Sum-Product Networks*
The Third Wave of Differentiable Programming

*Thanks for Pedro Domingos for making his slides publicly available

AI has impact

Data are now ubiquitous; there is great value from understanding this data, building models and making predictions.

However, data is not everything.
The third wave of AI

Data are now ubiquitous; there is great value from understanding this data, building models and making predictions.

However, data is not everything.

AI systems that can acquire human-like communication and reasoning capabilities, with the ability to recognise new situations and adapt to them.
Potentially much more powerful than shallow architectures, represent computations

DNNs often have no probabilistic semantics. They are not calibrated joint distributions. \[ P(Y|X) \neq P(Y,X) \]

Many DNNs cannot distinguish the datasets.


[Peharz, Vergari, Molina, Stelzner, Trapp, Kersting, Ghahramani UDL@UAI 2018]
The third wave of differentiable programming

Getting deep systems that know when they do not know and, hence, recognise new situations and adapt to them.
Can we borrow ideas from differentiable programming for probabilistic graphical models?

Judea Pearl, UCLA
Turing Award 2012
Alternative Representation: Graphical Models as (Deep) Networks

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$P(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

$$P(X) = 0.4 \cdot I[X_1=1] \cdot I[X_2=1]$$
$$+ 0.2 \cdot I[X_1=1] \cdot I[X_2=0]$$
$$+ 0.1 \cdot I[X_1=0] \cdot I[X_2=1]$$
$$+ 0.3 \cdot I[X_1=0] \cdot I[X_2=0]$$
Alternative Representation: Graphical Models as (Deep) Networks

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$P(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

$P(X) = 0.4 \cdot I[X_1=1] \cdot I[X_2=1]$
$+ 0.2 \cdot I[X_1=1] \cdot I[X_2=0]$
$+ 0.1 \cdot I[X_1=0] \cdot I[X_2=1]$
$+ 0.3 \cdot I[X_1=0] \cdot I[X_2=0]$
Shorthand using Indicators

\[
P(X) = 0.4 \cdot X_1 \cdot X_2 \\
+ 0.2 \cdot X_1 \cdot \overline{X}_2 \\
+ 0.1 \cdot \overline{X}_1 \cdot X_2 \\
+ 0.3 \cdot \overline{X}_1 \cdot \overline{X}_2
\]

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( P(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.3</td>
</tr>
</tbody>
</table>
Summing Out Variables

Let us say, we want to compute $P(X_1 = 1)$

$$P(e) = 0.4 \cdot X_1 \cdot X_2$$
$$+ 0.2 \cdot X_1 \cdot \overline{X}_2$$
$$+ 0.1 \cdot \overline{X}_1 \cdot X_2$$
$$+ 0.3 \cdot \overline{X}_1 \cdot \overline{X}_2$$

Set $X_1 = 1$, $\overline{X}_1 = 0$, $X_2 = 1$, $\overline{X}_2 = 1$

Easy: Set both indicators of $X_2$ to 1
This can be represented as a computational graph

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$P(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

network polynomial
However, the network polynomial of a distribution might be exponentially large

Example: Parity
 Uniform distribution over states with even number of 1’s
Make the computational graphs deep

Example: Parity
Uniform distribution over states with even number of 1’s

Induce many hidden layers
Reuse partial computation
Sum-Product Networks* a deep probabilistic learning framework [Poon, Domingos UAI 2011]

A SPN $S$ is a rooted DAG where nodes are sum, product, input indicator and weights are on edges from the sums of children

*SPNs are an instance of Arithmetic Circuits (ACs). ACs have been introduced into the AI literature more than 15 years ago as a tractable representation of probability distributions [Darwiche CACM 48(4):608-647 2001]
Valid SPN: General Conditions

SPN is valid if \( S(e) = \sum_{X \sim e} S(X) \) for all \( e \). If so, we can compute (conditional) marginals efficiently since the partition function \( Z \) can be computed by setting all indicators to 1.

**Theorem:** SPN is valid if it is **complete** & **consistent**

**Complete:** Under sum, children cover the same set of variables

**Consistent:** Under product, no variable in one child and negation in another

\[
S(e) \leq \sum_{X \sim e} S(X)
\]

\[
S(e) \geq \sum_{X \sim e} S(X)
\]
**Inference: Linear in Size of Network**

As long as weights sum to 1 at each sum node

\[ P(X) = S(X) \]

**X:** \( X_1 = 1, X_2 = 0 \)

| \( X_1 \) | 1 |
| \( \overline{X}_1 \) | 0 |
| \( X_2 \) | 0 |
| \( \overline{X}_2 \) | 1 |

How to set the indicator variables

Kristian Kersting - Sum-Product Networks: The Third Wave of Differentiable Programming
Inference: **Linear in Size of Network**

**Marginal:** \( P(e) = S(e) \)

\[ e: X_1 = 1 \]

| \( X_1 \) | 1 |
| \( \overline{X}_1 \) | 0 |
| \( X_2 \) | 1 |
| \( \overline{X}_2 \) | 1 |

How to set the indicator variables:

\[ 0.69 = 0.51 + 0.18 \]
Inference: Linear in Size of Network

**MAP: Replace sums with maxs**

<table>
<thead>
<tr>
<th></th>
<th>$X_1$</th>
<th>$\overline{X}_1$</th>
<th>$X_2$</th>
<th>$\overline{X}_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e: X_1 = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_1$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\overline{X}_1$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\overline{X}_2$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$0.7 \times 0.42 = 0.294$

$0.3 \times 0.72 = 0.216$

How to set the indicator variables:

Kristian Kersting - Sum-Product Networks: The Third Wave of Differentiable Programming
Building challenging multivariate distributions from well-known univariate distributions with flexible correlations, here multivariate Poisson distribution.
And also learning is simple. E.g. we can learn (the structure) via parameter estimation assuming a fixed network (like in Deep Neural Learning)

- Start with a dense SPN
- Find the structure by (online) learning weights
  Zero weights signify absence of connections
- (Hard) EM beneficial to avoid gradient vanishing
  Each sum node is a mixture over children

In principle you can turn a given SPN into, say, a TensorFlow computation graph and apply any known algorithm from there
Or we learn directly (Tree-)SPNs

Testing independence using a (non-parametric) independency test
Or we learn directly (Tree-)SPNs

Testing independence using a (non-parametric) independency test

E.g. for Poisson RVs: Learn Poisson model trees for $P(x|V-x)$ and $P(y|V-y)$. Check whether $X$ resp. $Y$ is significant in $P(y|V-x)$ resp. $P(x|V-y)$

In general use the independency test for your random variables at hand such as g-test for Gaussians


Kristian Kersting - Sum-Product Networks: The Third Wave of Differentiable Programming
Or we learn directly (Tree-)SPNs

Testing independence using a (non-parametric) independency test

In general some clustering for your random variables at hand such as kMeans for Gaussians

Mixture of Poisson Dependency Networks or random splits

Word

Documents

Word Counts
Or we learn directly (Tree-)SPNs

Testing independence using a (non-parametric) independency test

Clustering or random splits

keep growing alternatingly and + layers

[Kristian Kersting - Sum-Product Networks: The Third Wave of Differentiable Programming]
SPFlow: An Easy and Extensible Library for Sum-Product Networks

SPFlow, an open-source Python library providing a simple interface to inference, learning and manipulation routines for deep and tractable probabilistic models called Sum-Product Networks (SPNs). The library allows one to quickly create SPNs both from data and through a domain specific language (DSL). It efficiently implements several probabilistic inference routines like computing marginals, conditional and (approximate) most probable explanations (MPEs) along with sampling.

https://github.com/SPFlow/SPFlow

Domain Specific Language, Inference, EM, and Model Selection as well as Compilation of SPNs into TF and PyTorch and also into flat, library-free code even suitable for running on devices: C/C++,GPU, FPGA
Random sum-product networks

[Peharz, Vergari, Molina, Stelzner, Trapp, Kersting, Ghahramani UDL@UAI 2018]

Build a random SPN structure. This can be done in an informed way or completely at random.

SPNs can have similar predictive performances as (simple) DNNs.

SPNs can distinguish the datasets.

SPNs know when they do not know by design.
SPNs closely related to well known, advanced ML models, e.g. Poisson SPNs = Hierarchical Topic Models

Mutual Information
(NIPS corpus)

Poisson Multinomial SPN
= hierarchical topic model
SPNs feature distribution-agnostic deep probabilistic learning

Use nonparametric independency tests and piece-wise linear approximations of the univariate distributions in the leaves.
Putting a little bit of structure into SPN models allows one to realize autoregressive deep models akin to PixelCNNs [van den Oord et al. NIPS 2016]

Learn Conditional SPN (CSPNs) by non-parametric conditional independence testing and conditional clustering [Zhang et al. UAI 2011; Lee, Honvar UAI 2017; He et al. ICDM 2017; Zhang et al. AAAI 2018; Runge AISTATS 2018] encoded using gating functions

Conditional SPNs [Shao, Molina, Vergari, Peharz, Kersting 2019]
Learn Conditional SPN (CSPNs) by non-parametric conditional independence testing and conditional clustering [Zhang et al. UAI 2011; Lee, Honovar UAI 2017; He et al. ICDM 2017; Zhang et al. AAAI 2018; Runge AISTATS 2018] encoded using gating functions

**Conditional SPNs**
[Shao, Molina, Vergari, Peharz, Kersting 2019]
What have we learnt about SPNs?

**Sum-product networks (SPNs)**

- DAG of sums and products
- They are instances of Arithmetic Circuits (ACs)
- Compactly represent partition function
- Learn many layers of hidden variables

**Efficient marginal inference**

**Easy learning**

**Can outperform well-known alternatives e.g. faster Attend-Infer-Repeat models** [Stelzner, Peharz, Kersting ICML 2019]