Residual Sum-Product Networks

Fabrizio Ventola VENTOLA@AIPHES.TU-DARMSTADT.DE
Karl Stelzner STELZNER@CS.TU-DARMSTADT.DE
Alejandro Molina MOLINA@CS.TU-DARMSTADT.DE
Kristian Kersting KERSTING@CS.TU-DARMSTADT.DE
Department of Computer Science, TU Darmstadt, Germany

Abstract

Tractable yet expressive density estimators are a key building block of probabilistic machine learning. While sum-product networks (SPNs) offer attractive inference capabilities, obtaining structures large enough to fit complex, high-dimensional data has proven challenging. In this paper, we present a residual learning approach to ease the learning of SPNs, which are deeper and wider than those used previously. The main trick is to ensemble SPNs by explicitly reformulating sum nodes as residual functions. This adds references to substructures across the SPNs at different depths, which in turn helps to improve training. Our experiments demonstrate that the resulting residual SPNs (ResSPNs) are easy to optimize, gain performance from considerably increased depth and width, and are competitive to state-of-the-art SPN structure learning approaches. To combat overfitting, we introduce an iterative pruning technique that compacts models and yields better generalization.

Keywords: tractable inference; structure learning; sum-product networks.

1. Introduction

Arguably one of the most general approaches to unsupervised machine learning is *density estimation*, whereby the goal is to learn the joint probability distribution underlying a given dataset. Given an estimated distribution, classification and regression essentially boil down to probabilistic inference, and one can even generate new data via sampling. Traditionally, joint densities have been specified compactly as probabilistic graphical models (PGMs) using conditional independencies between random variables (RVs). Recently, a number of deep generative models have also been proposed such as variational autoencoders (Kingma and Welling, 2014), generative adversarial networks (Goodfellow et al., 2014), neural autoregressive models (van den Oord et al., 2016), and normalizing flows (Kingma and Dhariwal, 2018). While these models achieve high expressivity by leveraging the representational power of deep neural networks, they are also highly intractable and generally rely on approximate inference methods such as MCMC or variational inference (Hoffman, 2017; Ranganath et al., 2014).

One promising alternative to gain representational efficiency while remaining tractable is to directly leverage *context-specific independencies* in a computational graph for joint probabilities. This is the idea underlying *sum-product networks* (SPNs), a deep but tractable family of density estimators (Poon and Domingos, 2011). In their basic form, SPNs represent distributions as a deep network where the "neurons" are either mixtures (sums), context-specific independencies (products), or primitive input distributions. They are tractable in the sense that any marginal or conditional probability can be computed exactly in time linear in the size of the SPN.

While the parameters of a given SPN can be readily optimized via stochastic gradient descent or expectation-maximization, obtaining a suitable structure is much harder. Approaches such as LearnSPN (Gens and Domingos, 2013) typically rely on independence tests, which are hard to scale to large networks and datasets (Chechetka and Guestrin, 2008). Random SPN structures (Peharz et al., 2019) on the other hand run the risk of making wrong independence assumptions, leading to suboptimality. To address these issues, we explore the combination of random SPNs, residual links, and ensemble methods. Specifically, we start by learning a set of Extremely Random SPNs (ExtraSPNs). However, besides simply adding references only to ensemble components as in random forests (Breiman, 2001) and with the aim to make use of the highly specialized substructures of learned SPNs, we propose to employ *residual links*, i.e., references to nodes in other SPNs. The resulting Residual SPNs (ResSPNs) may be thought of as a probabilistic analog to ResNets (He et al., 2016), refining density estimates made by simple components. Our experimental evidence on a variety of datasets demonstrates that residual links in SPNs indeed improve performance. We proceed as follows. We start by briefly reviewing SPNs. We then introduce ResSPNs, and our SPN iterative pruning technique combating overfitting based on the Lottery Ticket hypothesis (Frankle and Carbin, 2019). Before concluding, we present our experimental evaluation.

2. Background on Sum-Product Networks

Sum-Product Networks (SPNs) are a family of tractable deep density estimators first presented in (Poon and Domingos, 2011). They have been successfully applied on a variety of domains such as computer vision (Amer and Todorovic, 2015), natural language processing (Molina et al., 2017), and speech recognition (Peharz et al., 2014).

Definition of Sum-Product Networks. An SPN S is a computational graph defined by a rooted DAG, encoding a probability distribution[1] $P_{\boldsymbol{X}}$ over a set of RVs $\boldsymbol{X} = \{X_1, \ldots, X_n\}$, where inner nodes can be either sum or product nodes over their children (graphically denoted respectively as $\bigoplus$ and $\bigotimes$), and leaves are univariate distributions defined on one of the RVs $X_i \in \boldsymbol{X}$. Each node $n \in S$ has a *scope* $\mathsf{scope}(n) \subseteq \mathbf{X}$, defined as the set of RVs appearing in its descendant leaves. The subnetwork S_i, rooted at node i, encodes a distribution over its scope i.e. $S_i(\boldsymbol{x}) = P_{\boldsymbol{X}_{|scope(i)}}(\boldsymbol{x})$ for each $\boldsymbol{x} \sim \boldsymbol{X}_{|scope(i)}$. Each edge (i, j) emanating from a sum node i to one of its children j has a non-negative weight w_{ij}, with $\sum_j w_{ij} = 1$. Sum nodes represent a mixture over the probability distributions encoded by their children, while product nodes represent factorizations over contextually independent distributions. In summary, an SPN can be viewed as a deep hierarchical mixture model of different factorizations, where the hierarchy is based on the scope of the nodes w.r.t. the whole set of RVs $\boldsymbol{X}$. In order to encode a valid and normalized probability distribution, an SPN has to fulfill two structural requirements (Poon and Domingos, 2011). First, the scopes of the children of each product node need to be disjoint (*decomposability*). Second, the scopes of the children of each sum node need to be identical (*completeness*). In a valid SPN, the probability assigned to a given state $\boldsymbol{x}$ of the RVs $\boldsymbol{X}$ can be read out at the root node, and will be denoted $S(\boldsymbol{x}) = P_{\boldsymbol{X}}(\boldsymbol{X} = \boldsymbol{x})$.

Probabilistic Inference within SPNs. Given an SPN S, $S(\boldsymbol{x})$ can be computed by evaluating the network bottom-up. When evaluating a leaf node i concerning variable X_j, $S_i(x_j)$ corresponds to the probability of that state $P_i(X_j = x_j)$. The value of a product node corresponds to the product of its children's values: $S_i(\boldsymbol{x}_{|\mathsf{scope}(i)}) = \prod_{i \to j \in S} S_j(\boldsymbol{x}_{|\mathsf{scope}(j)})$; while, for a sum node, its value corresponds to the weighted sum of its children's values: $S_i(\boldsymbol{x}_{|\mathsf{scope}(i)}) =$

1. We are not strict on "density" vs. "distribution."

Algorithm 1 LearnSPN($\mathcal{D}$, $\boldsymbol{X}$, μ, ρ)

Require: a set of samples $\mathcal{D}$ over RVs $\boldsymbol{X}$; μ: minimum number of instances to split; ρ: statistical independence threshold

Ensure: an SPN S encoding a pdf over $\boldsymbol{X}$ learned from $\mathcal{D}$

if $|\boldsymbol{X}| = 1$ **then**
  $S \leftarrow \mathsf{univariateDistribution}(\mathcal{D}, \boldsymbol{X})$
else if $|\mathcal{D}| < \mu$ **then**
  $S \leftarrow \mathsf{fullFactorization}(\mathcal{D}, \boldsymbol{X})$
else
  $\{\boldsymbol{X}_{d_1}, \boldsymbol{X}_{d_2}\} \leftarrow \mathsf{splitFeatures}(\mathcal{D}, \boldsymbol{X}, \rho)$
  if $|\boldsymbol{X}_{d_2}| > 0$ **then**
    $S \leftarrow \prod_{j=1}^{2} \mathsf{LearnSPN}(\mathcal{D}, \boldsymbol{X}_{d_j}, \mu, \rho)$
  else
    $\{\mathcal{D}_i\}_{i=1}^{R} \leftarrow \mathsf{clusterInstances}(\mathcal{D}, \boldsymbol{X})$
    $S \leftarrow \sum_{i=1}^{R} \frac{|\mathcal{D}_i|}{|\mathcal{D}|} \mathsf{LearnSPN}(\mathcal{D}_i, \boldsymbol{X}, \mu, \rho)$
return S

$\sum_{i \to j \in S} w_{ij} S_j(\boldsymbol{x}_{|\mathsf{scope}(j)})$. All the marginal probabilities, the partition function, and even approximate MPE queries and states can be computed in time linear in the *size* of the network i.e. its number of edges as shown in (Gens and Domingos, 2013).

Structure Learning of SPNs. The prototypical structure learning algorithm for SPNs is LearnSPN (Gens and Domingos, 2013). It provides a simple greedy learning schema to infer both the structure and the parameters of an SPN by executing a greedy top-down structure search in the space of *tree-structured* SPNs, as summarized in Alg. 1. Specifically, LearnSPN recursively partitions the training data matrix $\mathcal{D}$ consisting of i.i.d instances over $\boldsymbol{X}$, the set of columns, i.e., the RVs. For each call on a data slice, LearnSPN first tries to split the data slice by columns. This is done by splitting the current set of RVs into different groups such that the RVs in a group are statistically dependent while the groups are independent, i.e., the joint distribution factorizes over the groups of RVs. We denote this procedure as splitFeatures. If no independencies among features/RVs are found, i.e. the splitting fails, LearnSPN tries to cluster similar data slice rows (procedure clusterInstances) into groups of similar instances. In the original work of Gens and Domingos (2013), an online version of hard expectation-maximization (EM) algorithm is employed for this clustering step. Depending on the assumptions on the distribution of $\mathbf{X}$, other clustering and splitting algorithms may be more suitable and may be employed instead (Vergari et al., 2015; Molina et al., 2017, 2018). When a column split succeeds, LearnSPN adds a product node to the network whose children correspond to partitioned data slices. Similarly, after a row clustering step, it adds a sum node where children weights represent the proportions of instances falling into the obtained clusters. LearnSPN stops in two cases: (1) when the current data slice contains only one column or (2) when the number of its rows falls under a certain threshold μ. In the first case, a leaf node, representing a univariate distribution, is introduced by a maximum likelihood estimation from the data slice. In the second case, the data slice's RVs are modeled as a full factorization: They are assumed to be independent and a product node is put over a set of univariate leaf nodes (each estimated as described for the first case).

Indeed, clustering and splitting influence each other in terms of quality (Vergari et al., 2015). If a good instance clustering is achieved, then it is likely to enhance the variable splitting and the other way around. This holds for more advanced structure learning approaches too. For instance, Peharz et al. (2013) introduced a bottom-up approach to learn SPN structures, using an information-bottleneck method. Vergari et al. (2015) employed multivariate leaves for regularization. Rahman and Gogate (2016) compressed tree-shaped structures into general DAGs. Di Mauro et al. (2018) investigated approximate independence testing, and Molina et al. (2018) non-parametric independence tests for learning SPN structures over hybrid domains. Rooshenas and Lowd (2014) refined LearnSPN by learning leaf distributions using Markov networks represented by arithmetic circuits. The resulting learner, called ID-SPN, is state-of-the-art in density estimation on binary data when considering singleton models. Of course, any of the structure learning approaches can be improved by ensemble (Conaty et al., 2019) and boosting methods (Liang et al., 2017; Di Mauro et al., 2017).

3. Residual Sum-Product Networks

While SPNs have attractive inference properties, scaling them in a manner similar to deep neural networks has proven challenging. The structural constraints, which SPNs have to obey in order to guarantee validity, are a major reason for this. They necessitate the careful design of structures, either by hand or through learning from data. Consequently, learning SPN structures (as sketched by LearnSPN) has been proven hard to scale in practice, since determining how to split variables is costly: Ideally, one would always determine the two subsets with minimum empirical mutual information. This takes cubic time (Gens and Domingos, 2013), which is much too slow. Therefore, one typically considers only pairwise dependencies in practice, reducing the cost to quadratic time, which is unfortunately still prohibitive for many applications. Recently, Peharz et al. (2019) proposed random tensorized SPNs (RAT-SPNs), which forgo structure learning in favor of randomly generated structures. While this works surprisingly well, it carries the risk of introducing false independency assumptions. To repair such mistakes, we propose to employ residual learning between a set of SPNs and then, similarly to Random Forests (Breiman, 2001), we ensemble the resulting network with its components.

Step 1: Ensemble SPNs. In principle, any SPN learned via structure learning (potentially in combination with bagging) or just RAT-SPNs can be used as components in this ensemble. Here, we opt for a faster and even simpler method: Akin to extremely randomized trees (Geurts et al., 2006), each individual SPN is trained using the whole learning sample and the top-down splitting in LearnSPN is randomized. That is, instead of computing the locally optimal splitting of RVs based on, e.g., mutual information or independence tests, a random split is selected in Line 6 in Alg. 1. The resulting *extremely randomized SPNs* (ExtraSPNs) are similar in spirit to extremely randomized cutset networks (Di Mauro et al., 2017), except that ExtraSPNs do not use any statistical test for any splitting procedure. Instead, we fix a parameter β to set the probability of the splitting test to fail, and randomly group the features in two clusters. In this way, the splitting procedure has linear complexity instead of the quadratic complexity of the G-test employed in the original LearnSPN. Moreover, to foster diversity in the set of ExtraSPNs, we sample the parameter μ at random from the range $[1, |\mathcal{D}|/\gamma]$ for each SPN. Here, $\mathcal{D}$ is the dataset and γ is a hyperparameter that can shrink the range of the possible values of μ. When γ is greater than 1 then the range of the possible values for μ is smaller, in this way, we can both tune the learning times of the ExtraSPNs and also their depth.

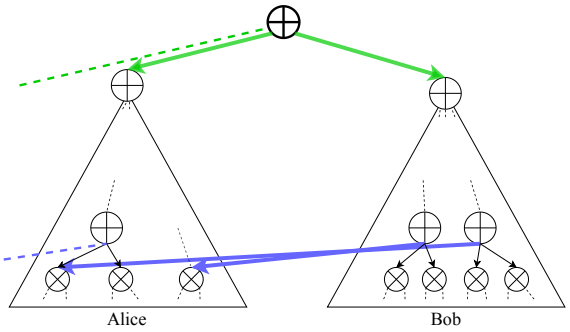

Figure 1: Residual Sum-Product Learning. The blue edges denote residual links between the right (Bob) and left (Alice) SPN, which let Bob directly fit a residual. Green edges realize the global mixture.

Step 2: Residual Sum-Product Learning. SPNs can be seen as a set of hierarchical mixtures of context-specific subdistributions. Thus, an ensemble method for SPNs should also work at the level of these substructures. Instead of hoping that each SPN fits a desired contribution to the global joint distribution, we explicitly let them fit residuals. Consider two SPNs called Alice and Bob as shown in Fig. 1. Our goal is to improve the performance of one of Bob's sum nodes using a node $a(\boldsymbol{z})$ of the same scope from Alice. We denote Bob's desired density as $b(\boldsymbol{z})$. Instead of learning this density directly, we instead let Bob fit a residual, namely, $r(\boldsymbol{z}) = b(\boldsymbol{z}) - a(\boldsymbol{z})$, up to proper weighting. That is, Bob's original density is recast into $b(\boldsymbol{z}) = a(\boldsymbol{z}) + r(\boldsymbol{z})$, up to proper weighting. This can be achieved in a simple way: We add Alice's product node as a child to Bob's sum node, cf. Fig. 1. Adding such residual links between individual SPNs does not affect the validity of individual SPNs, since the scope of the two nodes is the same — the sum-product networks remain complete. For additional flexibility, we can also pick one of Alice's nodes a with $\mathsf{scope}(b) \subset \mathsf{scope}(a)$, and marginalize the surplus RVs from a before adding the connection. The weights of the new mixture components can be set proportionally to their training data slices or eventually optimized jointly in a supplementary parameter estimation step. This may even take place after mixing Alice and Bob additionally using a single sum node at the top. Thus, we propose to apply residual sum-product learning to every ExtraSPN. To do so, we generate an array of ExtraSPNs $E_1, E_2, \ldots, E_n$, pick a random one E_i, and introduce residual links between it and each other SPN E_j. We then add a top sum node connecting all the ensemble components and finally train all parameters jointly. We name the resulting model residual SPN (ResSPN).

Step 3: Sparsification using Lottery Ticket Pruning. Since ResSPNs, as ensemble models, can have a considerable size compared to the models learned with state-of-the-art structure learners, we explore the application of a recent successful pruning technique for deep models based on the "Lottery Ticket Hypothesis" (Frankle and Carbin, 2019). This is an iterative pruning technique i.e. it iteratively trains, prunes lowest magnitude weights and then resets the weights of the network to their initial values from before the training. At each iteration, the percentage of the weights to be removed is reduced. It has been shown to be very effective on deep neural networks (DNNs) by being able to considerably reduce the size of the network, reaching better or similar accuracy with only 10-20% of the size of the original model. Specifically, the authors showed that this technique

Algorithm 2 ResSPN($\mathcal{D}$, $\boldsymbol{S}$, k)

Require: a set of samples $\mathcal{D}$; a set of input SPNs to be combined $\boldsymbol{S} = \mathcal{S}_1, \ldots, \mathcal{S}_n$; the ratio of nodes from which to draw residual links k

Ensure: a $ResSPN$ built from $\boldsymbol{S}$ and optimized on $\mathcal{D}$

```
ResSPN ← copy(RandomElement(S))
for S ∈ S \ ResSPN do
    η = 0
    for s_1 ∈ BFS(ResSPN) do
        for s_2 ∈ BFS(S) do
            if scope(s_1) ⊆ scope(s_2) then
                s_2' ← Marginalize(s_2, scope(s_1))
                Add connection from s_1 to s_2'
                η ← η + 1
        if η > k|ResSPN| then
            break
ResSPN ← buildMixture(D, S ∪ {ResSPN})
return ResSPN
```

is able to find the "winning tickets", i.e., those subnetworks which are responsible for the majority of the network's performance thanks to well-initialized weights. They showed that the randomly initialized weights of these networks resemble winning lottery tickets, since the same or better accuracy can be achieved consistently after pruning only when retraining from their initial state. To apply this technique to our ResSPNs, and generally to any SPN, we have to take into account that SPNs are not as dense as the DNNs used originally with this technique, and that we need to preserve the validity of the SPN. Therefore, to adapt it to SPNs, we make a number of adjustments: 1) At each iteration, we prune a fraction of sum node children with the lowest weight magnitude, keeping the SPN valid. We do not consider the root node children for pruning[2]. 2) We reset the weights by setting the surviving ones to the values they had in the original model before the optimization, and then renormalize. 3) We optimize the model and iterate until no more nodes are pruned.

Discussion. ResSPN is a sensible idea to follow. It is easier to optimize a residual density than to optimize the original, unreferenced density. To the extreme, if a density from Alice was optimal, it would be easier to push the residual in Bob to zero than to fit the density in Bob from scratch or by mixing Alice and Bob at the top node only. As a result, we build a stronger density estimator by referencing local *context-specific* substructures. In fact, residual connections make SPNs wider as also suggested by Fig. 1 and, in turn, increase the variance in sizes of training data slices used to train mixture components. This is akin to what is known for residual (neural) networks (Veit et al., 2016) but now we carried this over to hierarchical mixture models in the form of SPNs. Different procedures are imaginable for selecting the pairs of nodes between which edges are to be added. For this reason, one can see the residual sum-product learning as a general schema, where different strategies for adding residual links may be selected based on the problem at hand. Here, we adopt a randomized greedy approach as summarized in Alg. 2. From the set of input SPNs, we

2. In the original version a similar strategy is applied: the output layer is pruned at half of the rate of the current pruning rate. We also need to consider that, differently from DNNs, when we prune a sum node child in an SPN we might prune all its descendants, also.

	$\lvert\boldsymbol{X}\rvert$	$\lvert T_{train}\rvert$	$\lvert T_{val}\rvert$	$\lvert T_{test}\rvert$
NLTCS	16	16181	2157	3236
MSNBC	17	291326	38843	58265
Plants	69	17412	2321	3482
Audio	100	15000	2000	3000
Jester	100	9000	1000	4116
Netflix	100	15000	2000	3000

Table 1: Statistics of the datasets used in the experimental evaluation.

randomly select one to which all future edges will be added. Then, we iteratively add references to each of the remaining input SPNs S, where the maximum percentage of nodes from which to draw connections in each step is given by the hyperparameter k (usually no more than 10% or 20%). The connections are drawn in a greedy fashion: We start by selecting candidate nodes from our ResSPN by traversing it via breadth-first-search (BFS), excluding the root. For each visited node, we try to find a suitable partner in S, by also traversing it via BFS. If we find a suitable pairing, i.e. one where the scopes match as described above, we draw a residual link. We employ this k-limited BFS since it conveniently maximizes the chance to find compatible scopes given that scopes at top levels are generally larger and it does not blow-up in quadratic complexity. Finally, we sparsify the overparameretized ResSPNs.

4. Experimental Evidence

Our intention here is to investigate the pros and cons of ResSPNs. Along the way, we also explore how SPNs benefit from wider and deeper structures. Specifically, we examine the following questions: **(Q1)** Can ResSPNs improve upon singleton SPNs being an effective ensemble method for SPNs? **(Q2)** Can ResSPNs repair wrong independencies? **(Q3)** Do ResSPNs contain subnetworks that can reach a test accuracy comparable to the original ResSPNs, i.e., can we sparsify them?

To answer these questions, we learned ResSPNs on standard binary benchmarks summarized in Tab. 1, and split into training, validation and test sets. For parameter estimation we used the EM algorithm. The datasets are a subset of the ones used e.g. in (Gens and Domingos, 2013) and (Rooshenas and Lowd, 2014). All algorithms were implemented in Python[3] making use of the *SPFlow* library (Molina et al., 2019). All experiments were run on a DGX-2 system.

ResSPNs can improve upon singleton SPNs – (Q1). We learned ResSPNs by choosing the number n of ensemble components from $\{3, 5, 10\}$. For learning the ExtraSPNs—the ensemble components—we took the μ hyperparameter at random in the range $[1, |D|/5]$ and fixed β to 0.6. We similarly randomized the instance clustering step using the same hyperparameter β. The ResSPN was then built as described in Alg. 2. We optimized the parameters with EM with a limit of 1000 iterations. The optimization stopped if the variance of the training likelihood on the last 5 iterations was lower than 1e-7. We performed a hyperparameter search over k within the range $[0.1, 0.2]$, and selected the models with the best accuracy on a separate validation set. Then, to assess the accuracy of learned estimators, we computed the average log-likelihood on the test sets. We compare it with the ones obtained with LearnSPN as reported in (Rooshenas and Lowd, 2014). Tab. 2 shows the average test likelihood for LearnSPN and ResSPN built with 10 ExtraSPNs. One can see that ReSPNs is competitive compared to LearnSPN since it can achieve similar accuracy

3. Code at https://github.com/fabriziov/resspn

	Test Accuracy			
	Baseline Singleton SPN Learners			Ensemble SPN
	Best ExtraSPN	LearnSPN	ID-SPN	ResSPN
NLTCS	-6.153	-6.11	**-6.02**	-6.040⇑
MSNBC	-6.433	-6.11	**-6.04**	-6.097⇑
Plants	-16.683	-12.98	**-12.54**	-13.908•
Audio	-42.639	-40.5	**-39.79**	-40.762•
Jester	-55.335	-53.48	**-52.86**	-53.956•
Netflix	-60.330	-57.33	**-56.36**	-57.867•

Table 2: Average test log-likelihood of ResSPN compared to baseline singleton SPN learners. The best results, including the baselines, are shown in bold. As one can see, ID-SPN gives the best results. However, it uses Markov networks represented as ACs in the leaves. More importantly, ResSPNs improve upon ExtraSPNs, as denoted by "•", sometimes getting even better than LearnSPN, denoted as "⇑".

	Test Accuracy					
	With Residual Links			Without Residual Links		
	ResSPN 3	ResSPN 5	ResSPN 10	RSPF 3	RSPF 5	RSPF 10
NLTCS	-6.118•	-6.068•	-6.040•	-6.192	-6.109	-6.046
MSNBC	-6.235•	-6.145•	-6.097•	-6.316	-6.225	-6.104
Plants	-14.732•	-14.631•	-13.908•	-15.616	-15.161	-14.573
Audio	-41.492•	-41.433•	-40.762•	-41.883	-41.482	-40.833
Jester	-53.772•	-54.040	-53.995	-53.987	-53.734•	-53.885•
Netflix	-58.609•	-58.509•	-57.867•	-59.121	-58.570	-57.900
wins	6/6	5/6	5/6	0/6	1/6	1/6

Table 3: Average test log-likelihood of ResSPN with ensembles of 3, 5, and 10 ExtraSPNs compared to the simple shallow mixture RSPFs. The best results are denoted using "•". Overall, one can clearly see that adding residual links is beneficial. The only exception is Jester, where ResSPNs overfit.

and is more accurate on NLTCS and MSNBC without involving a meticulous hyperparameter selection. Notably, ResSPN accuracy is comparable to ID-SPN, which is the most accurate method so far. However, this comparison is rather unfair since ID-SPN employs Markov Networks as multivariate leaves and this also makes learning intractable and difficult to scale with the need for an attentive hyperparameters selection. Thus, we use ID-SPN as our accuracy gold standard. Moreover, we compared ResSPNs to the single best available ExtraSPN. To do so, we generated a set of ExtraSPNs as before, except that we included a proper clustering step for learning sum nodes as opposed to random clustering. We then optimized their parameters with EM and reported the best test log-likelihood achieved among all of them. By looking at the results in Tab. 2, one can see that the ResSPNs always outperform the best ExtraSPN[4]. Thus, ResSPNs do indeed improve upon its components, even when the latter are augmented using proper clustering and optimization. Therefore, ResSPNs improve upon singleton SPNs.

To investigate the effectiveness of our ensemble method based on residual links, we compared ResSPNs with simple ensembles built starting from the same sets of ExtraSPNs and connecting them in a shallow mixture model by a top sum node. We refer to these simple models as Random

	LearnSPN		ResSPN			LTP	
	layers	edges	layers	edges	test accuracy	edges	test accuracy
NLTCS	4	7509	**15**	14972	-6.04•	**13493**	-6.04•
MSNBC	4	22350	**13**	7420	-6.10•	**6946**	-6.10•
Plants	6	55668	**19**	58490	-13.91•	**37644**	-13.97
Audio	8	70036	**24**	138392	-40.76	**93324**	-40.75•
Jester	4	36528	**21**	92174	-53.96	**65956**	-53.71•
Netflix	4	17742	**24**	81084	-57.87	**54951**	-57.80•

Table 4: The structure of ResSPNs can be deeper and/or wider compared to singleton tree-SPNs learned via LearnSPN as reported in (Vergari et al., 2015). Sparsifying ResSPNs: Lottery Ticket Pruning (LTP) can considerably reduce the size of the model (smaller models are bold) without sacrificing considerably performance (• denotes best value), especially on Jester, a dataset where ResSPNs overfit.

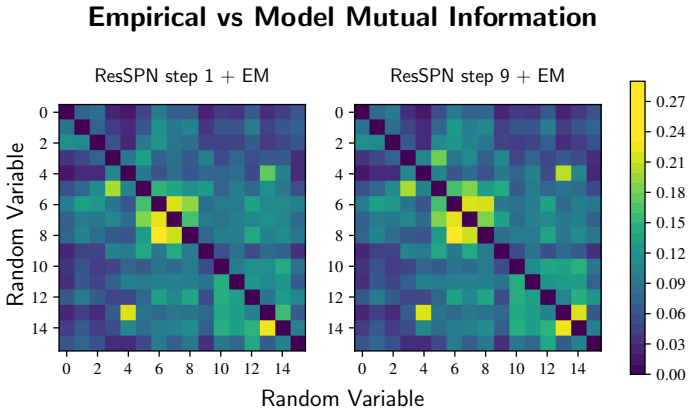

Figure 2: The lower triangle of the matrices shows the pairwise empirical mutual information between variables in the NLCTS dataset. (Upper triangle, left) MI computed with a ResSPN after adding links to one other SPN. (Upper triangle, right) MI computed with a ResSPN after adding links to 9 other SPNs. The MI of the fully trained ResSPN is much closer to the empirical MI, indicating that ResSPNs are able to repair wrong independence assumptions.

Sum-Product Forests (RSPF). Tab. 3 summarizes the average test log-likelihood of ResSPNs and RSPFs[4] comparing the models built with 3, 5, and 10 components. As one can clearly see, residual links generally[5] improve the test accuracy, with Jester being the only exception. The dataset appears to be generally prone to overfitting, as even for RSPFs better results were achieved with fewer than 10 components. In summary, residual links generally yield better test accuracy. These observations also hold when a smaller number of ExtraSPNs are used. Moreover, employing more components yields better accuracy. This further highlights the effectiveness of our ensemble method. Given this,

4. The difference is statistically significant according to a Wilcoxon signed rank test with $p = 0.05$.
5. For ResSPNs built with 5 components, the difference is not statistically significant only on Audio and Netflix, for the ones built with 10 components is not statistically significant only on NLTCS and Netflix.

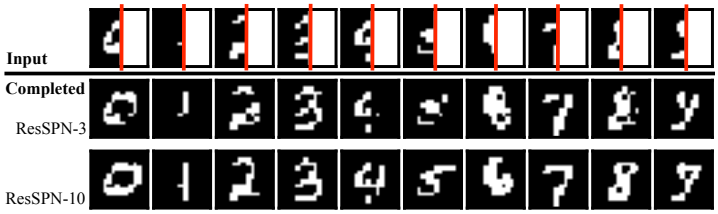

Figure 3: Image completion on Binary MNIST. (Top) test images without the right half. (Middle resp. bottom) completed images using a ResSPN with 3 resp. 10 component SPNs. Both ResSPNs were optimized using 10 EM iterations. As one can see, the deeper and wider ResSPN completes the test images in a more realistic way.

we can answer **(Q1)** affirmatively. ResSPN learning shows also to be particularly efficient on large datasets, in fact, compared to ID-SPN, it achieves comparable accuracy on 20 NewsGroup (11k instances, 910 features) and on our MNIST (see the following paragraph) being more than 10X faster. Additionally, by considering that ensembles are larger than their components, and by taking into account the structural details of the learned models shown in Tab. 4, we can indeed confirm that wider and deeper SPNs —learned by adding residual links— in general, perform considerably better than singleton tree-shaped models.

ResSPNs can repair wrong independencies – (Q2). ResSPNs are also capable of repairing wrong independence assumptions made by the singleton models. To demonstrate this, we compared the empirical pairwise mutual information between RVs on the training dataset with the one computed on our model at various stages of training. Fig. 2 depicts the empirical pairwise mutual information on the NLTCS dataset on the lower triangle of the matrices, and the mutual information computed from our model on the upper triangle. The left matrix uses a ResSPN in early iterations, after the first set of residual links has been added, while the right one shows the result for a fully trained ResSPN based on 10 ExtraSPNs. Both ResSPNs are optimized with EM for 10 iterations. It is apparent that after full training, the ResSPN is much more capable of representing the correlations among the RVs, since the matrix on the right is almost symmetric. Furthermore, in order to have a visual understanding of our results, we trained a ResSPN with 3 components and a ResSPN with 10 components on a binarized version of MNIST resizing the images to 14 by 14 pixels. We trained the models on 50k images and left apart 10k images for validation and 10k as test set. We optimized both models with EM for 10 iterations and we completed images with MPE giving the left half of test images as input. Besides obtaining a better average test log-likelihood, the ResSPN composed with 10 components is able to generate more realistic images as one can clearly see in Fig. 3. Thus, we can answer **(Q2)** affirmatively.

Sparse ResSPN: Lottery Ticket Pruning – (Q3). We pruned the ResSPNs built with 10 components starting pruning the 20% and 40% of the weights with lowest magnitude. After resetting the weights, we optimize the network with EM for 10 iterations. Then, we selected the pruned ResSPNs with the best likelihood on the validation set. As one can observe from Tab. 4, with Lottery Ticket Pruning, we obtain more compact networks that achieve similar or even better accuracy than the original models, especially on Jester where ResSPN overfits. Thus, we can answer (**Q3**) affirmatively. To summarize, all questions **(Q1)**-**(Q3)** can be answered affirmatively, i.e. residual learning can ease the training of SPNs.

5. Conclusion

Deeper and wider probabilistic models are more difficult to train. To make this task easier, we have introduced an ensemble learning framework for sum-product networks (SPNs), which makes it possible to combine arbitrary groups of SPNs into a larger model. Our experimental evidence shows that even when the input SPNs are generated randomly, the resulting ResSPNs are competitive with state-of-the-art SPN learners. ResSPNs can be seen as a general schema for building ensembles of SPNs and thus provide several interesting avenues for future work. One is to explore structure learners instead of ExtraSPN, e.g. ID-SPN, or even a mix of different ones. Furthermore, one should explore other strategies for adding residual links. For instance, given that we have a tractable model at hand, one may compute the expected value of adding a residual link. Employing dropout, as well as online learning, are further interesting avenues.

Acknowledgments

FV and KK acknowledge the support by the German Research Foundation (DFG) as part of the Research Training Group Adaptive Preparation of Information from Heterogeneous Sources (AIPHES) under grant No. GRK 1994/1. KK also acknowledges the support of the Rhine-Main Universities Network for "Deep Continuous-Discrete Machine Learning" (DeCoDeML).

References

M. Amer and S. Todorovic. Sum product networks for activity recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2015.

L. Breiman. Random forests. *Machine Learning*, pages 5–32, 2001. ISSN 0885-6125.

A. Chechetka and C. Guestrin. Efficient principled learning of thin junction trees. In *Proc. of NIPS*, pages 273–280, 2008.

D. Conaty, J. M. del Rincón, and C. P. de Campos. A hierarchy of sum-product networks using robustness. *Int. J. Approx. Reason.*, 113:245–255, 2019.

N. Di Mauro, A. Vergari, T. Basile, and F. Esposito. Fast and accurate density estimation with extremely randomized cutset networks. In *Proc. of ECML/PKDD*, 2017.

N. Di Mauro, F. Esposito, F. G. Ventola, and A. Vergari. Sum-product network structure learning by efficient product nodes discovery. *Intelligenza Artificiale*, pages 143–159, 2018.

J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proc. of ICLR*, 2019.

R. Gens and P. Domingos. Learning the Structure of Sum-Product Networks. In *Proc. of the ICML*, pages 873–880, 2013.

P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 2006.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proc. of NIPS*, pages 2672–2680, 2014.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of CVPR*, pages 770–778, 2016.

M. D. Hoffman. Learning deep latent Gaussian models with Markov chain Monte Carlo. In *Proc. of ICML*, pages 1510–1519, 2017.

D. P. Kingma and P. Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Proc. of NIPS*, pages 10236–10245, 2018.

D. P. Kingma and M. Welling. Auto-encoding variational Bayes. In *Proc. of ICLR*, 2014.

Y. Liang, J. Bekker, and G. Van den Broeck. Learning the structure of probabilistic sentential decision diagrams. In *Proc. of UAI*, 2017.

A. Molina, S. Natarajan, and K. Kersting. Poisson sum-product networks: A deep architecture for tractable multivariate poisson distributions. In *Proc. of AAAI*, 2017.

A. Molina, A. Vergari, N. D. Mauro, S. Natarajan, F. Esposito, and K. Kersting. Mixed sum-product networks: A deep architecture for hybrid domains. In *Proc. of AAAI*, pages 3828–3835, 2018.

A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N. D. Mauro, P. Poupart, and K. Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. *ArXiv*, abs/1901.03704, 2019.

R. Peharz, B. Geiger, and F. Pernkopf. Greedy Part-Wise Learning of Sum-Product Networks. In *Proc. of ECML-PKDD*, 2013.

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

R. Peharz, A. Vergari, K. Stelzner, A. Molina, M. Trapp, K. Kersting, and Z. Ghahramani. Random Sum-Product Networks: A Simple and Effective Approach to Probabilistic Deep Learning. *Proc. of UAI*, 2019.

H. Poon and P. Domingos. Sum-Product Networks: a New Deep Architecture. *Proc. of UAI*, 2011.

T. Rahman and V. Gogate. Merging strategies for sum-product networks: From trees to graphs. In *Proc. of UAI*, 2016.

R. Ranganath, S. Gerrish, and D. Blei. Black box variational inference. In *Proc. of AISTATS*, 2014.

A. Rooshenas and D. Lowd. Learning Sum-Product Networks with Direct and Indirect Variable Interactions. In *Proc. of ICML*, 2014.

A. van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves, et al. Conditional image generation with pixelcnn decoders. In *Proc. of NIPS*, pages 4790–4798, 2016.

A. Veit, M. J. Wilber, and S. J. Belongie. Residual networks behave like ensembles of relatively shallow networks. In *Proc. of NIPS*, pages 550–558, 2016.

A. Vergari, N. Di Mauro, and F. Esposito. Simplifying, Regularizing and Strengthening Sum-Product Network Structure Learning. In *Proc. of ECML-PKDD*, 2015.