End-to-End Differentiable Proving

Tim Rocktäschel[1] and Sebastian Riedel[2,3]

[1]University of Oxford
Whiteson Research Lab

[2]University College London
Machine Reading Lab

[3]Bloomsbury AI

http://rockt.github.com Twitter: @_rockt tim.rocktaschel@cs.ox.ac.uk

NIPS 2017

6th of December 2017

Combining Deep and Symbolic Reasoning

Neural Networks

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization
- Needs a lot of training data
- Not interpretable

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization
- Needs a lot of training data
- Not interpretable

First-order Logic Expert Systems

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization
- Needs a lot of training data
- Not interpretable

First-order Logic Expert Systems

- Rules manually defined
- No generalization

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization
- Needs a lot of training data
- Not interpretable

First-order Logic Expert Systems

- Rules manually defined
- No generalization
- No/little training data
- Interpretable

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization

First-order Logic Expert Systems

- No/little training data
- Interpretable

Combining Deep and Symbolic Reasoning

Neural Networks
- Trained end-to-end
- Strong generalization

First-order Logic Expert Systems
- No/little training data
- Interpretable

Aim

- Neural network for proving queries to a knowledge base

Combining Deep and Symbolic Reasoning

Neural Networks
- Trained end-to-end
- Strong generalization

First-order Logic Expert Systems
- No/little training data
- Interpretable

Aim

- Neural network for proving queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols

Combining Deep and Symbolic Reasoning

Neural Networks
- Trained end-to-end
- Strong generalization

First-order Logic Expert Systems
- No/little training data
- Interpretable

Aim

- Neural network for proving queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learn vector representations of symbols** end-to-end from proof success

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization

First-order Logic Expert Systems

- No/little training data
- Interpretable

Aim

- Neural network for proving queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learn vector representations of symbols** end-to-end from proof success
- **Make use of provided rules** in soft proofs

Combining Deep and Symbolic Reasoning

Neural Networks

- Trained end-to-end
- Strong generalization

First-order Logic Expert Systems

- No/little training data
- Interpretable

Aim

- Neural network for proving queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learn vector representations of symbols** end-to-end from proof success
- **Make use of provided rules** in soft proofs
- **Induce interpretable rules** end-to-end from proof success

Machine Learning & Logic

- Fuzzy Logic (Zadeh, 1965)

Machine Learning & Logic

- Fuzzy Logic (Zadeh, 1965)
- Probabilistic Logic Programming, e.g.,
  - IBAL (Pfeffer, 2001), BLOG (Milch et al., 2005), Markov Logic Networks (Richardson and Domingos, 2006), ProbLog (De Raedt et al., 2007) ...

Machine Learning & Logic

- Fuzzy Logic (Zadeh, 1965)
- Probabilistic Logic Programming, e.g.,
  - IBAL (Pfeffer, 2001), BLOG (Milch et al., 2005), Markov Logic Networks (Richardson and Domingos, 2006), ProbLog (De Raedt et al., 2007) ...
- Inductive Logic Programming, e.g.,
  - Plotkin (1970), Shapiro (1991), Muggleton (1991), De Raedt (1999) ...
  - Statistical Predicate Invention (Kok and Domingos, 2007)

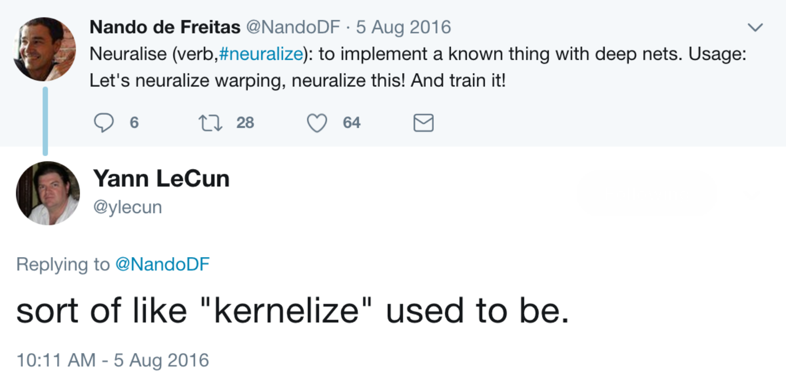
Machine Learning & Logic

- Fuzzy Logic (Zadeh, 1965)
- Probabilistic Logic Programming, e.g.,
  - IBAL (Pfeffer, 2001), BLOG (Milch et al., 2005), Markov Logic Networks (Richardson and Domingos, 2006), ProbLog (De Raedt et al., 2007) ...
- Inductive Logic Programming, e.g.,
  - Plotkin (1970), Shapiro (1991), Muggleton (1991), De Raedt (1999) ...
  - Statistical Predicate Invention (Kok and Domingos, 2007)
- Neural-symbolic Connectionism
  - Propositional rules: EBL-ANN (Shavlik and Towell, 1989), KBANN (Towell and Shavlik, 1994), C-LIP (Garcez and Zaverucha, 1999)
  - First-order inference (no training of symbol representations): Unification Neural Networks (Holldöbler, 1990; Komendantskaya 2011), SHRUTI (Shastri, 1992), Neural Prolog (Ding, 1995), CLIP++ (Franca et al. 2014), Lifted Relational Networks (Sourek et al. 2015)

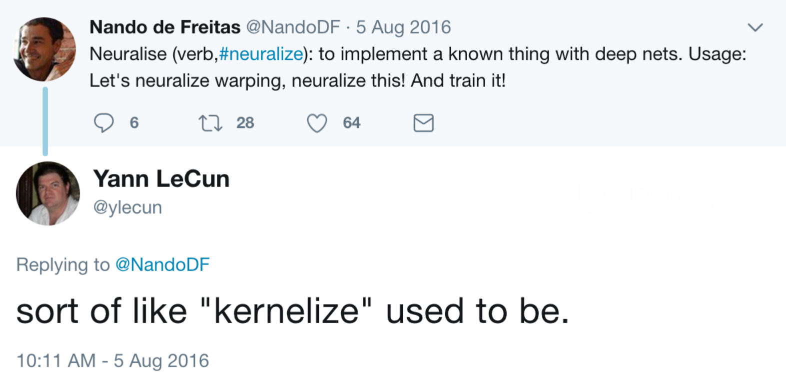
Approach


Nando de Freitas @NandoDF · 5 Aug 2016
Neuralise (verb,#neuralize): to implement a known thing with deep nets. Usage: Let's neuralize warping, neuralize this! And train it!
6
28
64

Approach


Nando de Freitas @NandoDF · 5 Aug 2016
Neuralise (verb,#neuralize): to implement a known thing with deep nets. Usage: Let's neuralize warping, neuralize this! And train it!
6
28
64
Yann LeCun
@ylecun
Replying to @NandoDF
sort of like "kernelize" used to be.
10:11 AM - 5 Aug 2016

Approach



Let's **neuralize** Prolog's Backward Chaining using a Radial Basis Function **kernel** for unifying vector representations of symbols!

Prolog's Backward Chaining

Example Knowledge Base:

1. `fatherOf`(ABE, HOMER).
2. `parentOf`(HOMER, BART).
3. `grandfatherOf`(X, Y) :–
   `fatherOf`(X, Z),
   `parentOf`(Z, Y).

Prolog's Backward Chaining

Example Knowledge Base:

1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).

Intuition:

- Backward chaining translates a query into subqueries via rules, *e.g.*,
  grandfatherOf(ABE, BART) $\xrightarrow{3.}$ fatherOf(ABE, Z), parentOf(Z, BART)

Prolog's Backward Chaining

Example Knowledge Base:

1. $\texttt{fatherOf}(\text{ABE}, \text{HOMER}).$
2. $\texttt{parentOf}(\text{HOMER}, \text{BART}).$
3. $\texttt{grandfatherOf}(X, Y) \text{ :– }$
   $\texttt{fatherOf}(X, Z),$
   $\texttt{parentOf}(Z, Y).$

Intuition:

- Backward chaining translates a query into subqueries via rules, *e.g.*,
  $\texttt{grandfatherOf}(\text{ABE}, \text{BART}) \xrightarrow{3.} \texttt{fatherOf}(\text{ABE}, Z), \texttt{parentOf}(Z, \text{BART})$
- It attempts this for all rules in the knowledge base and thus specifies a depth-first search

Unification

Example Knowledge Base:

1. `fatherOf`(ABE, HOMER).
2. `parentOf`(HOMER, BART).
3. `grandfatherOf`(X, Y) :–
   `fatherOf`(X, Z),
   `parentOf`(Z, Y).

Query

`grandfatherOf` ABE BART

Unification

Example Knowledge Base:

1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
    fatherOf(X, Z),
    parentOf(Z, Y).

Query

grandfatherOf	ABE	BART

1.	fatherOf	ABE	HOMER

Unification

Example Knowledge Base:

1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).

Query

grandfatherOf	ABE	BART
$\stackrel{?}{=}$	$\stackrel{?}{=}$	$\stackrel{?}{=}$
1. fatherOf	ABE	HOMER

Unification

Example Knowledge Base:

1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).

Query

grandfatherOf ABE BART

$\stackrel{?}{=}$ $\stackrel{?}{=}$ $\stackrel{?}{=}$

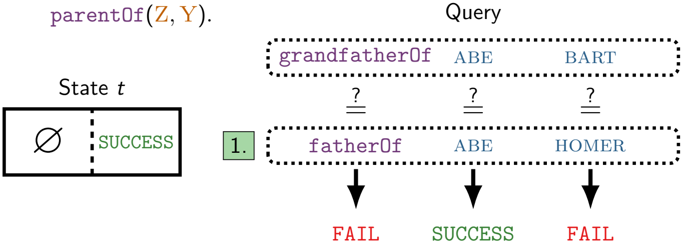
1. fatherOf ABE HOMER

FAIL SUCCESS FAIL

Unification

Example Knowledge Base:

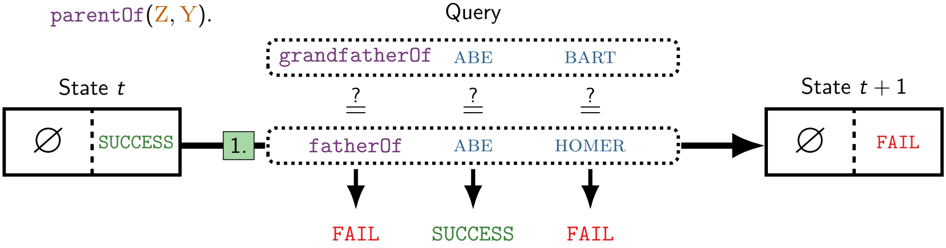
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Unification

Example Knowledge Base:

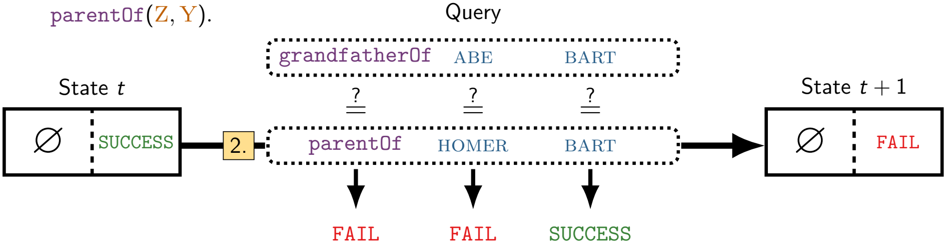
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Unification

Example Knowledge Base:

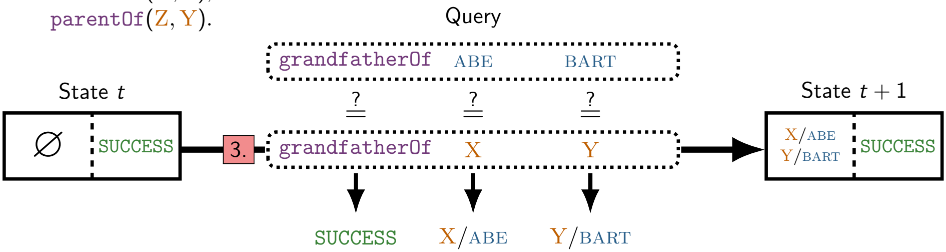
1. `fatherOf`(ABE, HOMER).
2. `parentOf`(HOMER, BART).
3. `grandfatherOf`(X, Y) :–
   `fatherOf`(X, Z),
   `parentOf`(Z, Y).



Unification

Example Knowledge Base:

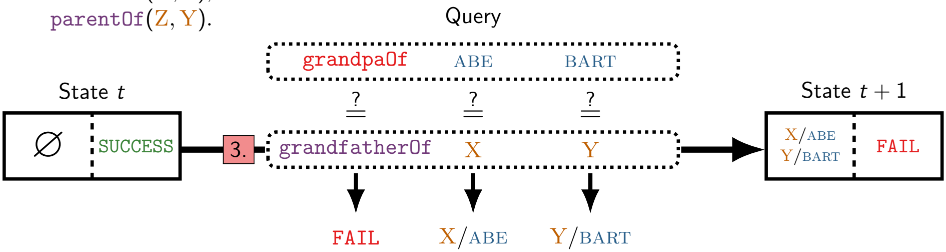
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Unification Failure

Example Knowledge Base:

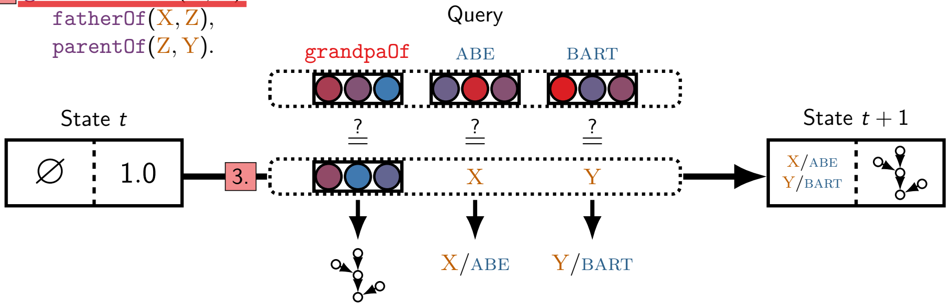
1. `fatherOf`(ABE, HOMER).
2. `parentOf`(HOMER, BART).
3. `grandfatherOf`(X, Y) :–
   `fatherOf`(X, Z),
   `parentOf`(Z, Y).



Neural Unification

Example Knowledge Base:

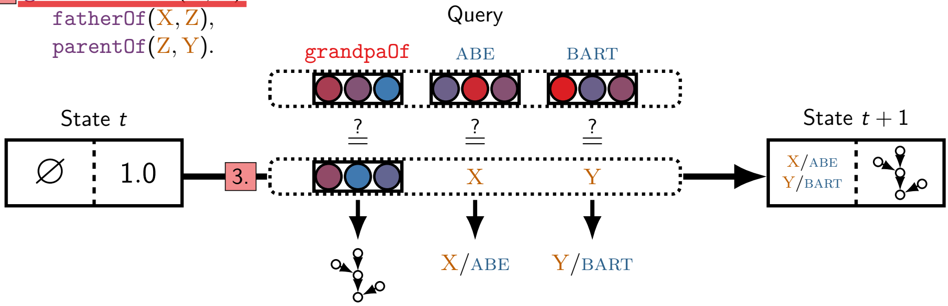
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Neural Unification

Example Knowledge Base:

1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).

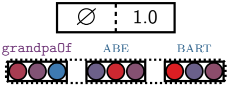


$$\min\left(1.0, \exp\left(\frac{-\|\mathbf{v}_{\texttt{grandpaOf}} - \mathbf{v}_{\texttt{grandfatherOf}}\|_2}{2\mu^2}\right)\right)$$

Differentiable Prover

Example Knowledge Base:

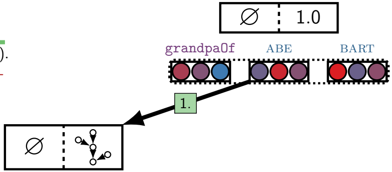
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

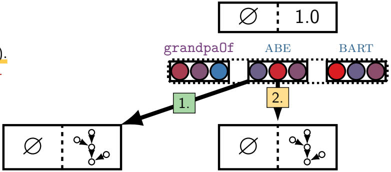
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

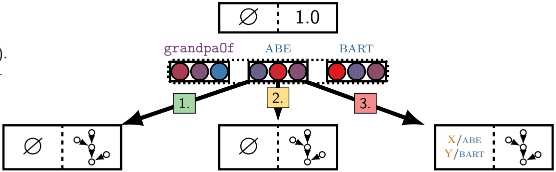
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).

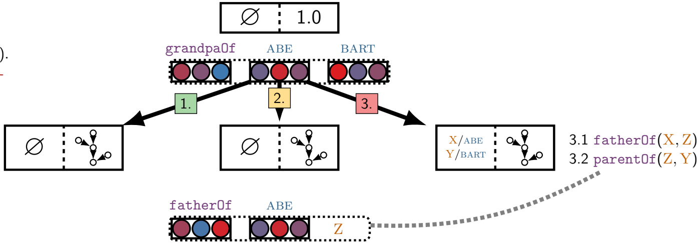


3.1 fatherOf(X, Z)
3.2 parentOf(Z, Y)

Differentiable Prover

Example Knowledge Base:

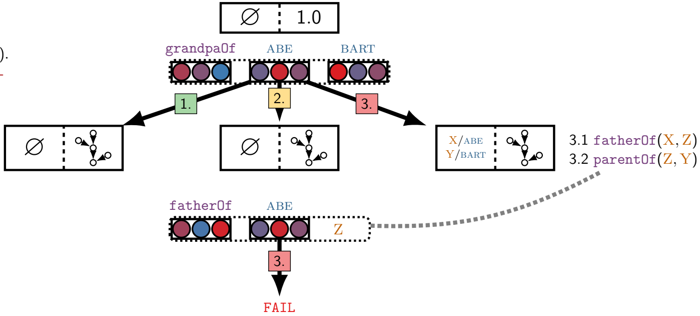
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

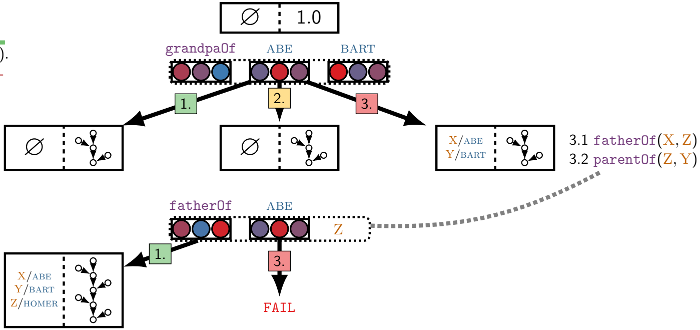
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

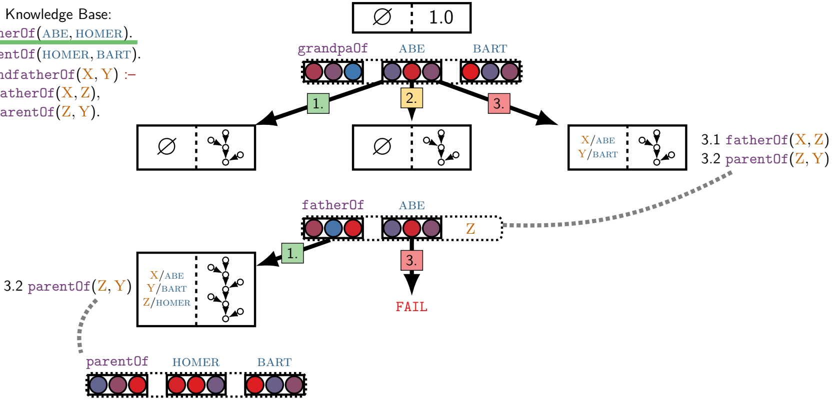
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
    fatherOf(X, Z),
    parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

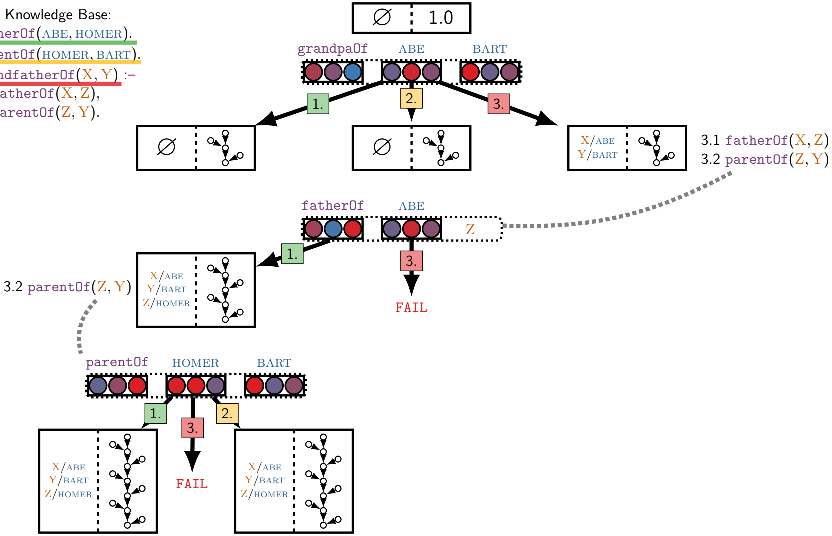
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

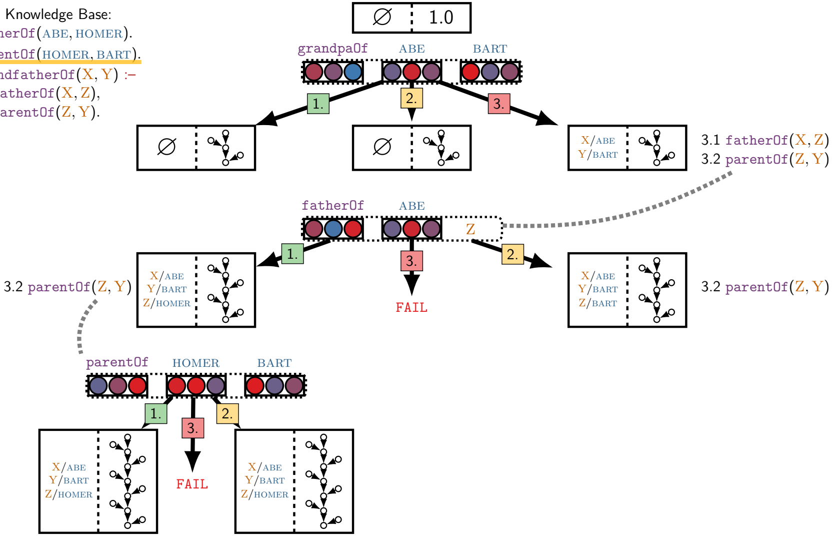
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

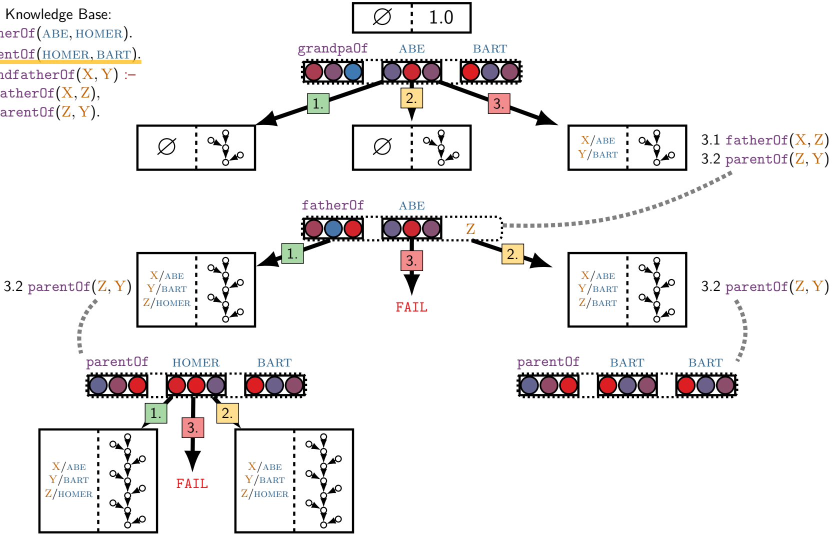
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
    fatherOf(X, Z),
    parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

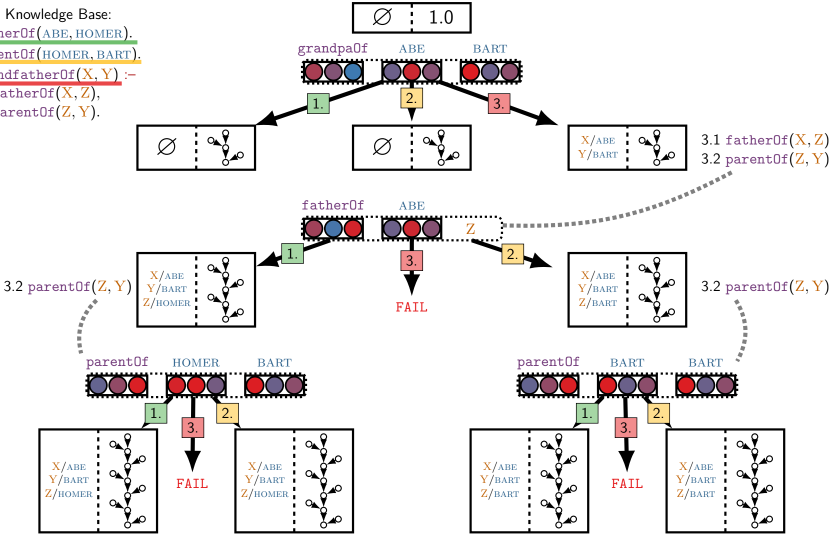
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

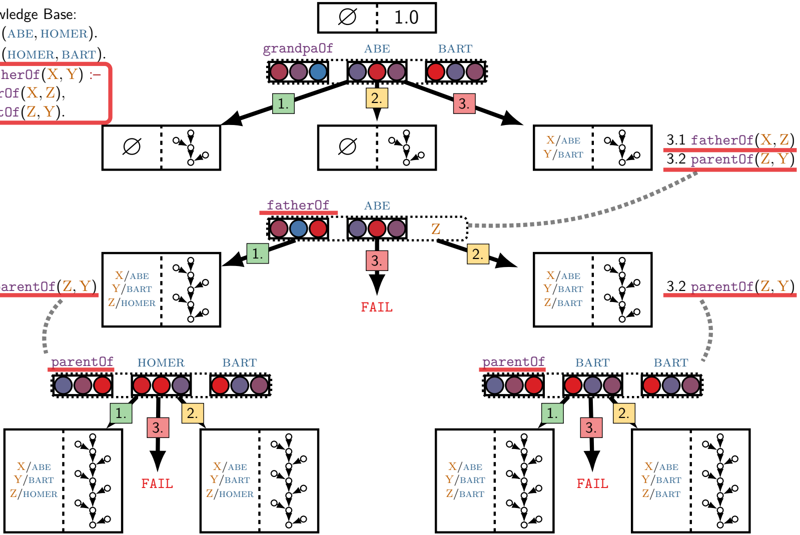
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



Neural Program Induction

Example Knowledge Base:

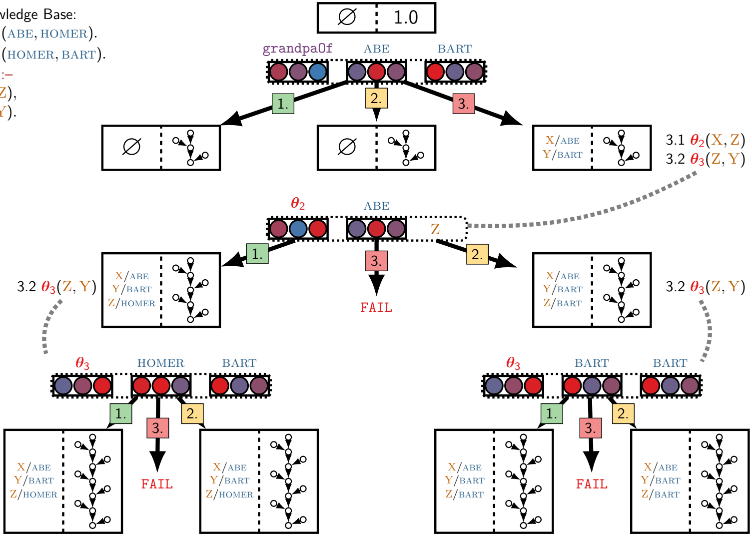
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :–
   fatherOf(X, Z),
   parentOf(Z, Y).



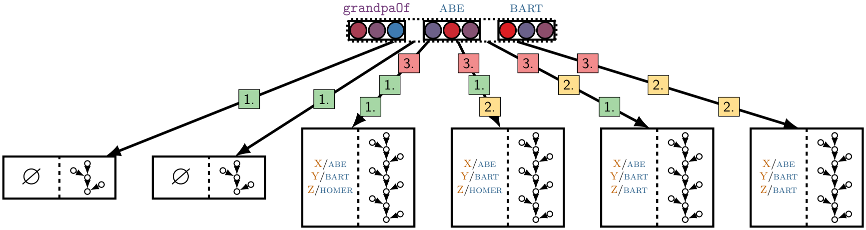
Neural Program Induction

Example Knowledge Base:

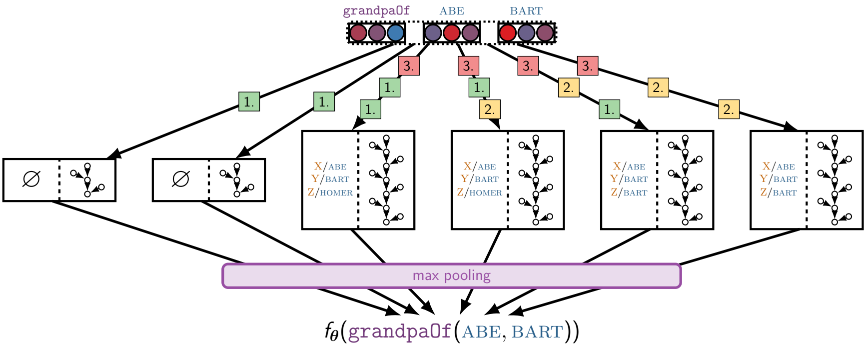
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. $\theta_1(X, Y)$:–
   $\theta_2(X, Z)$,
   $\theta_3(Z, Y)$.



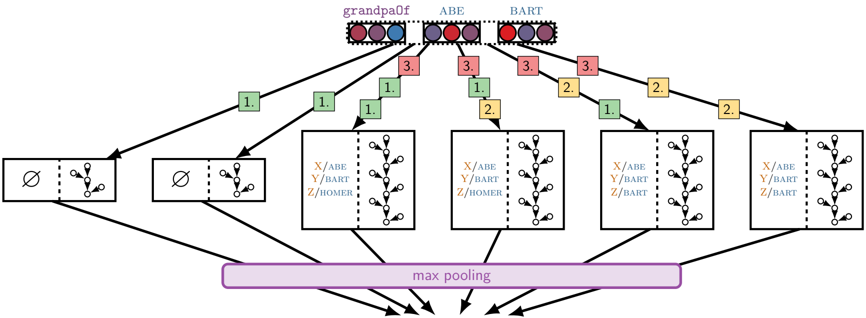
Training Objective



Training Objective



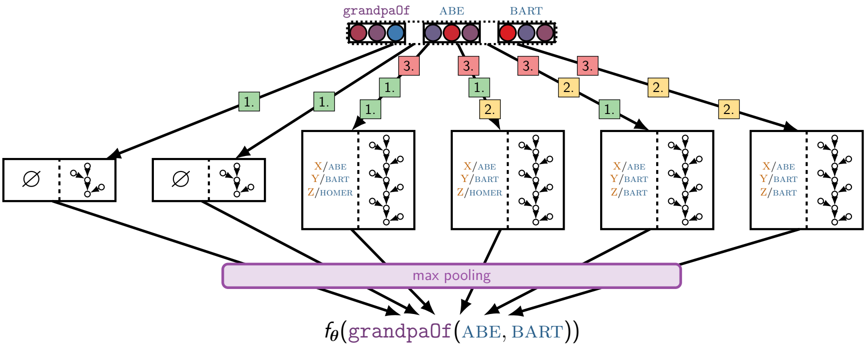
Training Objective



$f_\theta(\texttt{grandpaOf}(\text{ABE}, \text{BART}))$

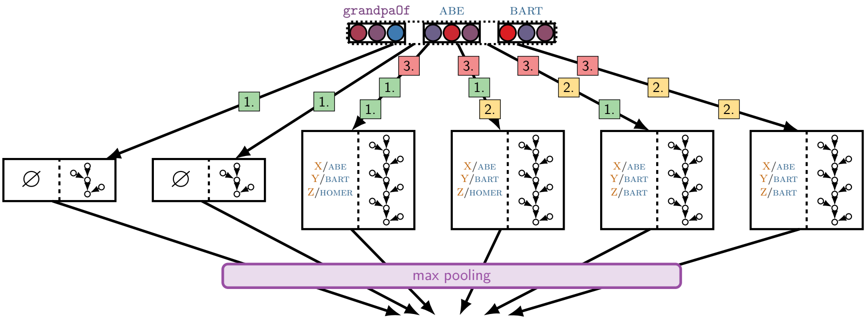
- Loss: negative log-likelihood w.r.t. target proof success

Training Objective



- Loss: negative log-likelihood w.r.t. target proof success
- Trained end-to-end using stochastic gradient descent

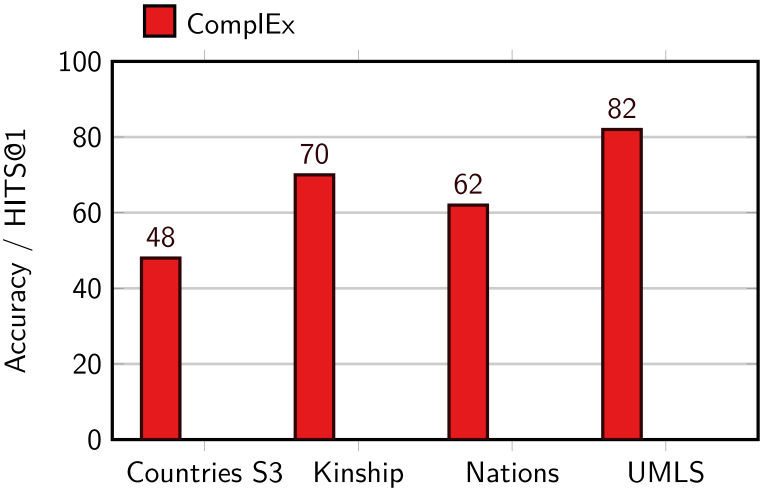
Training Objective



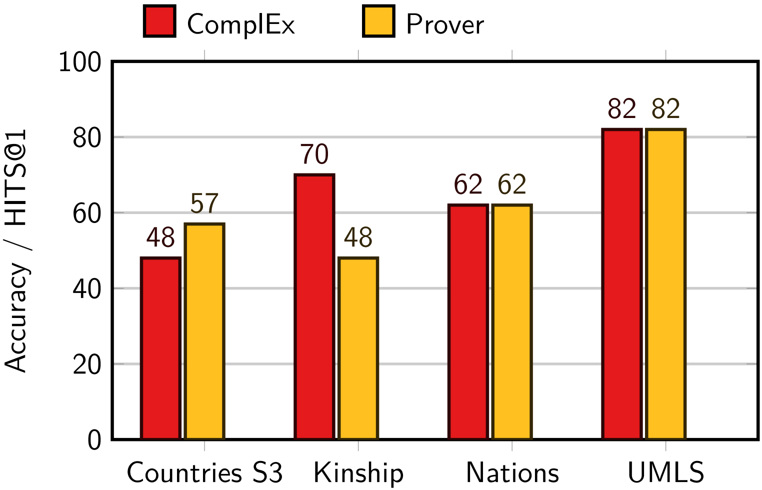
$f_\theta(\texttt{grandpaOf}(\text{ABE}, \text{BART}))$

- Loss: negative log-likelihood w.r.t. target proof success
- Trained end-to-end using stochastic gradient descent
- Vectors are **learned such that proof success is high for known facts** and low for sampled negative facts

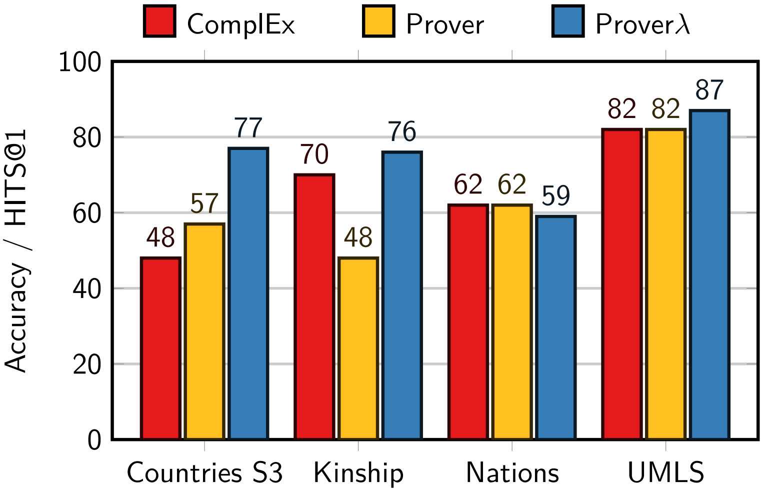
Results


ComplEx
Accuracy / HITS@1
100
80
60
40
20
0
48
70
62
82
Countries S3
Kinship
Nations
UMLS

Results


ComplEx
Prover
Accuracy / HITS@1
100
80
60
40
20
0
48
57
70
48
62
62
82
82
Countries S3
Kinship
Nations
UMLS

Results


ComplEx
Prover
Proverλ
100
80
60
40
20
0
Accuracy / HITS@1
48
57
77
70
48
76
62
62
59
82
82
87
Countries S3
Kinship
Nations
UMLS

Examples of Induced Rules

$\texttt{locatedIn}(X, Y) \coloneq \texttt{locatedIn}(X, Z), \texttt{locatedIn}(Z, Y).$

$\texttt{interacts\_with}(X, Y) \coloneq \texttt{interacts\_with}(X, Z), \texttt{interacts\_with}(Z, Y).$

$\texttt{derivative\_of}(X, Y) \coloneq \texttt{derivative\_of}(X, Z), \texttt{derivative\_of}(Z, Y).$

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learns vector representations of symbols** from data via gradient descent

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learns vector representations of symbols** from data via gradient descent
- **Induces interpretable rules** from data via gradient descent

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learns vector representations of symbols** from data via gradient descent
- **Induces interpretable rules** from data via gradient descent
- Various computational optimizations: batch proving, tree pruning etc.
  Come to the poster!

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learns vector representations of symbols** from data via gradient descent
- **Induces interpretable rules** from data via gradient descent
- Various computational optimizations: batch proving, tree pruning etc.
  Come to the poster!
- Future research:

Summary

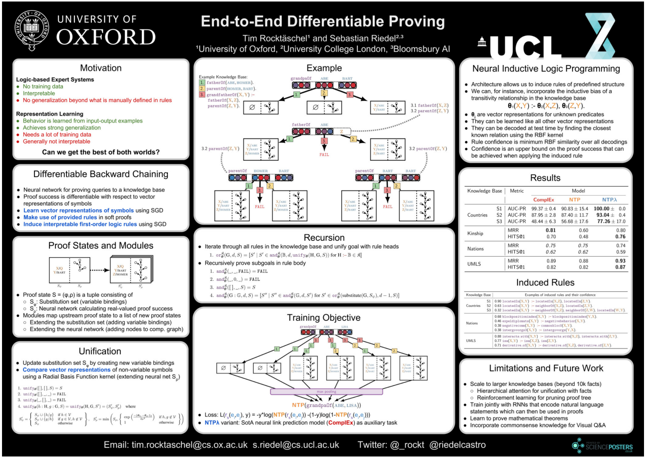
- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learns vector representations of symbols** from data via gradient descent
- **Induces interpretable rules** from data via gradient descent
- Various computational optimizations: batch proving, tree pruning etc.
  Come to the poster!
- Future research:
  - **Scaling up** to larger knowledge bases

Summary

- We used Prolog's backward chaining as recipe for recursively constructing a neural network to prove queries to a knowledge base
- Proof success differentiable w.r.t. vector representations of symbols
- **Learns vector representations of symbols** from data via gradient descent
- **Induces interpretable rules** from data via gradient descent
- Various computational optimizations: batch proving, tree pruning etc.
  Come to the poster!
- Future research:
  - **Scaling up** to larger knowledge bases
  - **Connecting to RNNs** for proving with natural language statements

Thank you!

Poster: Today 6:30-10:30pm, Pacific Ballroom #128



http://rockt.github.com Twitter: @_rockt tim.rocktaschel@cs.ox.ac.uk