DeepStochLog:
Neural Stochastic Logic Programming

Thomas Winters*[1], Giuseppe Marra*[1]
Robin Manhaeve[1], Luc De Raedt[1,2]

[1]KU Leuven, Belgium
[2]AASS, Örebro University, Sweden

* shared first author

DeepStochLog =

- ★ **Neural-Symbolic Framework**
- ★ Introduces “**Neural Definite Clause Grammars**”
- ★ **SOTA results** on wide variety of tasks
- ★ **Scales better** than others

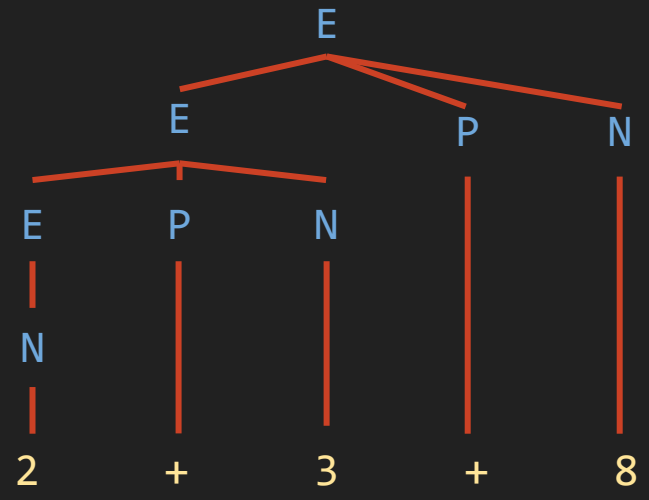
Neural Definite Clause Grammar

CFG: Context-Free Grammar

```
E --> N
E --> E, P, N

P --> [“+”]

N --> [“0”]
N --> [“1”]
…
N --> [“9”]
```



Useful for:

- Is sequence an element of the specified language?
- What is the *“part of speech”*-tag of a terminal
- Generate all elements of language

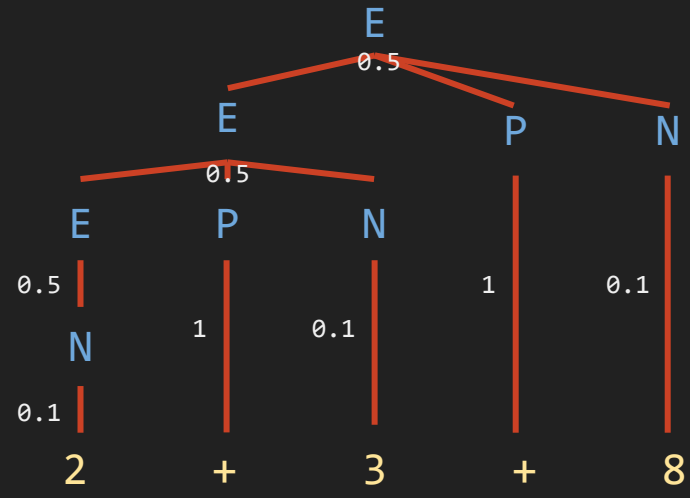
PCFG: Probabilistic Context-Free Grammar

```
0.5 :: E --> N
0.5 :: E --> E, P, N

1.0 :: P --> [“+”]

0.1 :: N --> [“0”]
0.1 :: N --> [“1”]
       …
0.1 :: N --> [“9”]
```

Always sums to 1 per non-terminal



*Probability of this parse = 0.5*0.5*0.5*0.1*1*0.1*1*0.1 = 0.000125*

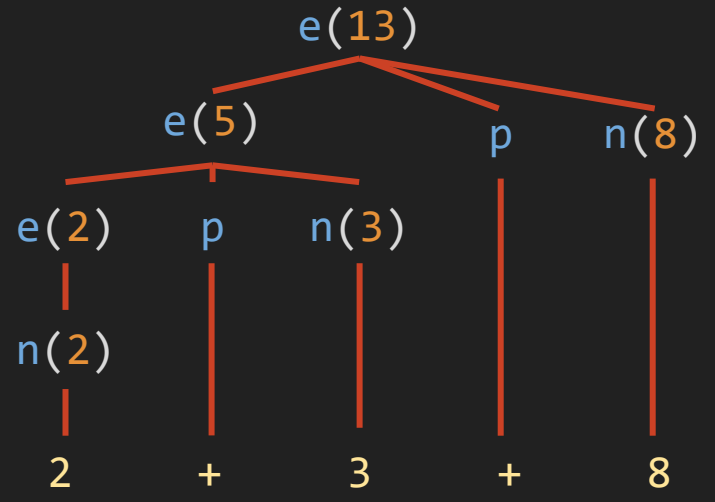
Useful for:

- What is the most likely parse for this sequence of terminals? *(useful for ambiguous grammars)*
- What is the probability of generating this string?

DCG: Definite Clause Grammar

```
e(N) --> n(N).
e(N) --> e(N1), p, n(N2),
         {N is N1 + N2}.
p    --> ["+"].

n(0) --> ["0"].
n(1) --> ["1"].
…
n(9) --> ["9"].
```



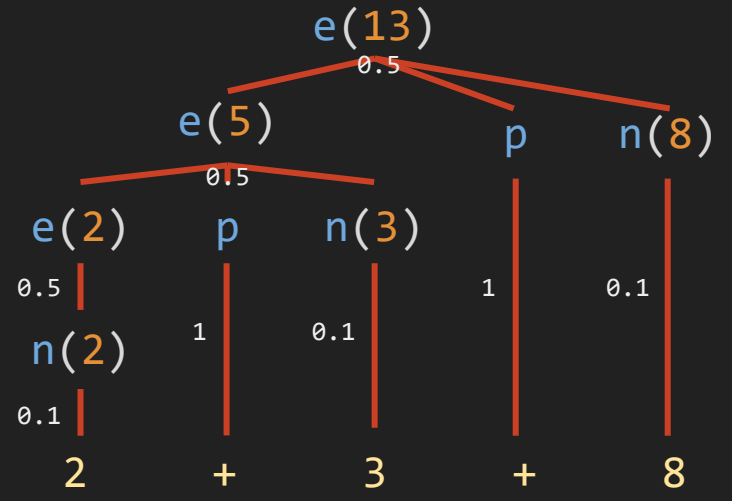
Useful for:

- Modelling more complex languages *(e.g. context-sensitive)*
- Adding constraints between non-terminals thanks to Prolog power *(e.g. through unification)*
- Extra inputs & outputs aside from terminal sequence *(through unification of input variables)*

SDCG: Stochastic Definite Clause Grammar

```
0.5 :: e(N) --> n(N).
0.5 :: e(N) --> e(N1), p, n(N2),
                {N is N1 + N2}.
1.0 :: p    --> ["+"].

0.1 :: n(0) --> ["0"].
0.1 :: n(1) --> ["1"].
       …
0.1 :: n(9) --> ["9"].
```



*Probability of this parse = 0.5*0.5*0.5*0.1*1*0.1*1*0.1 = 0.000125*

Useful for:

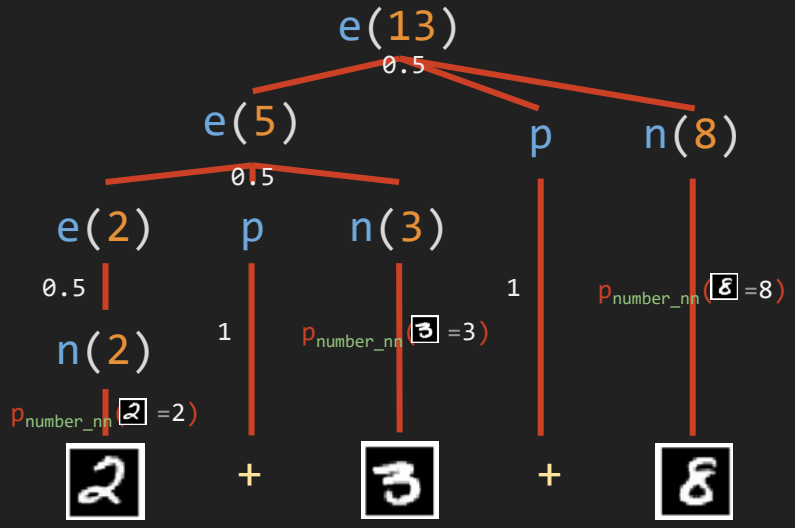
- Same benefits as PCFGs give to CFG *(e.g. most likely parse)*
- But: loss of probability mass possible due to failing derivations

NDCG: Neural Definite Clause Grammar (= DeepStochLog)

```
0.5 :: e(N) --> n(N).
0.5 :: e(N) --> e(N1), p, n(N2),
                {N is N1 + N2}.
1.0 :: p    --> ["+"].

nn(number_nn,[X],[Y],[digit]) :: n(Y) --> [X].

digit(Y) :- member(Y,[0,1,2,3,4,5,6,7,8,9]).
```



Probability of this parse =

$0.5*0.5*0.5*p_{number_nn}(\ =2)*1*p_{number_nn}(\ =3)*1*p_{number_nn}(\ =8)$

Useful for:

- Subsymbolic processing: e.g. tensors as terminals
- Learning rule probabilities using neural networks

DeepStochLog NDCG definition

$$nn(m,[I_1,\ldots,I_m],[O_1,\ldots,O_L],[D_1,\ldots,D_L]) :: nt \dashrightarrow g_1, \ldots, g_n.$$

Where:

- nt is an atom
- $g_1, \ldots, g_n$ are goals *(goal = atom or list of terminals & variables)*
- $I_1,\ldots,I_m$ and $O_1,\ldots,O_L$ are variables occurring in $g_1, \ldots, g_n$ and are the inputs and outputs of m
- $D_1,\ldots,D_L$ are the predicates specifying the domains of $O_1,\ldots,O_L$
- m is a neural network mapping $I_1,\ldots,I_m$ to probability distribution over $O_1,\ldots,O_L$ *(= over cross product of $D_1,\ldots,D_L$)*

DeepStochLog Inference

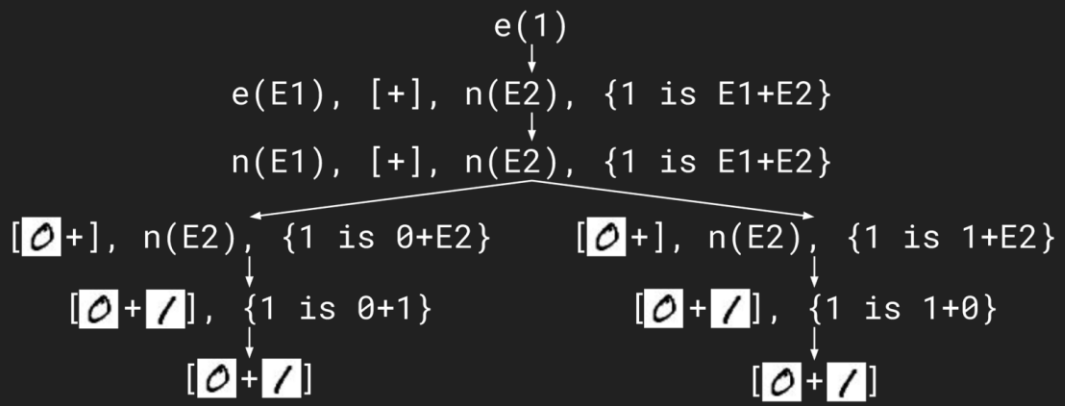
Derivation of SDCG

Example 2 (Derivations)**.** Consider the following successful derivation using the SDCG in Example 1 for the goal $G = [e(X)]$, the answer substitution $\theta = \{X/2\}$ and the terminal sequence $T = [2, +, 0]$.

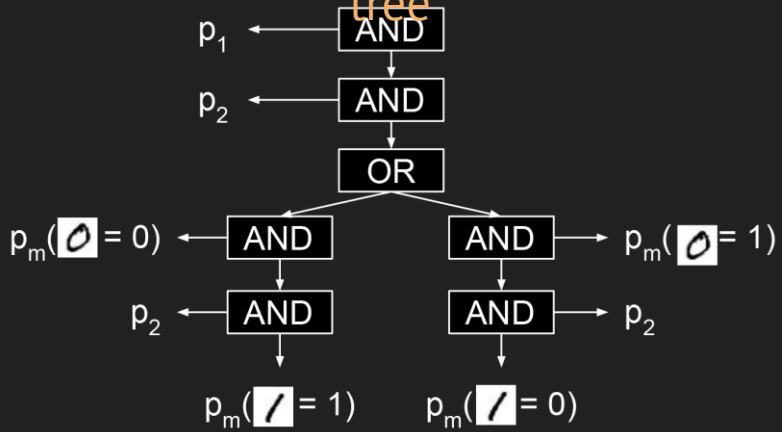
$$
\begin{array}{lrr}
e(X) \vdash e(N1), [+], n(N2), \{X \text{ is } N1 + N2\} & \theta_1 = \{\} & p = 0.5 \\
\vdash n(N1), [+], n(N2), \{X \text{ is } N1 + N2\} & \theta_2 = \{\} & \times 0.5 \\
\vdash [2, +], n(N2), \{X \text{ is } 2 + N2\} & \theta_3 = \{N1/2\} & \times 0.1 \\
\vdash [2, +, 0], 2 \text{ is } 2 + 0 & \theta_4 = \{X/2, N2/0\} & \times 0.1
\end{array}
$$

Deriving probability of goal for given terminals in NDCG

Proof derivations d(e(1),[0+1])



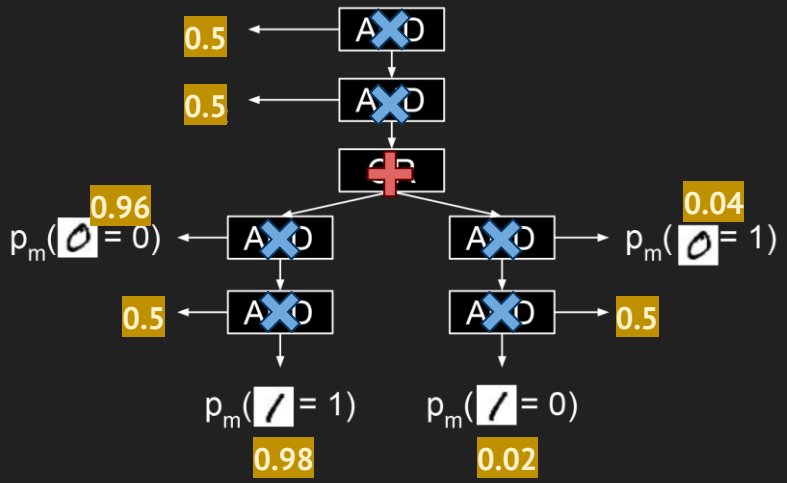
then turn it into and/or tree



And/Or tree + semiring for different inference types

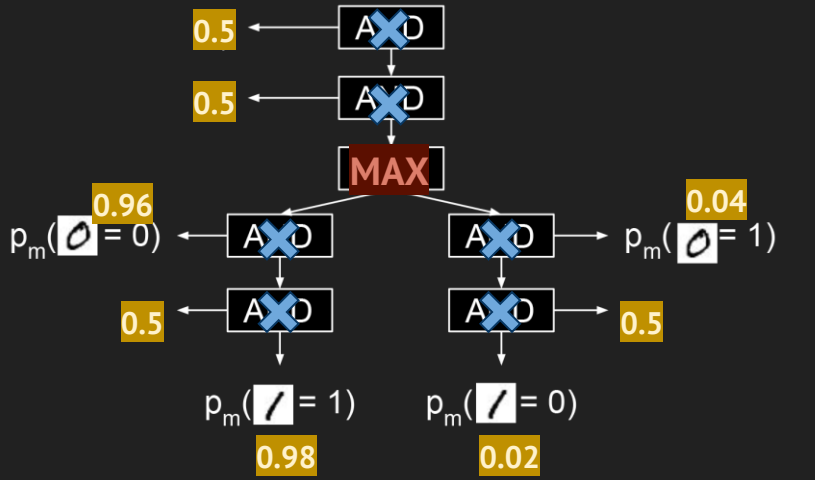
Probability of goal

$P_G(\text{derives}(e(1), [0, +, 1])) = 0.1141$



Most likely derivation

$d_{max}(e(1), [0, +, 1]) = \text{argmax}_{d(e(t))=[0,+,1]} P_G(d(e(1))) = [0,+,1]$



Inference optimisation

Inference is optimized using

1. **SLG resolution**: Prolog tables the returned proof tree(s), and thus creates forest

   → Allows for reusing probability calculation results from intermediate nodes

Table 6: **Q4** Parsing time in seconds (**T2**). Comparison of the DeepStochLog with and without tabling (SLD vs SLG resolution).

Lengths	# Answers	No Tabling	Tabling
1	10	0.067	0.060
3	95	0.081	0.096
5	1066	3.78	0.95
7	10386	30.42	10.95
9	68298	1494.23	132.26
11	416517	timeout	1996.09

1. **Batched network calls**: Evaluate all the required neural network queries first

   → Very natural for neural networks to evaluate multiple instances at once using batching & less overhead in logic & neural network communication

Learning in DeepStochLog

Learning in DeepStochLog

- All AND/OR tree leafs are (neural) probabilities
  - → DeepStochLog “links” these predictions
- Minimize the loss w.r.t. rule probabilities *p*:

$$\min_{p} \sum_{(G_i\theta_i, T_i, t_i) \in \mathcal{D}} \mathcal{L}\Big(P_G(derives(G_i\theta_i, T_i); p), t_i \Big)$$

G_i = Goal
θ_i = substitution
T_i = sequence of terminals
t_i = target probability
D = dataset
L = loss function

DeepStochLog ≈ DeepProbLog for SLP

Sibling of DeepProbLog, but different semantic (PLP vs SLP)

DeepProbLog: neural predicate probability *(and thus implicitly over "possible worlds")*

$$\texttt{nn(m,[}I_1,\ldots,I_m\texttt{],O,[}x_1,\ldots,x_L\texttt{])} :: \texttt{neural_predicate(X).}$$

DeepStochLog: neural grammar rule probability *(and thus no disjoint sum problem)*

$$\texttt{nn(m,[}I_1,\ldots,I_m\texttt{],[}O_1,\ldots,O_L\texttt{],[}D_1,\ldots,D_L\texttt{])} :: \texttt{nt} \texttt{ --> } g_1, \ldots, g_n.$$

PLP vs SLP ~ akin to difference between random graph and random walk

Experiments

Research questions

Q1: Does DeepStochLog reach state-of-the-art predictive performance on neural-symbolic tasks?

Q2: How does the inference time of DeepStochLog compare to other neural-symbolic frameworks and what is the role of tabling?

Q3: Can DeepStochLog handle larger-scale tasks?

Q4: Can DeepStochLog go beyond grammars and encode more general programs?

Mathematical expression outcome

T1: Summing MNIST numbers with pre-specified # digits

 = 137

T2: Expressions with images representing operator or single digit number.

 = 19

Table 1: The test accuracy (%) on the MNIST addition (**T1**).

	Number of digits per number (N)			
Methods	1	2	3	4
NeurASP	97.3 ± 0.3	93.9 ± 0.7	timeout	timeout
DeepProbLog	97.2 ± 0.5	95.2 ± 1.7	timeout	timeout
DeepStochLog	97.9 ± 0.1	96.4 ± 0.1	94.5 ± 1.1	92.7 ± 0.6

Table 2: The accuracy (%) on the HWF dataset (**T2**).

	Expression length			
Method	1	3	5	7
NGS	90.2 ± 1.6	85.7 ± 1.0	91.7 ± 1.3	20.4 ± 37.2
DeepProbLog	90.8 ± 1.3	85.6 ± 1.1	timeout	timeout
DeepStochLog	90.8 ± 1.0	86.3 ± 1.9	92.1 ± 1.4	94.8 ± 0.9

Performance comparison

Table 7: Inference times in milliseconds for DeepStochLog, DeepProbLog and NeurASP on task **T1** for variable number lengths.

Numbers Length	1	2	3	4
DeepStochLog	1.3 ± 0.9	2.3 ± 0.4	4.0 ± 0.4	5.7 ± 1.8
DeepProbLog	13.5 ± 3.0	36.0 ± 0.5	199.7 ± 14.0	timeout
NeurASP	9.2 ± 1.4	85.7 ± 22.6	158.2 ± 47.7	timeout

Classic grammars, but with MNIST images as terminals

T3: Well-formed brackets as input (without parse). Task: predict parse.



→ parse = ( ) (( ) ( ))

T4: inputs are strings $a^k b^l c^m$ (or permutations of [a,b,c], and (k+l+m)%3=0).
Predict 1 if k=l=m, otherwise 0.

 = 1

 = 0

Table 3: The parse accuracy (%) on the well-formed parentheses dataset (**T3**).

	Maximum expression length		
Method	10	14	18
DeepProbLog	100.0 ± 0.0	99.4 ± 0.5	99.2 ± 0.8
DeepStochLog	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0

Table 4: The accuracy (%) on the $a^n b^n c^n$ dataset (**T4**).

	Expression length		
Method	3-12	3-15	3-18
DeepProbLog	99.8 ± 0.3	timeout	timeout
DeepStochLog	99.4 ± 0.5	99.2 ± 0.4	98.8 ± 0.2

Natural way of expressing this grammar knowledge

```
brackets_dom(X) :- member(X, ["(",")"]).

nn(bracket_nn, [X], Y, brackets_dom) :: bracket(Y) --> [X].


t(_) :: s --> s, s.

t(_) :: s --> bracket("("), s, bracket(")").

t(_) :: s --> bracket("("), bracket(")").
```

All power of Prolog DCGs (here: $a^n b^n c^n$)

```
letter(X) :- member(X, [a,b,c]).

0.5 :: s(0) --> akblcm(K,L,M),
                {K \= L; L \= M; M \= K},
                {K \= 0, L \= 0, M \= 0}.

0.5 :: s(1) --> akblcm(N,N,N).

akblcm(K,L,M) --> rep(K,A),
                  rep(L,B),
                  rep(M,C),
                  {A \= B, B \= C, C \= A}.

rep(0, _) --> [].
nn(mnist, [X], C, letter) :: rep(s(N), C) --> [X],
                                              rep(N,C).
```

Citation networks

T5: Given scientific paper set with only few labels & citation network, find all labels

Table 5: **Q3** Accuracy (%) of the classification on the test nodes on task **T5**

Method	Citeseer	Cora
ManiReg	60.1	59.5
SemiEmb	59.6	59.0
LP	45.3	68.0
DeepWalk	43.2	67.2
ICA	69.1	75.1
GCN	70.3	81.5
DeepProbLog	timeout	timeout
DeepStochLog	65.0	69.4

Word Algebra Problem

T6: natural language text describing algebra problem, predict outcome

E.g. "Mark has 6 apples. He eats 2 and divides the remaining among his 2 friends. How many apples did each friend get?"

Uses *"empty body trick"* to emulate SLP logic rules through SDCGs:

$$\texttt{nn(m,[}I_1\texttt{,…,}I_m\texttt{],[}O_1\texttt{,…,}O_L\texttt{],[}D_1\texttt{,…,}D_L\texttt{]) :: nt --> [].}$$

Enables fairly straightforward translation of DeepProbLog programs for a lot of tasks

DeepStochLog performs equally well as DeepProbLog: 96% accuracy

Translating DPL to DSL

```
nn(nn_permute, [X],Y,[0,1,2,3,4,5]) :: net1(X,Y).
nn(nn_op1, [X],Y, [plus,minus,times,div]) :: net2(X,Y).
nn(nn_swap, [X],Y, [no_swap,swap]) :: net3(X,Y).
nn(nn_op2, [X],Y, [plus,minus,times,div]) :: net4(X,Y).

permute(0,A,B,C,A,B,C).
permute(1,A,B,C,A,C,B).
permute(2,A,B,C,B,A,C).
permute(3,A,B,C,B,C,A).
permute(4,A,B,C,C,A,B).
permute(5,A,B,C,C,B,A).


swap(no_swap,X,Y,X,Y).
swap(swap,X,Y,Y,X).

operator(plus,X,Y,Z) :- Z is X+Y.
operator(minus,X,Y,Z) :- Z is X-Y.
operator(times,X,Y,Z) :- Z is X*Y.
operator(div,X,Y,Z) :- Y > 0, 0 =:= X mod Y,Z is X//Y.

wap(Text,X1,X2,X3,Out) :-
    net1(Text,Perm),
    permute(Perm,X1,X2,X3,N1,N2,N3),
    net2(Text,Op1),
    operator(Op1,N1,N2,Res1),
    net3(Text,Swap),
    swap(Swap,Res1,N3,X,Y),
    net4(Text,Op2),
    operator(Op2,X,Y,Out).
```

DeepProbLog

```
dom_permute(X) :- member(X, [0,1,2,3,4,5]).
dom_op1(X) :- member(X, [plus,minus,times,div]).
dom_swap(X) :- member(X,[no_swap,swap]).
dom_op2(X) :- member(X, [plus,minus,times,div]).

nn(nn_permute, [Embed, Perm], Perm, dom_permute) :: nn_permute(Embed,Perm) --> [].
nn(nn_op1, [Embed], Op1, dom_op1) :: nn_op1(Embed, Op1) --> [].
nn(nn_swap, [Embed], Swap, dom_swap) :: nn_swap(Embed, Swap) --> [].
nn(nn_op2, [Embed], Op2, dom_op2) :: nn_op2(Embed, Op2) --> [].

0.1666666 :: permute(0,A,B,C,A,B,C) --> [].
0.1666666 :: permute(1,A,B,C,A,C,B) --> [].
0.1666666 :: permute(2,A,B,C,B,A,C) --> [].
0.1666666 :: permute(3,A,B,C,B,C,A) --> [].
0.1666666 :: permute(4,A,B,C,C,A,B) --> [].
0.1666666 :: permute(5,A,B,C,C,B,A) --> [].

0.5 :: swap(no_swap,X,Y,X,Y) --> [].
0.5 :: swap(swap,X,Y,Y,X) --> [].

0.25 :: operator(plus,X,Y,Z) --> [], {Z is X+Y}.
0.25 :: operator(minus,X,Y,Z) --> [], {Z is X-Y}.
0.25 :: operator(times,X,Y,Z) --> [], {Z is X*Y}.
0.25 :: operator(div,X,Y,Z) --> [], {Y > 0, 0 =:= X mod Y, Z is X//Y}.

s(Out,X1,X2,X3) --> [String],
                    nn_permute(String, Perm),
                    nn_op1(String, Op1),
                    nn_swap(String, Swap),
                    nn_op2(String, Op2),
                    permute(Perm,X1,X2,X3,N1,N2,N3),
                    operator(Op1,N1,N2,Res1),
                    swap(Swap,Res1,N3,X,Y),
                    operator(Op2,X,Y,Out).
```

DeepStochLog

Ongoing/future research

Ongoing/future DeepStochLog research

- Structure learning using t(_) *(syntactic sugar for PRISM-like switches)*
- Support for calculation of true probability *(without probability mass loss)* by dividing by probabilities of all possible other possible outputs of given goal
- Larger-scale experiment(s) *(e.g. on CLEVR)*
- Comparing with more neural symbolic frameworks
- Generative setting *(with generative neural networks and using grammar to aid generation)*

Thanks!

Paper: DeepStochLog: Neural Stochastic Logic Programming

Code: https://github.com/ml-kuleuven/deepstochlog

Python: `pip install deepstochlog`

Authors: Thomas Winters*, Giuseppe Marra*, Robin Manhaeve, Luc De Raedt

Twitter: @thomas_wint, @giuseppe__marra, @ManhaeveRobin, @lucderaedt

** shared first author*