Language Modeling with Sum-Product Networks

Wei-Chen Cheng [1], *Stanley Kok* [1], *Hoai Vu Pham* [1], *Hai Leong Chieu* [2], *Kian Ming A. Chai* [2]

[1]Information Sys. Tech. & Design Pillar, Singapore University of Technology & Design, Singapore
[2]DSO National Laboratories, Singapore

{weichen_cheng,stanleykok,hoaivu_pham}@sutd.edu.sg
{chaileon,ckianmin}@dso.org.sg

10.21437/Interspeech.2014-476

Abstract

Sum product networks (SPNs) are a new class of deep probabilistic models. They can contain multiple hidden layers while keeping their inference and training times tractable. An SPN consists of interleaving layers of sum nodes and product nodes. A sum node can be interpreted as a hidden variable, and a product node can be viewed as a feature capturing rich interactions among an SPN's inputs. We show that the ability of SPN to use hidden layers to model complex dependencies among words, and its tractable inference and learning times, make it a suitable framework for a language model. Even though SPNs have been applied to a variety of vision problems [1, 2], we are the first to use it for language modeling. Our empirical comparisons with six previous language models indicate that our SPN has superior performance.

Index Terms: language models, sum-product networks, deep learning, probabilistic graphical models

1. Introdution

Language models play a critical role in automatic speech recognition by modeling prior knowledge about a natural language and bringing it to bear on the likelihood of speech transcriptions. Typically they model the probability distribution over the sequence of words w_1^m in a transcription as $P(w_1^m) \approx \prod_{k=1}^{m} P(w_k|w_{k-n+1}^{k-1})$ where w_i^j is a sequence of words $w_i, w_{i+1}, \ldots, w_{j-1}, w_j$. From the right-hand side of the above equation, we observe that the crux of a language model lies in the conditional probability of a word w_k given its previous $n-1$ words, i.e., $P(w_k|w_{k-n+1}^{k-1})$. A basic language model is the n-gram model, which simply counts the fraction of times w_k appears after a fixed (n-1)-length sequence w_{k-n+1}^{k-1} among all occurences of the sequence in a corpus. However, n-gram models for moderately large n's often do not reliably estimate the conditional probability because many plausible word sequences have too few (often zero) occurrences in a corpus. To ameliorate this *data sparsity* problem, several approaches combine n-gram models by computing their weighted sum over a range of n's (to *smooth* over unseen sequences). One prominent method among such approaches is the Kneser-Neys KN5 algorithm [3].

More sophisticated language models include the log-bilinear model [4], feedforward neural networks [5], and recurrent neural networks (RNN) [6].

The log-bilinear model [4] is a probabilistic graphical model [7] that encodes the dependencies between all pairs of words in a vocabulary. It performs moderately well but cannot exploit the rich information that exists among three or more words. Although its creators proposed other probabilistic graphical models that use *hidden variables* to represent higher-order interactions among words, they found that those models performed similarly to their log-bilinear counterpart but took longer to train.

Bengio et al. [5] used a feedforward neural network as a language model. It uses the common 1-of-N representation of a word (i.e., an N-dimensional vector with a single 1 at the index corresponding to the word and 0's everywhere else), but compresses it into a smaller continuous-valued *feature vector*. (Our proposed approach uses feature vectors too as will be described in Section 3.) Intuitively, a feature vector provides a distributed continuous representation of an input word, with the vector's continuous values varying gradually among similar words and differing greatly among dissimilar ones. Such vectors are then used to learn a probability distribution over the words they represent. The continuity in the vectors automatically smoothens the distribution and alleviates the data sparsity problem. To model high-order interactions among words, a neural network adds a *hidden layer* that uses the feature vectors as inputs. As more hidden layers are added (one on top of another), a neural network can model more complex interactions, but at the expense of longer training times. Thus, Bengio et al.'s neural network uses only one hidden layer to capture the dependencies among words, without incurring too large a penalty in training time. To improve upon Bengio et al.'s neural network, Emani and Jelink [8] augmented words with their syntactic information.

Recurrent neural networks (RNNs) [6] have also been proposed as language models. An RNN is similar to a feedforward neural network in having an input layer of words that is connected to a hidden layer, which in turn is connected to an output layer representing a probability distribution over words. It differs by linking the hidden layer back to itself with recurrent connections, which propagate information across a sequence of words in an RNN. Conceptually, when an RNN is "unrolled", it is equivalent to a feedforward neural network with an infinite number of connected hidden layers stacked on top of one another. Because of this depth of hidden layers, it can potentially learn complex dependencies among words, but also incur a large penalty in training time. To improve upon the RNN language models, Mikolov et al. [9] augmented them with contextual information via latent Dirichlet allocation [10] to obtain state-of-the-art results. Recently, Sundermeyer et al. [11] provided empirical evidence that RNNs are better than their feedforward counterparts as language models. However, they used a feedforward neural network with only one hidden layer. Conceivably, feedforward neural networks with more hidden layers

Copyright © 2014 ISCA

could be competitive against RNNs (as we will show with our proposed approach in Section 4).

In this paper, we show that a new probabilistic graphical model called sum-product networks (SPNs) [12, 2] can function as a language model. Our proposed SPN is able to encapsulate multiple hidden layers while maintaining tractable inference and training. Empirically, it achieves better predictive accuracy than the aforementioned methods.

SPNs have been successfully applied to vision problems [1, 2], but to date, no one has brought it to bear on the problem of language modeling. To our knowledge, we are the first to do so.

We begin by describing SPNs in the next section. Then we describe our SPN architecture in detail (Section 3) and report our experiments (Section 4). Finally, we conclude with future work (Section 5).

2. Sum-Product Networks

We briefly review sum-product networks (SPNs). More details can be found in [12, 2, 13]. An SPN is a rooted directed acyclic graph that efficiently computes the marginals and modes of a probabilistic graphical model (PGM) [7] by compactly representing the PGM's partition function. A PGM encodes a probability distribution over a set of variables X as

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z} \prod_c \phi_c(\mathbf{x_c})$$

where ϕ_c is a function over a subset of variables $\mathbf{X_c}$ and $Z = \sum_x \prod_c \phi_c(\mathbf{x_c})$ is the partition function. We can regard $\Phi(\mathbf{x}) = \prod_c \phi_c(\mathbf{x_c})$ as an unnormalized probability that we divide by Z to obtain a valid probability. Computing marginals in a PGM is generally intractable because it involves a sum in Z over an exponential number of terms (i.e., all combinations of values of the variables in $\mathbf{X}$). Since Z involves only sums and products, it can be computed efficiently if we can reorganize it compactly in terms of a polynomial number of sums and products using the distributive law. SPNs overcome the intractability of Z by learning such a compact structure.

Definition 1. *(Gens & Domingos [13]) An SPN is recursively defined as follows.*

1. *A tractable univariate distribution is an SPN (a tractable univariate distribution is one whose partition function and mode can be computed in $O(1)$ time).*
2. *A product of SPNs with disjoint scopes is an SPN (an SPN's scope consists of the variables that appear in it).*
3. *A weighted sum of SPNs with the same scope is an SPN, provided all weights are positive.*
4. *Nothing else is an SPN.*

Figure 1 shows an example of an SPN over two binary variables X_1 and X_2. An SPN has internal nodes that are alternating layers of sums and products, and leaves that are indicators $\bar{x}_1, \ldots, \bar{x}_n$ and $x_1, \ldots, x_n$. (Indicators $\bar{x}_i$ and x_i take on the values of 1 when variable X_i is respectively false and true, and the value of 0 when X_i is respectively true and false.) Each edge linking a sum node i to a child product node j is associated with a non-negative weight w_{ij} . The value of a product node is given by the product of its children's values. The value of a sum node is the sum of its children's values weighted by the values of the children's edges, i.e., $\sum_{j \in C(i)} w_{ij} v_j$ where $C(i)$ is the set of i's children and v_j is the value of child j. The value of an SPN is given by its root value and is denoted as $S(\bar{x}_1, \ldots, \bar{x}_n, x_1, \ldots, x_n)$.

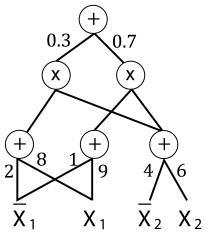


Figure 1: An SPN over two variables.

Theorem. *(Gens & Domingos [13]) An SPN can compute each of the following quantities in time linear in its number of edges: the partition function, the probability of evidence, and the maximum a posteriori (MAP) state.*

The partition function Z is the value at the root node, which can be tractably computed by setting all indicators to 1 and making a single upward pass. In Figure 1, the partition function is $S(\bar{x}_1 = 1, x_1 = 1, \bar{x}_2 = 1, x_2 = 1) = 0.3(2\bar{x}_1+8x_1)(4\bar{x}_2+6x_2)+0.7(\bar{x}_1+9x_1)(4\bar{x}_2+6x_2) = 100$. It is easy to see that if the weights of each sum node are normalized to add to 1, then $Z = 1$ and $P(\mathbf{X})$ is given by the value of the root node. The marginal of a variable can also be tractably computed via a single upward-downward pass through the SPN as described by [12]. A multi-valued categorical variable in an SPN is modeled by replacing the Boolean indicators x and $\bar{x}$ with an indicator for each of the variable's possible values (which is what we do for our SPN described in the next section). Continuous variables are dealt with by replacing sum nodes with integral nodes and assuming a parametric distribution (e.g., Gaussian) over the variables.

Poon and Domingos [12] have shown that each sum node of an SPN can be viewed as a *hidden variable* whose value is defined in terms of its children. Alternatively, a sum node can be interpreted as a mixture model with its children as its mixture components, and an entire SPN can be seen as a mixture model with exponentially many mixture components formed through the layers, with higher-level components reusing lower-level ones.

The SPN described thus far is a *generative* model, i.e., one that encodes the probability distribution over *all* variables, $P(\mathbf{X})$. However, it is generally noted that better predictive performance is obtained with a *discriminative* model, i.e., one which represents the conditional probability distribution $P(\mathbf{Y}|\mathbf{X})$ only over variables of interest $\mathbf{Y}$ (called *query* variables) given the values of input variables $\mathbf{X}$ (known as *evidence*). Intuitively, discriminative models concentrate on encoding interactions among query variables, without modeling the (unimportant) distribution among evidence, whose values are always provided (and thus never inferred). The constraints in Definition 1 only apply to query variables, thus allowing flexible features to be defined over evidence.

Gens & Domingos [2] propose a discriminative SPN that divides a set of variables into disjoint subsets $\mathbf{Y}$ (query), $\mathbf{X}$ (evidence) and $\mathbf{H}$ (hidden variables). It models the conditional

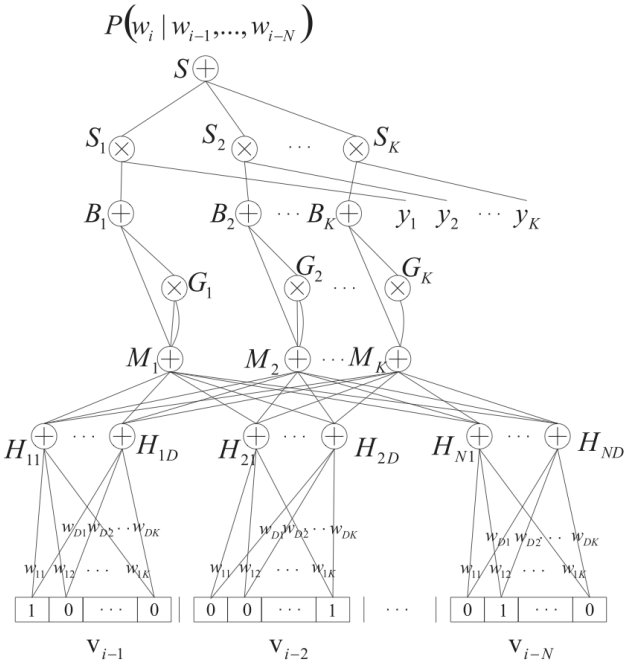


Figure 2: SPN for language modeling.

probability as

$$P(\mathbf{Y}=\mathbf{y}|\mathbf{X}=\mathbf{x}) = \frac{\Phi(\mathbf{Y}=\mathbf{y}|\mathbf{X}=\mathbf{x})}{\sum_{\mathbf{y}'}\Phi(\mathbf{Y}=\mathbf{y}'|\mathbf{X}=\mathbf{x})} = \frac{\sum_{\mathbf{h}}\Phi(\mathbf{Y}=\mathbf{y},\mathbf{H}=\mathbf{h}|\mathbf{X}=\mathbf{x})}{\sum_{\mathbf{y}',\mathbf{h}}\Phi(\mathbf{Y}=\mathbf{y}',\mathbf{H}=\mathbf{h}|\mathbf{X}=\mathbf{x})}$$

where $\Phi(\mathbf{Y}=\mathbf{y}|\mathbf{X}=\mathbf{x})$ is an unnormalized probability. Thus the partial derivative of the conditional log-likelihood with respect to a weight w in an SPN is given by:

$$\frac{\partial}{\partial w}\log P(\mathbf{y}|\mathbf{x}) = \frac{\partial}{\partial w}\log\sum_{\mathbf{h}}\Phi(\mathbf{y},\mathbf{h}|\mathbf{x}) - \frac{\partial}{\partial w}\log\sum_{\mathbf{y}',\mathbf{h}}\Phi(\mathbf{y}',\mathbf{h}|\mathbf{x}) \quad (1)$$

To train an SPN, we first specify its architecture, i.e., its sum and product nodes, and the connections between them. Then we learn the weights of the sum nodes via gradient descent to maximize the conditional log-likelihood of a training set of $(\mathbf{x},\mathbf{y})$ examples. The gradient of each weight (Equation 1) is computed via backpropagation. The first summation on the right-hand side of Equation 1 can be computed tractably in a single upward pass through the SPN by setting all hidden variables to 1, and the second summation can be computed similarly by setting both hidden and query variables to 1. The partial derivatives are passed from parent to child according to the chain rule as described by [14]. Each weight is changed by multiplying a learning rate parameter η to Equation 1, i.e., $\Delta w = \eta\frac{\partial}{\partial w}\log P(\mathbf{y}|\mathbf{x})$. To speed up training, we could estimate the gradient by computing it with a subset (mini-batch) of examples from the training set, rather than using all examples.

3. SPN Architecture

Figure 2 shows the architecture of our discriminative SPN for language modeling[1]. To predict a word (a query variable), we use its previous N words as evidence in our SPN. Each previous word is represented by a K-dimensional vector where K is the number of words in a vocabulary. Each vector has exactly one 1 at the index corresponding to the word it represents, and 0's everywhere else. When we predict the i^{th} word, we have a vector $\mathbf{v}_{i-j}$ $(1 \leq j \leq N)$ at the bottommost layer for each of the previous N words.

Above the bottommost layer, we have a (hidden) layer of sum nodes. There are D sum nodes $H_{j1} \ldots H_{jD}$ for each vector $\mathbf{v}_{i-j}$. Each sum node H_{jl} has an edge connecting it to every entry in $\mathbf{v}_{i-j}$. Let the m^{th} entry in $\mathbf{v}_{i-j}$ be denoted by $\mathbf{v}_{i-j}^m$, and the weight of the edge from H_{jl} to $\mathbf{v}_{i-j}^m$ be denoted by w_{lm}. We constrain each weight w_{lm} to be the same for each pair of H_{jl} and $\mathbf{v}_{i-j}^m$ $(1 \leq j \leq N)$. This layer of sum nodes can be interpreted as compressing each K-dimensional vectors $\mathbf{v}_{i-j}$ into a smaller continuous-valued D-dimensional feature vector (thus gaining the same advantages of [5] as described in Section 1). Because the weights w_{lm}'s are constrained to be the same between each pair of K-dimensional input vector and D-dimensional feature vector, we ensure that the weights are position independent, i.e., the same word will be compressed into the same feature vector regardless of its position. This also makes it easier to train the SPN by reducing the number of weights to be learned.

Above the H_{jl} layer, we have another layer of sum nodes. In this layer, each node M_k $(1 \leq k \leq K)$ is connected to every H_{jl} node. Moving up, we have a layer of product nodes. Each G_k product node is connected via two edges to an M_k node. Each G_k node transforms the output from its child M_k node by squaring it. This helps to capture more complicated dependency among the input words.

Moving up, we have another layer of sum nodes. Each B_k node in this layer is connected to an M_k node and a G_k node in the lower layers. Above this, there is a layer of S_k nodes, each of which is connected to a B_k node and an indicator variable y_k representing a value in our categorical query variable (i.e., the i^{th} word which we are predicting). $y_k = 1$ if the query variable is the k^{th} word, and $y_k = 0$ otherwise. Intuitively, the indicator variables select which part of the SPN below an S_k node gets "activated". Finally, we have an S node which connects to all S_k nodes. When we normalize the weights between S and the S_k nodes to sum to 1, S's output is the conditional probability of the i^{th} word given its previous N words.

4. Experiments

4.1. Dataset

We performed our experiments on the commonly used Penn Treebank corpus [15], and adhered to the experimental setup used in previous work [6, 9]. We used sections 0-20, sections 21-22, and sections 23-24 respectively as training, validation and test sets. These sections contain segments of news reports from the Wall Street Journal. We treated punctuation as words, and used the 10,000 most frequent words in the corpus to create a vocabulary. All other words are regarded as unknown and mapped to the token `<unk>`. The percentages of out-of-vocabulary (`<unk>`) tokens in them are about 5.91%, 6.96% and 6.63% respectively. Thus only a small fraction of the dataset consists of unknown words.

4.2. Methodology

Using the training set, we learned the weights of all sum nodes in our SPN described in Section 3. To evaluate

[1] https://github.com/stakok/lmspn/blob/master/faq.md contains more details about the architecture.

its performance on the test set, we used the standard (per-word) perplexity measure. The perplexity (PPL) on a sequence of words $w_1, w_2, \ldots, w_M$ is given by

$$PPL = \sqrt[M]{\prod_{i=1}^{M} \frac{1}{P(w_i|w_1, ..., w_{i-1})}}.$$

We estimated the probability $P(w_i|w_1, ..., w_{i-1})$ in PPL as $P(w_i|w_{i-1}, ..., w_{i-N})$ that is given by our SPN.

We used a learning rate of $\eta = 0.1$, a mini-batch size of 100, randomly initialized the weights to a value between 0 and 1, and imposed an L2 penalty of 10^{-5} on all weights. With reference to Figure 2, We used $K = 10000$, feature vectors with $D = 100$ dimensions, and $N = 3$ and $N = 4$ previous words. We denote an SPN that uses N previous words as SPN-N. We stopped training our SPN when its performance on the validation set stops improving at two consecutive evaluation points, or when it has run for 40 hours, whichever occurred first. (It turned out that both SPN-3 and SPN-4 ran for the maximum of 40 hours.) We parallelized our SPN code[2] to run on a GPU, and ran our experiments on a machine with a 2.4 GHz CPU and an NVIDIA Tesla C2075 GPU (448 CUDA cores, 5GB of device memory).

We compared our SPNs to an interpolated 5-gram model with modified Kneser-Ney smoothing and no count cutoffs (KN5) [3], the log-bilinear model [4], feedforward neural networks [5], syntactical neural networks [8], recurrent neural networks (RNN) [6], and LDA-augmented RNN [9], all of which are described in Section 1.

4.3. Results

Table 1 shows the results of our experiments. The scores of comparison systems are obtained from [9]. The "Individual PPL" column shows the perplexity score of the respective systems. The "+KN5" column shows the perplexity score after taking a weighted average of a system's predictions and KN5's predictions (both equally weighted). 'TrainingSetFrequency' refers to a system that sets the probability of a token to its frequency of occurrence in the training set. This baseline is outperformed by all other models, suggesting that they are capturing some form of dependency among words when making their predictions. As the table shows, both SPN-3 and SPN-4 outperform all other systems. Note that even though LDA-augmented RNN uses additional information from latent Dirichlet allocation (LDA; which is not used by our SPNs), SPN-3 and SPN-4 still do better by 8.4% and 5.4% respectively on "Individual PPL", and by 16.6% and 16.2% respectively on "+KN5". They have more pronounced improvements over the next best comparison system, RNN (which is a fairer comparison because it does not use information beyond what is available to our SPNs). SPN-3 and SPN-4 outperform RNN by 16.4% and 13.7% respectively on "Individual PPL", and by 22.4% and 22.0% respectively on "+KN5".

We were initially surprised by SPN-3's better performance over SPN-4 (because the latter uses more information and thus should make better predictions). Upon inspecting their perplexity scores on the training set, we found that SPN-4 consistently had lower perplexity than SPN-3 during the later stages of training. This suggests that SPN-4 is overfitting the data. (From Figure 2, we see that SPN-4 has $D \times K + D \times K = 2 \times 10^6$ more parameters than SPN-3, and hence is more likely to overfit.)

[2]Our implementation is publicly available at https://github.com/stakok/lmspn.

Table 1: Perplexity scores (PPL) of different language models.

Model	Individual PPL	+KN5
TrainingSetFrequency	528.4	
KN5 [3]	141.2	
Log-bilinear model [4]	144.5	115.2
Feedforward neural network [5]	140.2	116.7
Syntactical neural network [8]	131.3	110.0
RNN [6]	124.7	105.7
LDA-augmented RNN [9]	113.7	98.3
SPN-3	**104.2**	**82.0**
SPN-4	**107.6**	**82.4**
SPN-4'	**100.0**	**80.6**

To ameliorate this problem, we used the weights of the smaller SPN to guide the weight learning in the larger SPN. We trained an SPN-$(N-1)$ for 10 hours, and used its weights to initialize the corresponding weights in an SPN-N (all other weights are initialized to zero) before training the SPN-N for another 10 hours. We repeated this process for $N = 2, 3, 4$. The final SPN thus obtained uses 4 previous words and is denoted SPN-4'. As Table 1 shows, SPN-4' is the best performing system[3].

Running a test example on our SPNs is typically very fast (sub-second). Our SPNs took less time to train than RNN. To attain the level of KN5's perplexity score, RNN[4] and SPN-4 took about 10 hours and 4 hours to train respectively.

To demonstrate that our SPN can scale to larger data, we trained an SPN-4 for 40 hours on the Brown Laboratory for Linguistic Information Processing 1987-89 WSJ corpus, which is about 40 times larger than Penn Treebank (PTB). We tested this SPN-4 on the same test set (section 23-24 of PTB) and obtained a perplexity of 93.0 (an improvement of 13.6% over the SPN-4 trained on the smaller PTB dataset). This suggests that our model can scale, and can perform better with more data.

To show that our trained SPN is encapsulating useful information, we "seeded" it with some random initial words, and used it to generate a sequence of words. Some examples of the generated word sequences are shown below. These sentences have the "flavor" of news reports, and qualitatively suggest that our SPN is capturing meaningful information from the data.

- IT COULD BE SIMPLY EARNINGS FOR MANY INVESTOR IN THE WATERS FEDERAL CAPITAL
- BUSINESS REGULATORY SAID IT EXPECTS TO ARGUE OWN 'S THREE MEDICAL INVESTMENT IN <unk>

5. Conclusion and Future Work

We presented the first SPN that is used for language modeling. Our proposed SPN is able to contain multiple hidden layers to capture rich dependencies among words, while maintaining tractable inference and training times. Our empirical comparisons with six previous language models on the standard Penn Treebank corpus demonstrate the effectiveness of our SPN.

As future work, we want to combine our SPN language model with an SPN for acoustic modeling to create an integrated speech recognition system. We also want to create a "recurrent" SPN to capture long range dependencies in word sequences.

Acknowledgements. This work is supported by DSO grant DSOCL13083.

[3]Note that the total training time for SPN-4' is also 40 hours, so its better performance is not due to longer training times.

[4]We used the RNNLM Toolkit at http://www.fit.vutbr.cz/ imikolov/rnnlm.

6. References

[1] M. R. Amer and S. Todorovic, "Sum-product networks for modeling activities with stochastic structure," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Patttern Recognition*. Providence, RI: IEEE Computer Society Press, 2012, pp. 1314–1321.

[2] R. Gens and P. Domingos, "Discriminative learning of sum-product networks," in *Advances in Neural Information Processing Systems 25*, Lake Tahoe, Nevada, 2012.

[3] R. Kneser and H. Ney, "Improved backing-off for M-gram language modeling," in *Proceedings of the Twentieth International Conference on Acoustics, Speech, and Signal Processing*, 1995, pp. 181–184.

[4] A. Mnih and G. E. Hinton, "Three new graphical models for statistical language modelling," in *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, 2007, pp. 641–648.

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

[6] T. Mikolov, M. Karafiat, J. Cernocky, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH*, 2010.

[7] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[8] A. Emami and F. Jelinek, "Exact training of a neural syntactic language model," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 2004, pp. I–245–8.

[9] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language model." IEEE Workshop on Spoken Language Technology, Tech. Rep., 2012.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[11] M. Sundermeyer, I. Oparin, J.-L. Gauvain, B. Freiberg, R. Schlüter, and H. Ney, "Comparison of feedforward and recurrent neural network language models," in *International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8430–8434.

[12] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, Barcelona, Spain, 2011.

[13] R. Gens and P. Domingos, "Learning the structure of sum-product networks," in *Proceedings of the Thirtieth International Conference on Machine Learning*. Atlanta, GA: Omnipress, 2013.

[14] A. Darwiche, Ed., *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[15] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: the Penn Treebank," *Computational Linguistics*, vol. 19, pp. 313–330, 1993.