tractable probabilistic modeling
with probabilistic circuits

antonio vergari (he/him)

@tetraduzione

2nd Apr 2025 - **Neuro-explicit retreat** *Saarbruecken*

april

april-tools.github.io

april

***a**bout*
***pr**obabilities*
***i**ntegrals &*
***l**ogic*

deep generative models

+

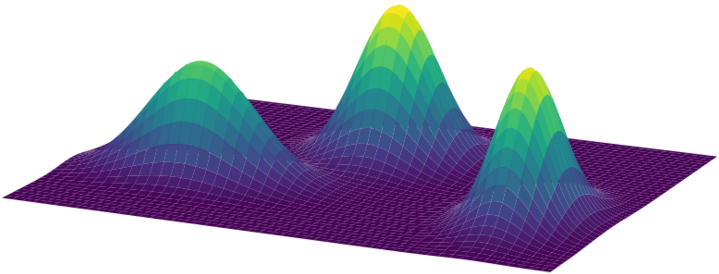
flexible and reliable

(logic &) probabilistic reasoning?

i) probabilistic circuits: syntax and semantics

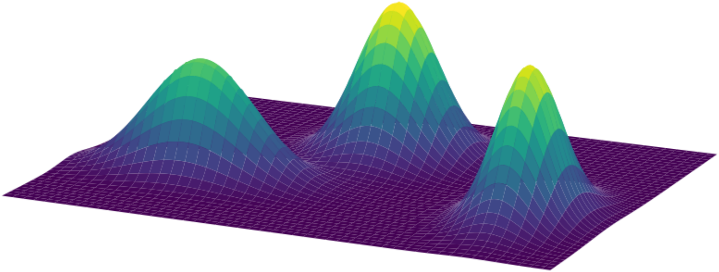
ii) reliable and efficient neuro-symbolic AI

iii) you talk: use cases from your research



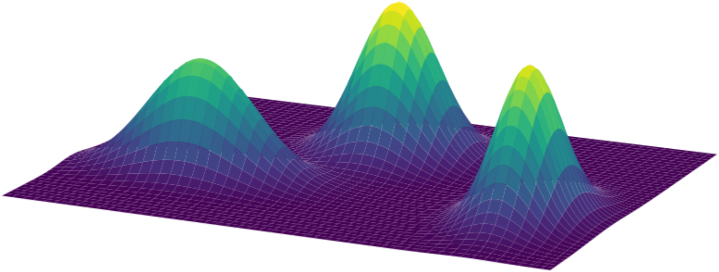
who knows mixture models?

image taken from Hao Tang's course on ASR



who loves mixture models?

image taken from Hao Tang's course on ASR

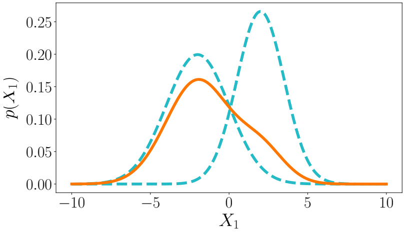


$$c(\mathbf{X}) = \sum_{i=1}^{K} w_i c_i(\mathbf{X}), \quad \text{with} \quad w_i \geq 0, \quad \sum_{i=1}^{K} w_i = 1$$

image taken from Hao Tang's course on ASR

GMMs

as computational graphs

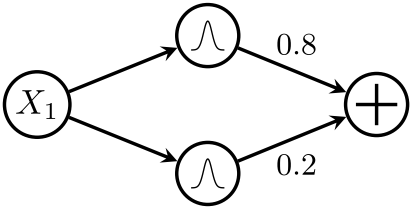


$$p(X) = w_1 \cdot p_1(X_1) + w_2 \cdot p_2(X_1)$$

$\Longrightarrow$ *translating inference to data structures...*

GMMs

as computational graphs

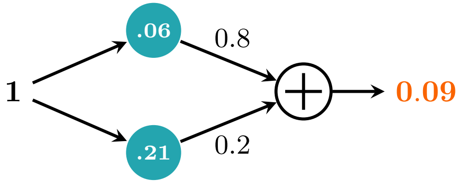


$$p(X_1) = 0.2 \cdot p_1(X_1) + 0.8 \cdot p_2(X_1)$$

$\Longrightarrow$ *...e.g., as a weighted sum unit over Gaussian input distributions*

GMMs

as computational graphs

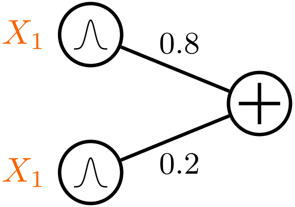


$$p(X=1) = 0.2 \cdot p_1(X_1 = 1) + 0.8 \cdot p_2(X_1 = 1)$$

$\Longrightarrow$ *inference = feedforward evaluation*

GMMs

as computational graphs



A simplified notation:

$\Rightarrow$ ***scopes*** *attached to inputs*

$\Rightarrow$ *edge directions omitted*

GMMs

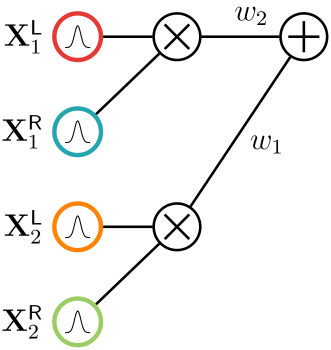
as computational graphs

$$p(\mathbf{X}) = w_1 \cdot p_1(\mathbf{X}_1^{\mathsf{L}}) \cdot p_1(\mathbf{X}_1^{\mathsf{R}}) + w_2 \cdot p_2(\mathbf{X}_2^{\mathsf{L}}) \cdot p_2(\mathbf{X}_2^{\mathsf{R}})$$

$\Longrightarrow$ *local factorizations...*

GMMs

as computational graphs



$$p(\mathbf{X}) = w_1 \cdot p_1(\mathbf{X}_1^{\mathsf{L}}) \cdot p_1(\mathbf{X}_1^{\mathsf{R}}) + w_2 \cdot p_2(\mathbf{X}_2^{\mathsf{L}}) \cdot p_2(\mathbf{X}_2^{\mathsf{R}})$$

$\Rightarrow$ *...are product units*

probabilistic circuits (PCs)

a grammar for tractable computational graphs

I. *A simple tractable function is a circuit*

$\Rightarrow$ *e.g., a multivariate Gaussian or orthonormal polynomial*

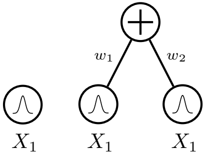
X_1

probabilistic circuits (PCs)

a grammar for tractable computational graphs

I. *A simple tractable function is a circuit*

II. *A weighted combination of circuits is a circuit*



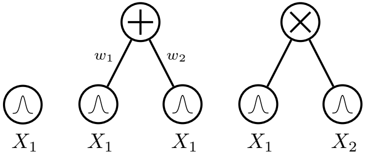
probabilistic circuits (PCs)

a grammar for tractable computational graphs

I. *A simple tractable function is a circuit*

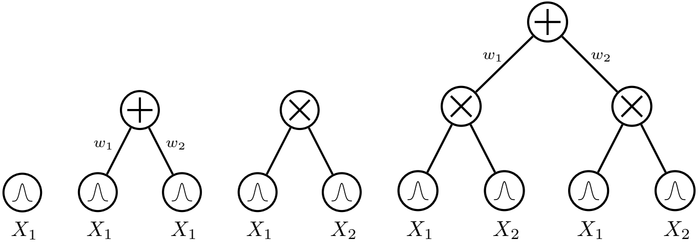
II. *A weighted combination of circuits is a circuit*

III. *A product of circuits is a circuit*



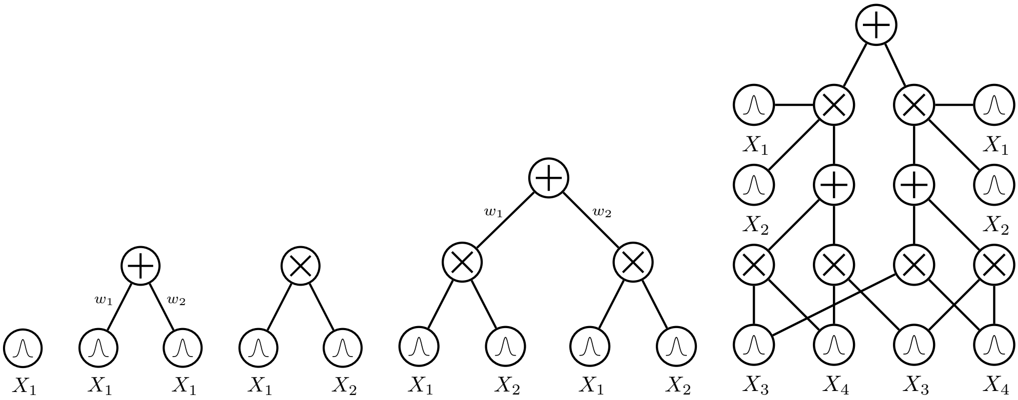
probabilistic circuits (PCs)

a grammar for tractable computational graphs



probabilistic circuits (PCs)

a grammar for tractable computational graphs



probabilistic circuits (PCs)

a tensorized definition

I. *A set of tractable functions is a circuit layer*

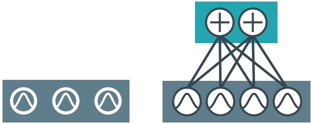
probabilistic circuits (PCs)

a tensorized definition

I. *A set of tractable functions is a circuit layer*

II. *A linear projection of a layer is a circuit layer*

$$c(\mathbf{x}) = \mathbf{W}\boldsymbol{l}(\mathbf{x})$$



probabilistic circuits (PCs)

a tensorized definition

I. *A set of tractable functions is a circuit layer*

II. *A linear projection of a layer is a circuit layer*

$$c(\mathbf{x}) = \mathbf{W}\boldsymbol{l}(\mathbf{x})$$

probabilistic circuits (PCs)

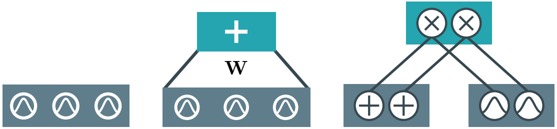
a tensorized definition

I. *A set of tractable functions is a circuit layer*

II. *A linear projection of a layer is a circuit layer*

III. *The product of two layers is a circuit layer*

$$c(\mathbf{x}) = \boldsymbol{l}(\mathbf{x}) \odot \boldsymbol{r}(\mathbf{x}) \qquad \text{// Hadamard}$$



probabilistic circuits (PCs)

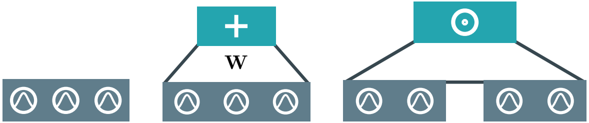
a tensorized definition

I. *A set of tractable functions is a circuit layer*

II. *A linear projection of a layer is a circuit layer*

III. *The product of two layers is a circuit layer*

$$c(\mathbf{x}) = \boldsymbol{l}(\mathbf{x}) \odot \boldsymbol{r}(\mathbf{x}) \qquad \text{// Hadamard}$$



probabilistic circuits (PCs)

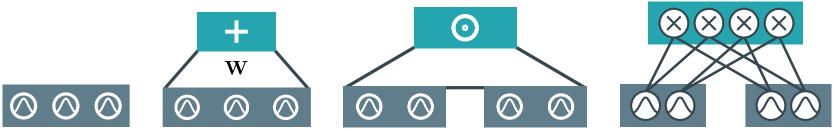
a tensorized definition

I. *A set of tractable functions is a circuit layer*

II. *A linear projection of a layer is a circuit layer*

III. *The product of two layers is a circuit layer*

$$c(\mathbf{x}) = \mathsf{vec}(\boldsymbol{l}(\mathbf{x})\boldsymbol{r}(\mathbf{x})^\top) \qquad \text{// Kronecker}$$



probabilistic circuits (PCs)

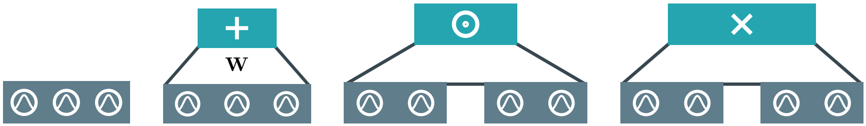
a tensorized definition

I. *A set of tractable functions is a circuit layer*

II. *A linear projection of a layer is a circuit layer*

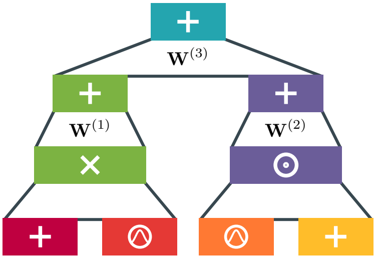
III. *The product of two layers is a circuit layer*

$$c(\mathbf{x}) = \text{vec}(\boldsymbol{l}(\mathbf{x})\boldsymbol{r}(\mathbf{x})^\top) \qquad \text{// Kronecker}$$



probabilistic circuits (PCs)

a tensorized definition



I. *A set of tractable functions is a circuit layer*

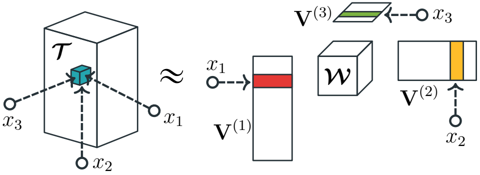
II. *A linear projection of a layer is a circuit layer*

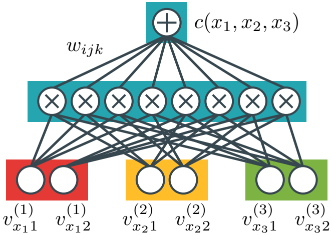
III. *The product of two layers is a circuit layer*

stack layers to build a deep circuit!

tensor factorizations

as circuits

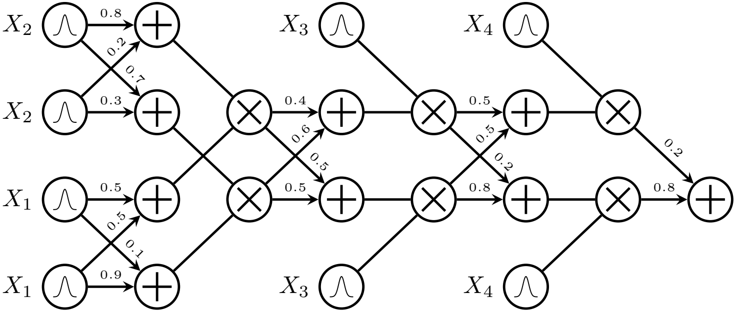




Loconte et al., "What is the Relationship between Tensor Factorizations and Circuits (and How Can We Exploit it)?", TMLR, 2025

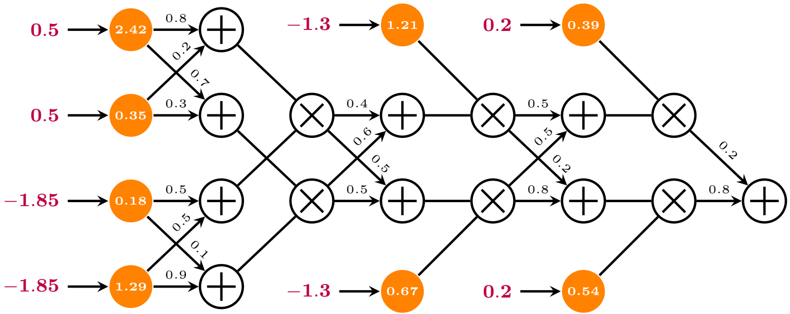
Probabilistic queries = feedforward evaluation

$$p(X_1 = \mathbf{-1.85}, X_2 = \mathbf{0.5}, X_3 = \mathbf{-1.3}, X_4 = \mathbf{0.2})$$



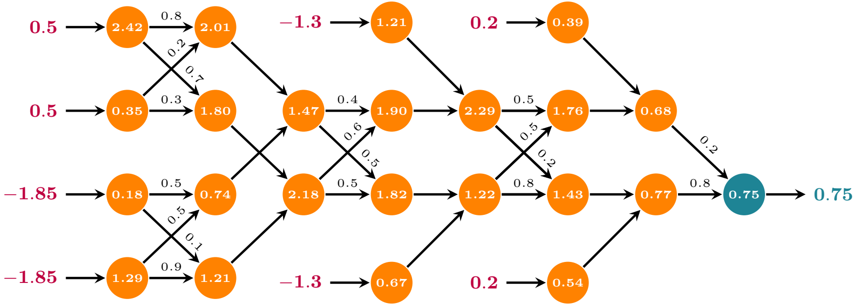
Probabilistic queries = feedforward evaluation

$$p(X_1 = \mathbf{-1.85}, X_2 = \mathbf{0.5}, X_3 = \mathbf{-1.3}, X_4 = \mathbf{0.2})$$



Probabilistic queries = feedforward evaluation

$$p(X_1 = -\mathbf{1.85}, X_2 = \mathbf{0.5}, X_3 = -\mathbf{1.3}, X_4 = \mathbf{0.2}) = 0.75$$



cirkit

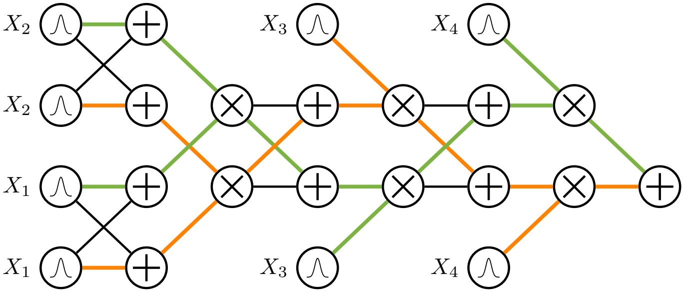
learning & reasoning with circuits in pytorch

`github.com/april-tools/cirkit`

```python
from cirkit.templates import circuit_templates

symbolic_circuit = circuit_templates.image_data(
(1, 28, 28),                    # The shape of MNIST
    region_graph='quad-graph',
    input_layer='categorical',  # input distributions
    sum_product_layer='cp',     # CP, Tucker, CP-T
    num_input_units=64,         # overparameterizing
    num_sum_units=64,
    sum_weight_param=circuit_templates.Parameterization(
        activation='softmax',
        initialization='normal'
    )
)
```

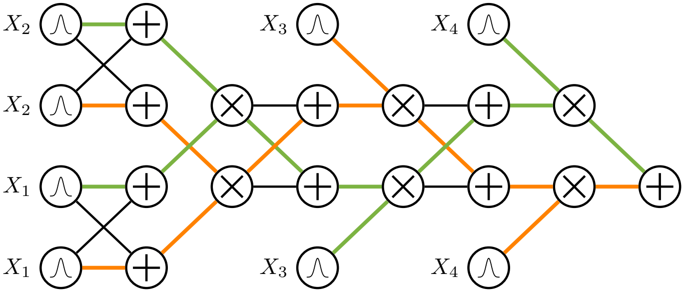
```python
from cirkit.pipeline import compile
circuit = compile(symbolic_circuit)

with torch.no_grad():
    test_lls = 0.0
    for batch, _ in test_dataloader:
        batch = batch.to(device).unsqueeze(dim=1)
        log_likelihoods = circuit(batch)
        test_lls += log_likelihoods.sum().item()
    average_ll = test_lls / len(data_test)
    bpd = -average_ll / (28 * 28 * np.log(2.0))
    print(f"Average LL: {average_ll:.3f}") # Average LL:
    ↪  -682.916
    print(f"Bits per dim: {bpd:.3f}") # Bits per dim: 1.257
```

deep mixtures



$$p(\mathbf{x}) = \sum_{\mathcal{T}} \left(\prod_{w_j \in \mathbf{w}_{\mathcal{T}}} w_j \right) \prod_{l \in \text{leaves}(\mathcal{T})} p_l(\mathbf{x})$$

deep mixtures



an exponential number of mixture components!

...why PCs?

1. A grammar for tractable models

One formalism to represent many probabilistic models

$\Longrightarrow$ *#HMMs #Trees #XGBoost, Tensor Networks, ...*

...why PCs?

1. A grammar for tractable models

One formalism to represent many probabilistic models

$\Longrightarrow$ *#HMMs #Trees #XGBoost, Tensor Networks, ...*

2. Tractability == structural properties!!!

Exact computations of reasoning tasks are certified by guaranteeing certain structural properties. *#marginals #expectations #MAP, #product ...*

structural properties

smoothness

decomposability

compatibility

structural properties

property A

property B

property C

structural properties

smoothness

decomposability

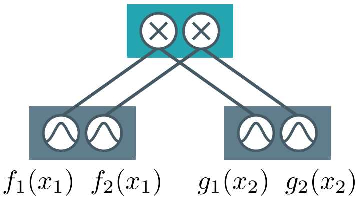
property C

smoothness $\wedge$ decomposability

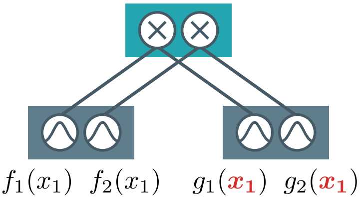
$\Longrightarrow$ multilinearity

Multilinearity in circuits

the inputs of product units are defined over disjoint sets of variables



✓ **multilinear**

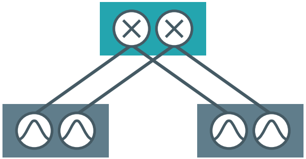


✗ **not multilinear**

Darwiche and Marquis, "A knowledge compilation map", , 2002

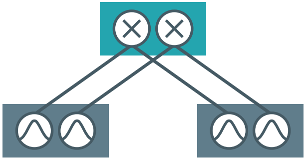
Multilinearity in circuits

the inputs of product units are defined over disjoint sets of variables



$f_1(x_1)$ $f_2(x_1)$ $g_1(x_2)$ $g_2(x_2)$

decomposable circuit



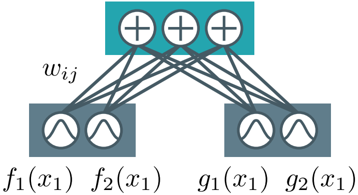
$f_1(x_1)$ $f_2(x_1)$ $g_1(\boldsymbol{x_1})$ $g_2(\boldsymbol{x_1})$

non-decomposable circuit

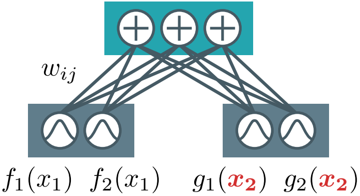
Darwiche and Marquis, "A knowledge compilation map", , 2002

Multilinearity in circuits

the inputs of sum units are defined over the same variables



✓ **multilinear**

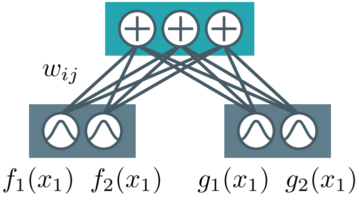


✗ **not multilinear**

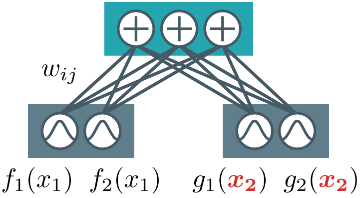
Darwiche and Marquis, "A knowledge compilation map", , 2002

Multilinearity in circuits

the inputs of sum units are defined over the same variables



smooth circuit



non-smooth circuit

Darwiche and Marquis, "A knowledge compilation map", , 2002

structural properties

smoothness

decomposability

property C

smoothness $\wedge$ decomposability

$\Longrightarrow$ multilinearity

structural properties

smoothness

decomposability

property C

tractable computation of **arbitrary integrals** in probabilistic circuits

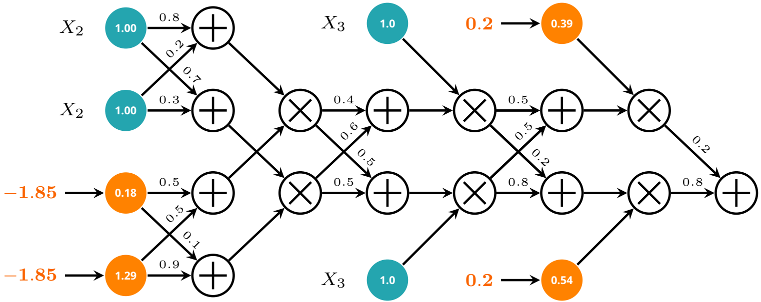
$$p(\mathbf{y}) = \int p(\mathbf{y}, \mathbf{z})\, \mathrm{d}\mathbf{z}, \quad \forall \mathbf{Y} \subseteq \mathbf{X}, \quad \mathbf{Z} = \mathbf{X} \setminus \mathbf{Y}$$

$\Longrightarrow$ tractable partition function

$\Longrightarrow$ also any conditional is tractable

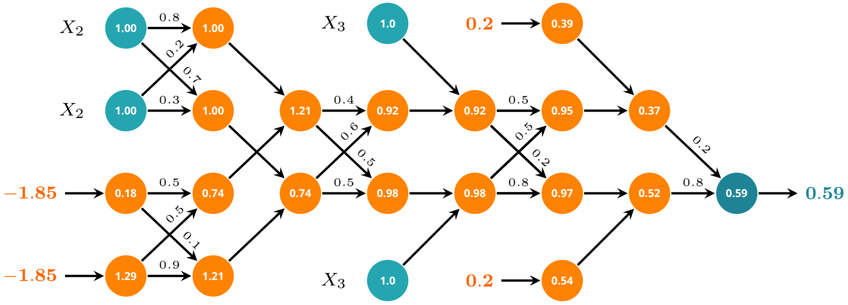
Probabilistic queries = feedforward evaluation

$$p(X_1 = -1.85, X_4 = 0.2)$$



Probabilistic queries = feedforward evaluation

$$p(X_1 = -1.85, X_4 = 0.2)$$



smooth + ***decomposable*** circuits = ...

Computing arbitrary integrations (or summations)

$\Longrightarrow$ *linear in circuit size!*

E.g., suppose we want to compute Z:

$$\int \boldsymbol{p}(\mathbf{x})d\mathbf{x}$$

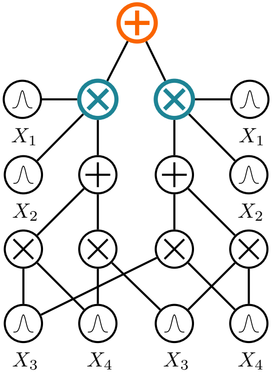
smooth + **decomposable** circuits = ...

If $\boldsymbol{p}(\mathbf{x}) = \sum_i w_i \boldsymbol{p}_i(\mathbf{x})$, (***smoothness***):

$$\int \boldsymbol{p}(\mathbf{x}) d\mathbf{x} = \int \sum_i w_i \boldsymbol{p}_i(\mathbf{x}) d\mathbf{x} =$$

$$= \sum_i w_i \int \boldsymbol{p}_i(\mathbf{x}) d\mathbf{x}$$

$\Rightarrow$ *integrals are "pushed down" to inputs*



smooth + decomposable circuits = ...

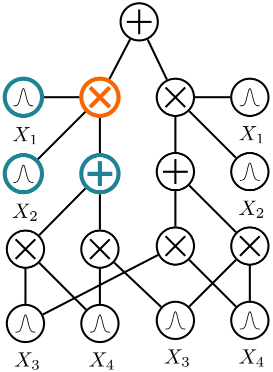
If $p(\mathbf{x}, \mathbf{y}, \mathbf{z}) = p(\mathbf{x})p(\mathbf{y})p(\mathbf{z})$, (***decomposability***):

$$\int\int\int p(\mathbf{x}, \mathbf{y}, \mathbf{z})d\mathbf{x}d\mathbf{y}d\mathbf{z} =$$

$$= \int\int\int p(\mathbf{x})p(\mathbf{y})p(\mathbf{z})d\mathbf{x}d\mathbf{y}d\mathbf{z} =$$

$$= \int p(\mathbf{x})d\mathbf{x} \int p(\mathbf{y})d\mathbf{y} \int p(\mathbf{z})d\mathbf{z}$$

$\Longrightarrow$ *integrals decompose into easier ones*



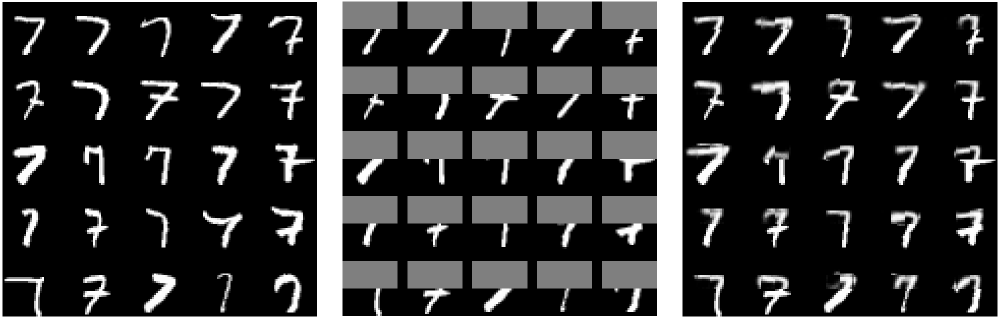
```python
from cirkit.backend.torch.queries import IntegrateQuery
marginal_query = IntegrateQuery(circuit)

with torch.no_grad():
    test_marginal_lls = 0.0

    for batch, _ in test_dataloader:
        batch = batch.to(device).unsqueeze(dim=1)
        marginal_log_likelihoods = marginal_query(batch,
        ↪  integrate_vars=vars_to_marginalize)
        test_marginal_lls +=
        ↪  marginal_log_likelihoods.sum().item()

    marg_ll = test_marginal_lls / len(data_test)
    print(f"marg LL: {marg_ll:.3f}") # marg LL:: -378.417
```

tractable marginals on PCs

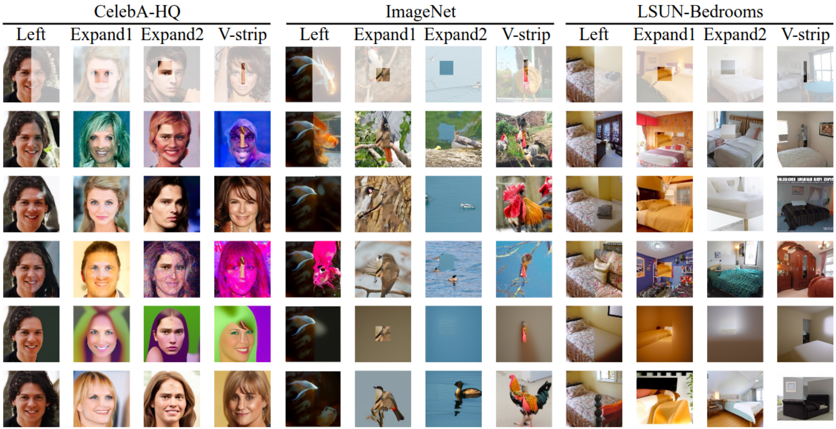
Original

Missing

Conditional sample



Peharz et al., "Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits", , 2020



Liu, Niepert, and Broeck, “Image Inpainting via Tractable Steering of Diffusion Models”, The Twelfth International Conference on Learning Representations (ICLR), 2024

structural properties

smoothness

decomposability

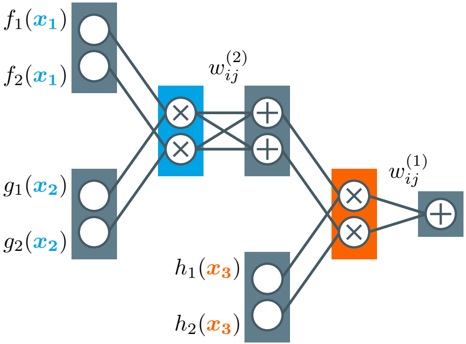
compatibility

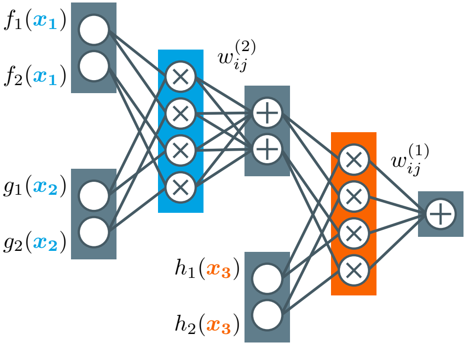
Integrals involving two or more functions:
e.g., expectations

$$\mathbb{E}_{\mathbf{x}\sim p}\, f(\mathbf{x}) = \int p(\mathbf{x})\, f(\mathbf{x})\, \mathrm{d}\mathbf{x}$$

when both $p(\mathbf{x})$ and $f(\mathbf{x})$ are circuits

compatibility

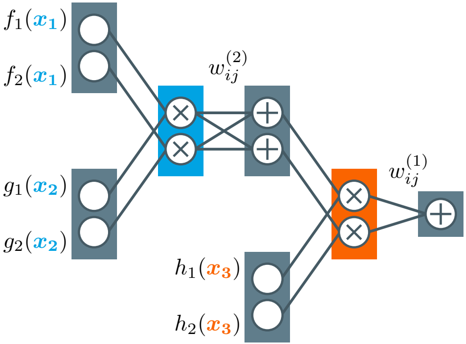


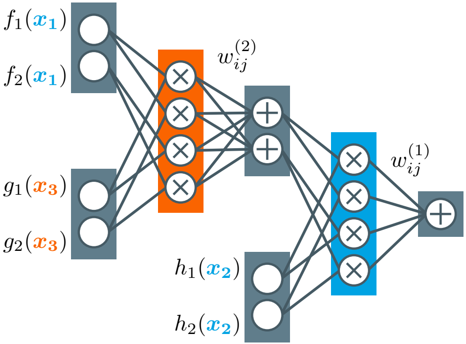


compatibile circuits

Darwiche and Marquis, "A knowledge compilation map", , 2002

compatibility





non-compatibile circuits

Darwiche and Marquis, "A knowledge compilation map", , 2002

structural properties

smoothness

decomposability

compatibility

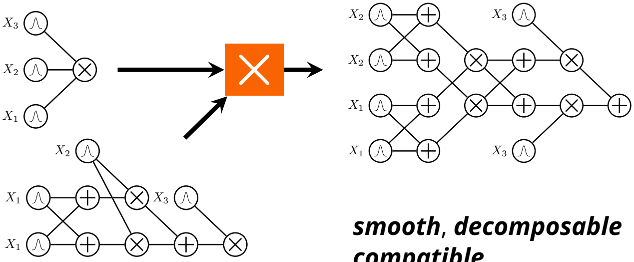
compatibility

$$\Downarrow$$

smoothness $\wedge$ **decomposability**

compatiblity $\Longrightarrow$ tractable expectations

Tractable products



smooth, ***decomposable***
compatible

compute $\mathbb{E}_{\mathbf{x}\sim p}\, f(\mathbf{x}) = \int p(\mathbf{x})\, f(\mathbf{x})\, \mathrm{d}\mathbf{x}$ **in** $O(|p||f|)$

Vergari et al., "A Compositional Atlas of Tractable Circuit Operations for Probabilistic Inference", NeurIPS, 2021

```python
from cirkit.symbolic.circuit import Circuit
from cirkit.symbolic.functional import (
    integrate, multiply)

# Circuits expectation \int [p(x) f(x)]dx
def expectation(p: Circuit, f: Circuit) -> Circuit:
    i = multiply(p, f)
    return integrate(i)

# Squared loss \int [p(x)-q(x)]^2dx = E_p[p] + E_q[q] - 2E_p[q]
#              = \int p^2(x)dx + \int q^2(x)dx - 2\int p(x)q(x)dx
def squared_loss(p: Circuit, q: Circuit) -> Circuit:
    p2 = multiply(p, p)
    q2 = multiply(q, q)
    pq = multiply(p, q)
    return integrate(p2) + integrate(q2) - 2 * integrate(pq)
```

...why PCs?

1. A grammar for tractable models

One formalism to represent many probabilistic models

$\Longrightarrow$ *#HMMs #Trees #XGBoost, Tensor Networks, ...*

2. Tractability == structural properties!!!

Exact computations of reasoning tasks are certified by guaranteeing certain structural properties. *#marginals #expectations #MAP, #product* ...

...why PCs?

1. A grammar for tractable models

One formalism to represent many probabilistic models

$\Longrightarrow$ *#HMMs #Trees #XGBoost, Tensor Networks, ...*

2. Tractability == structural properties!!!

Exact computations of reasoning tasks are certified by guaranteeing certain structural properties. *#marginals #expectations #MAP, #product ...*

3. Realiable neuro-symbolic AI

logical constraints as circuits, multiplied to probabilistic circuits

```python
from cirkit.templates import circuit_templates

symbolic_circuit = circuit_templates.image_data(
(1, 28, 28),                  # The shape of MNIST
    region_graph='quad-graph',
    input_layer='categorical',  # input distributions
    sum_product_layer='cp',     # CP, Tucker, CP-T
    num_input_units=64,         # overparameterizing
    num_sum_units=64,
    sum_weight_param=circuit_templates.Parameterization(
        activation='softmax',
        initialization='normal'
    )
)
```

learning probabilistic circuits

learning probabilistic circuits

Probabilistic circuits are (peculiar) neural networks...***just backprop with SGD!***

learning probabilistic circuits

Probabilistic circuits are (peculiar) neural networks...***just backprop with SGD!***

...end of Learning section!

learning probabilistic circuits

Probabilistic circuits are (peculiar) neural networks...***just backprop with SGD!***

wait but...

which loss?

how to learn normalized weights?

how to exploit structural properties?

maximum likelihood

the go-to objective in ProbML

Given a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$ and your parametric model $p_\theta(\mathbf{X})$ solve

$$\hat{\theta}_{\mathsf{ML}} = \max_\theta \prod_{i=1}^{N} p_\theta(\mathbf{x}^{(i)}) = \min_\theta - \sum_{i=1}^{N} \log p_\theta(\mathbf{x}^{(i)})$$

$\Longrightarrow$ *minimize the negative log-likelihood (NLL)*

```python
from cirkit.templates import circuit_templates

symbolic_circuit = circuit_templates.image_data(
(1, 28, 28),                    # The shape of MNIST
    region_graph='quad-graph',
    input_layer='categorical',  # input distributions
    sum_product_layer='cp',     # CP, Tucker, CP-T
    num_input_units=64,         # overparameterizing
    num_sum_units=64,
    sum_weight_param=circuit_templates.Parameterization(
        activation='softmax',
        initialization='normal'
    )
)
```

which parameters?

how to reparameterize circuits

Input distributions.

Sum unit parameters.

which parameters?

how to reparameterize circuits

Input distributions. Each input can be a different parametric distribution

$\Longrightarrow$ *Bernoullis, Categoricals, Gaussians, **exponential families**, small NNs, ...*

Sum unit parameters.

which parameters?

how to reparameterize circuits

Input distributions. Each input can be a different parametric distribution

Sum unit parameters. Enforce them to be non-negative, i.e., $w_i \geq 0$ but unnormalized

$$w_i = \exp(\alpha_i), \quad \alpha_i \in \mathbb{R}, \quad i = 1, \ldots, K$$

and renormalize the loss

$$\min_{\theta} - \left(\sum_{i=1}^{N} \log \tilde{p}_{\theta}(\mathbf{x}^{(i)}) - \log \int \tilde{p}_{\theta}(\mathbf{x}^{(i)})\, d\mathbf{X} \right)$$

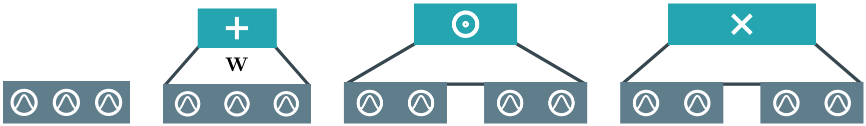
or just renormalize the weights, i.e., $\sum_i w_i = 1$

$$\mathbf{w} = \mathsf{softmax}(\boldsymbol{\alpha}), \quad \boldsymbol{\alpha} \in \mathbb{R}^K$$

```python
from cirkit.templates import circuit_templates

symbolic_circuit = circuit_templates.image_data(
(1, 28, 28),                    # The shape of MNIST
    region_graph='quad-graph',
    input_layer='categorical',  # input distributions
    sum_product_layer='cp',     # CP, Tucker, CP-T
    num_input_units=64,         # overparameterizing
    num_sum_units=64,
    sum_weight_param=circuit_templates.Parameterization(
        activation='softmax',
        initialization='normal'
    )
)
```

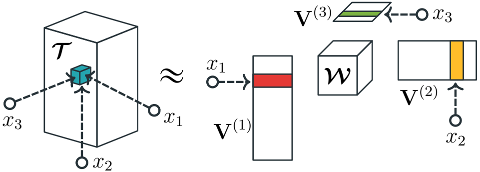
Probabilistic Circuits (PCs)

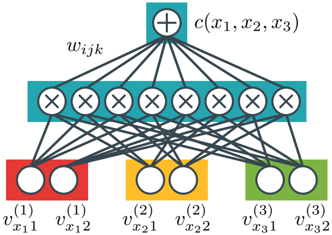
the layer-wise definition



circuits layers

as tensor factorizations

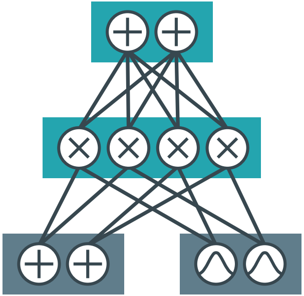


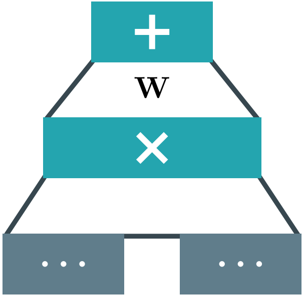


loconte2024relationship, loconte2024relationship, loconte2024relationship, loconte2024relationship

more layers

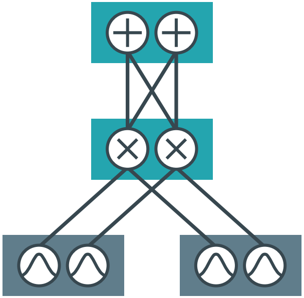
Tucker decomposition



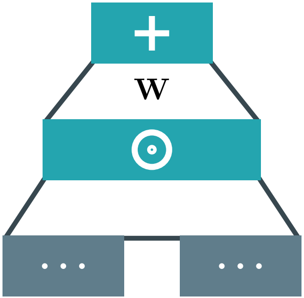


more layers

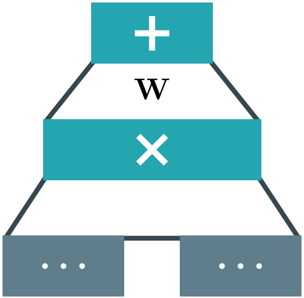
Candecomp Parafac (CP) decomposition



more layers



CP layer

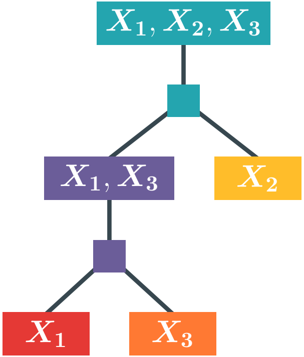


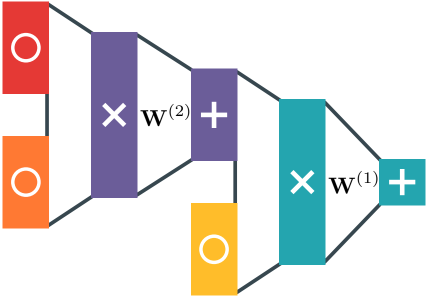
Tucker layer

```python
from cirkit.templates import circuit_templates

symbolic_circuit = circuit_templates.image_data(
(1, 28, 28),                    # The shape of MNIST
    region_graph='quad-graph',
    input_layer='categorical',  # input distributions
    sum_product_layer='cp',     # CP, Tucker, CP-T
    num_input_units=64,         # overparameterizing
    num_sum_units=64,
    sum_weight_param=circuit_templates.Parameterization(
        activation='softmax',
        initialization='normal'
    )
)
```

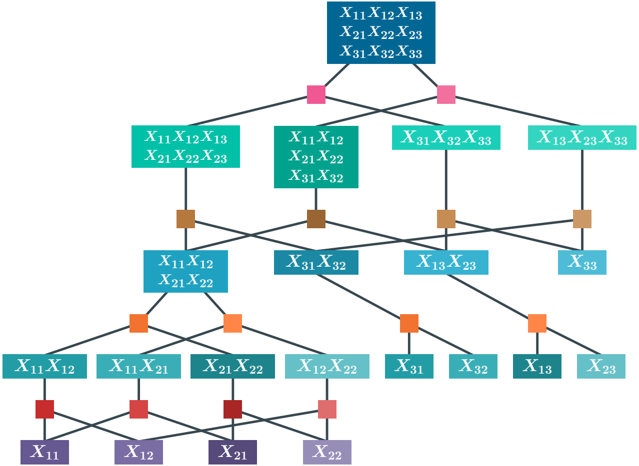
region graphs

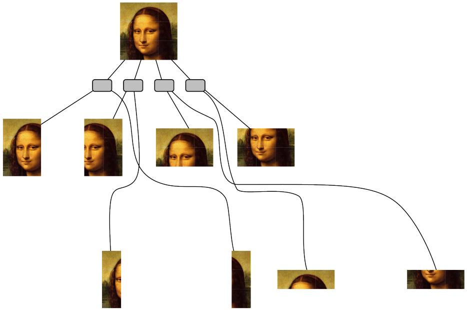
a template for smooth&decomposable PCs

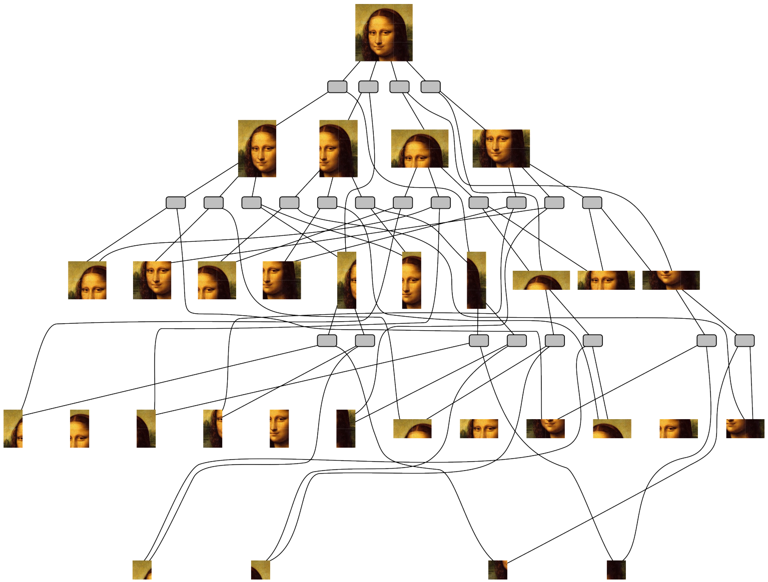


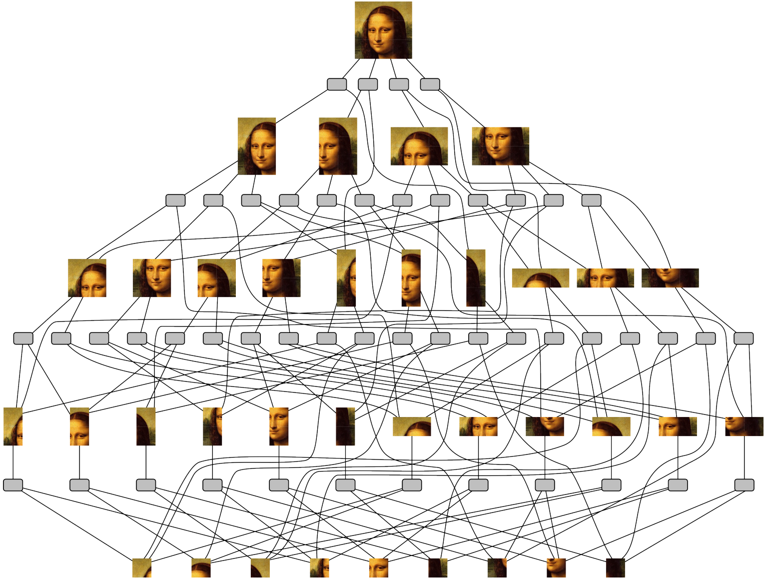


which region graph?





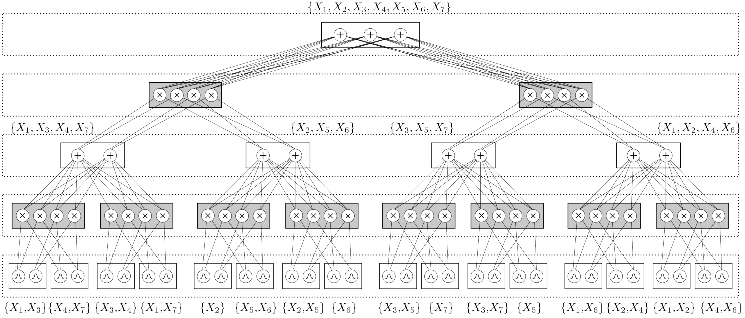




random regions graphs

The "no-learning" option

Generating a random region graph, by recursively splitting $\mathbf{X}$ into two random parts:



main

cirkit / notebooks / region-graphs-and-parametrisation.ipynb

Go to file

loreloc updated notebooks with respect to API changes

e3e7e80 · 2 days ago

Preview | Code | Blame | 1082 lines (1082 loc) · 793 KB | Raw

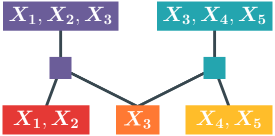
Notebook on Region Graphs and Sum Product Layers

Goals

By the end of this tutorial you will:

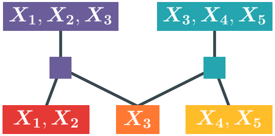
- know what a region graph is
- know how to choose between region graphs for your circuit
- understand how to parametrize a circuit by choosing a sum product layer
- build circuits to **tractably** estimate a probability distribution over images[1]

learning recipe

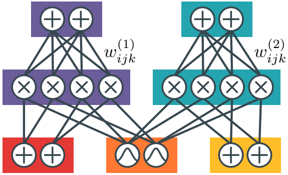


1) Build a ***region graph***

learning recipe



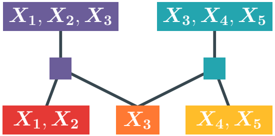
1) Build a ***region graph***



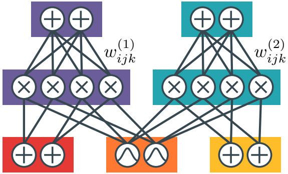
2) Overparameterize

2.1) pick a (composite) layer type
2.2) choose how many units per layer

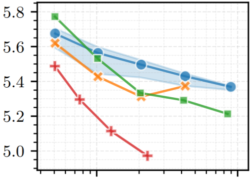
learning recipe



1) Build a ***region graph***



2) Overparameterize



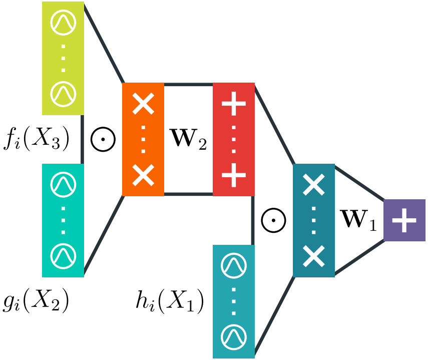
3) Learn parameters

use any optimizer in pytorch

cirkit

learning & reasoning with circuits in pytorch

https://github.com/april-tools/cirkit



questions?