, pages 139–147, 1998.

R. Gens and P. Domingos. Learning the structure of sum-product networks. In *ICML*, pages 873–880, 2013.

N. Hammami and M. Sellam. Tree distribution classifier for automatic spoken arabic digit recognition. In *Internet Technology and Secured Transactions*, pages 1–4, 2009.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comp*, 9(8):1735–1780, 1997.

F. Jensen, U. Kjærulff, K. Olesen, and J. Pedersen. Et forprojekt til et ekspertsystem for drift af spildevandsrensning (an expert system for control of waste water treatment). Technical report, 1989.

D. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. 2006.

D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. 2009.

M. Kudo, J. Toyama, and M. Shimbo. Multidimensional curve classification using passing-through regions. *Pattern Recognition Letters*, 20(11):1103–1111, 1999.

S. Liang, S. Fuhrman, and R. Somogyi. Reveal, a general reverse engineering algorithm for inference of genetic network architectures. In *Pacific symposium on biocomputing*, volume 3, pages 18–29, 1998.

M. Lichman. UCI machine learning repository, 2013. URL `archive.ics.uci.edu/ml`.

D. Lowd and P. Domingos. Learning arithmetic circuits. *arXiv:1206.3271*, 2012.

M. Melibari, P. Poupart, P. Doshi, and G. Trimponias. Dynamic sum-product networks for tractable inference on sequence data (extended version). *arXiv:1511.04412*, 2016.

K. Murphy. The Bayes net toolbox for matlab. *Computing Science and Statistics*, 33, 2001.

R. Neapolitan. *Learning Bayesian networks*, volume 38. Prentice Hall, 2004.

J. Park and A. Darwiche. A differential semantics for jointree algorithms. *Artificial Intelligence*, 156(2):197–216, 2004.

R. Peharz. *Foundations of Sum-Product Networks for Probabilistic Modeling*. PhD thesis, Medical University of Graz, 2015.

R. Peharz, B. Geiger, and F. Pernkopf. Greedy part-wise learning of sum-product networks. In *ECML PKDD*, pages 612–627, 2013.

R. Peharz, G. Kapeller, P. Mowlaee, and F. Pernkopf. Modeling speech with sum-product networks: Application to bandwidth extension. In *ICASSP*, pages 3699–3703, 2014.

H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In *UAI*, pages 2551–2558, 2011.

A. Rooshenas and D. Lowd. Learning sum-product networks with direct and indirect variable interactions. In *ICML*, pages 710–718, 2014.

I. Sutskever, O. Vinyals, and Q. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, 2016.

R. Williams and J. Peng. An efficient gradient-based algorithm for online training of recurrent network trajectories. *Neural Computation*, 2(4):490–501, 1990.

H. Zhao, M. Melibari, and P. Poupart. On the relationship between sum-product networks and Bayesian networks. In *ICML*, 2015.