

End-to-End Differentiable Proving

Tim Rocktäschel¹ and Sebastian Riedel^{2,3}



¹University of Oxford
Whiteson Research Lab

<http://rockt.github.com>



²University College London
Machine Reading Lab

Twitter: @_rockt



³Bloomsbury AI

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



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



Yann LeCun

@ylecun

Replying to @NandoDF

sort of like "kernelize" used to be.

10:11 AM - 5 Aug 2016

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. `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)`
- 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

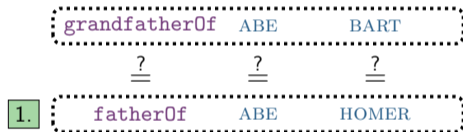
1. fatherOf ABE HOMER

Unification

Example Knowledge Base:

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

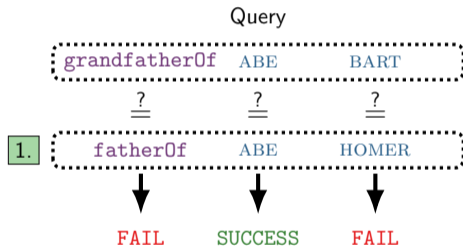
Query



Unification

Example Knowledge Base:

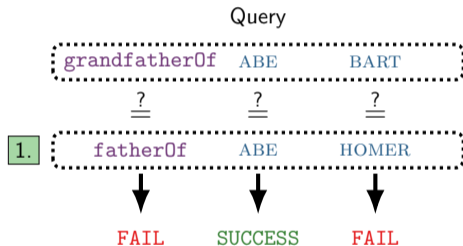
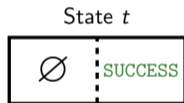
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :-
 fatherOf(X, Z),
 parentOf(Z, Y).



Unification

Example Knowledge Base:

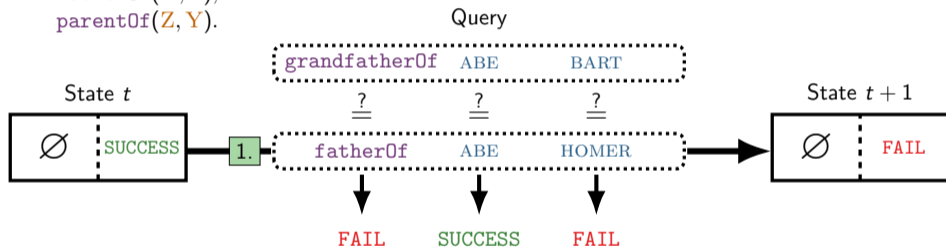
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-
 fatherOf(X, Z),
 parentOf(Z, Y).`



Unification

Example Knowledge Base:

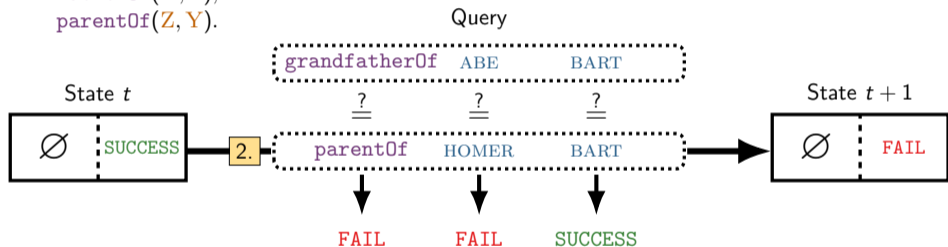
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :-
 fatherOf(X, Z),
 parentOf(Z, Y).



Unification

Example Knowledge Base:

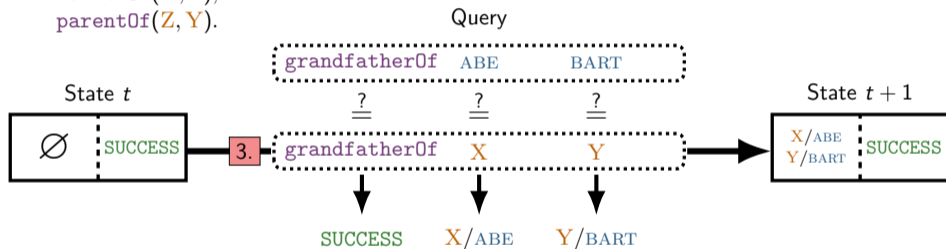
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-
 fatherOf(X, Z),
 parentOf(Z, Y).`



Unification

Example Knowledge Base:

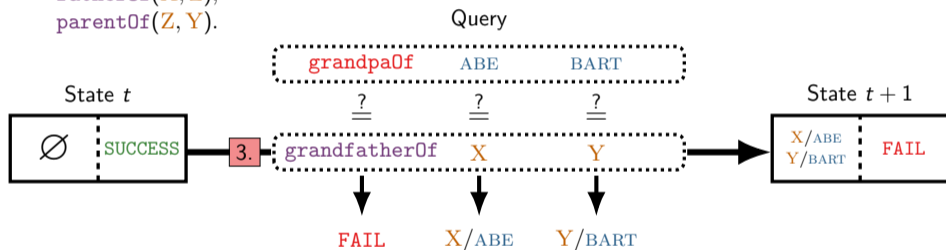
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
 `fatherOf(X, Z),`
 `parentOf(Z, Y).`



Unification Failure

Example Knowledge Base:

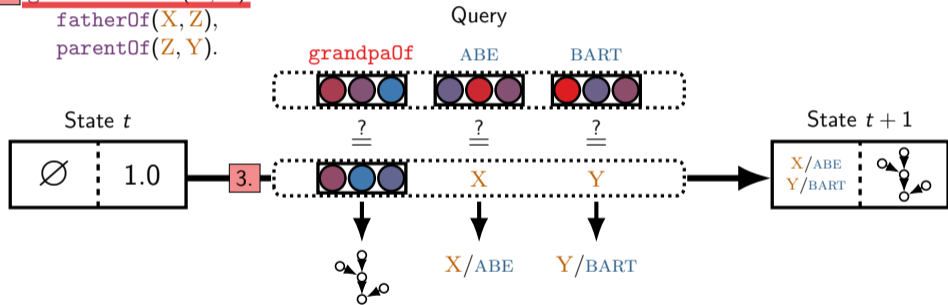
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
 `fatherOf(X, Z),`
 `parentOf(Z, Y).`



Neural Unification

Example Knowledge Base:

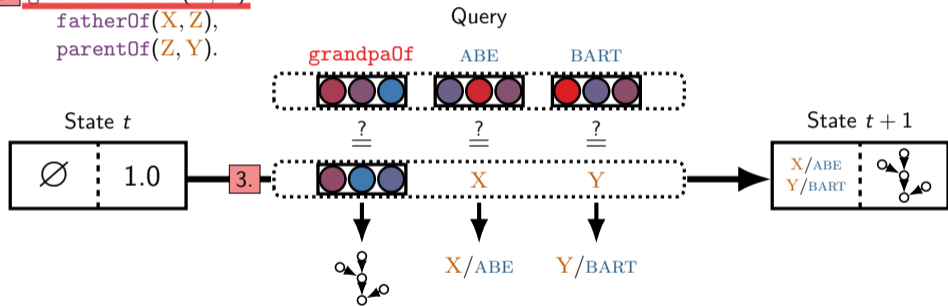
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Neural Unification

Example Knowledge Base:

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



$$\min \left(1.0, \exp \left(\frac{-\|v_{\text{grandpaOf}} - v_{\text{grandfatherOf}}\|^2}{2\mu^2} \right) \right)$$

Differentiable Prover

Example Knowledge Base:

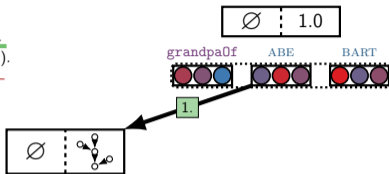
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-
fatherOf(X, Z),
parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

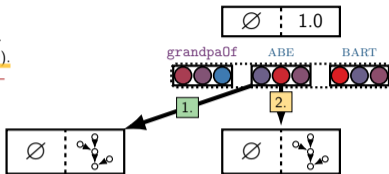
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :-
fatherOf(X, Z),
parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

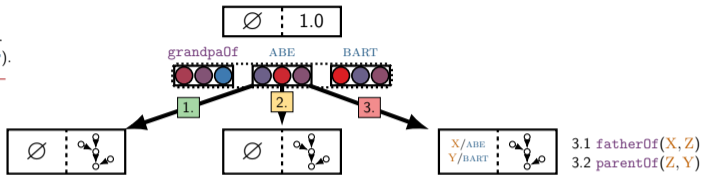
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-
fatherOf(X, Z),
parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

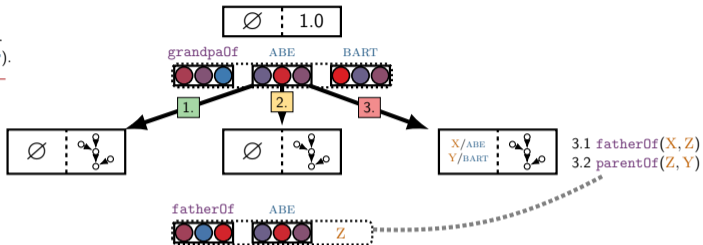
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

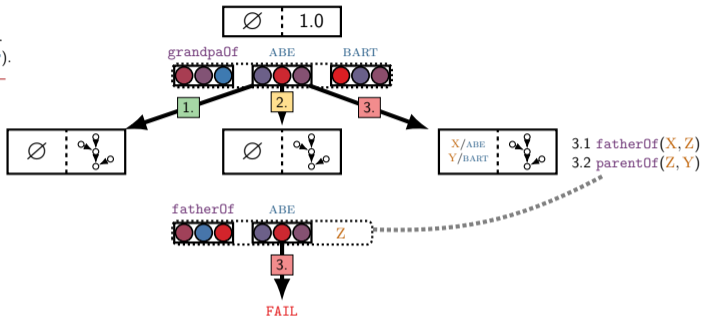
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

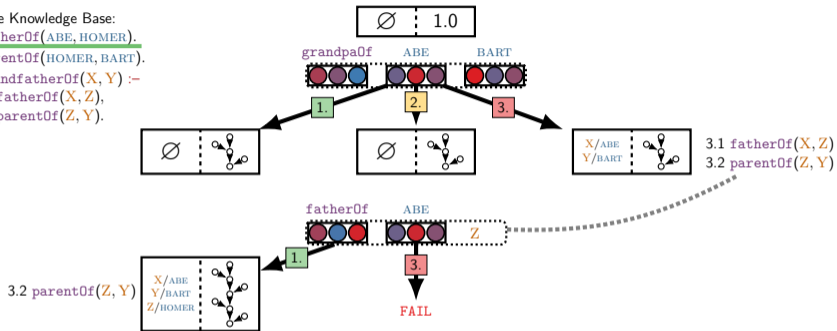
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

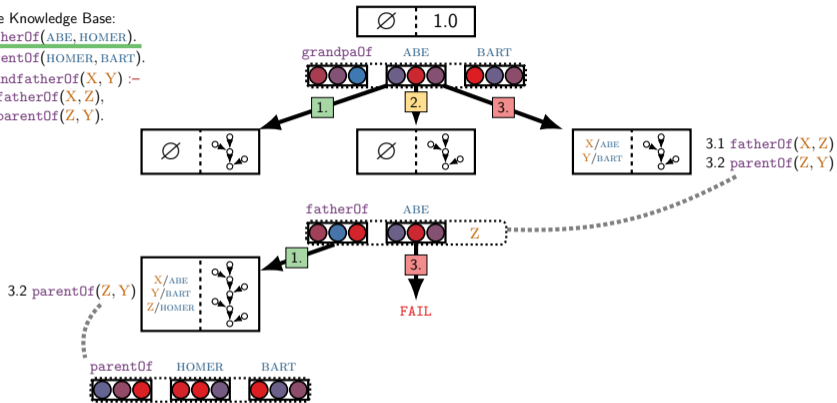
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-
fatherOf(X, Z),
parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

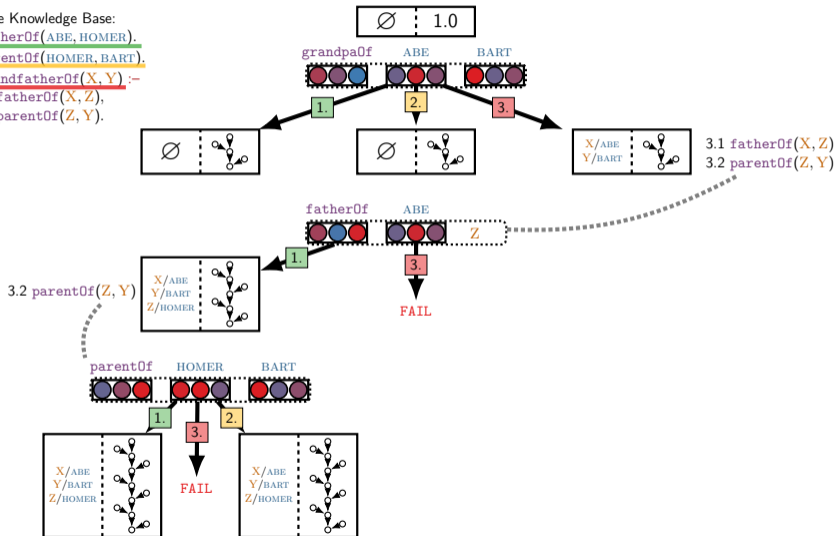
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

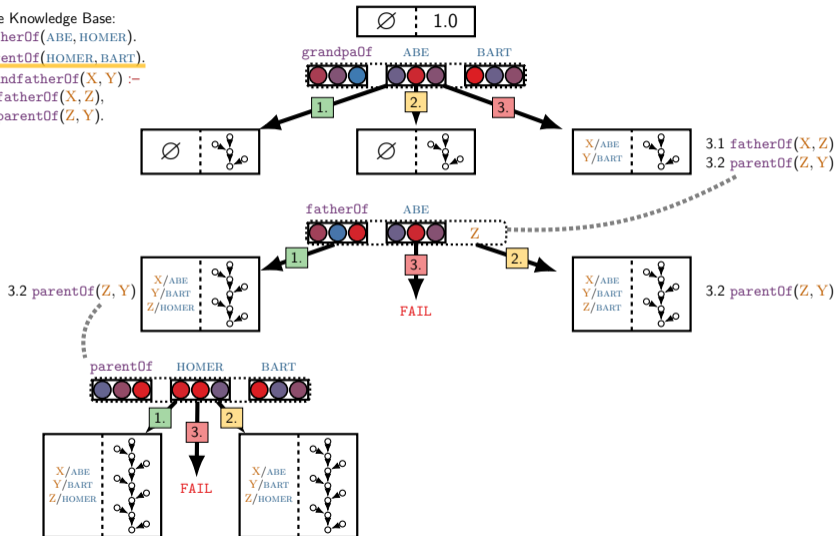
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :-
 fatherOf(X, Z),
 parentOf(Z, Y).



Differentiable Prover

Example Knowledge Base:

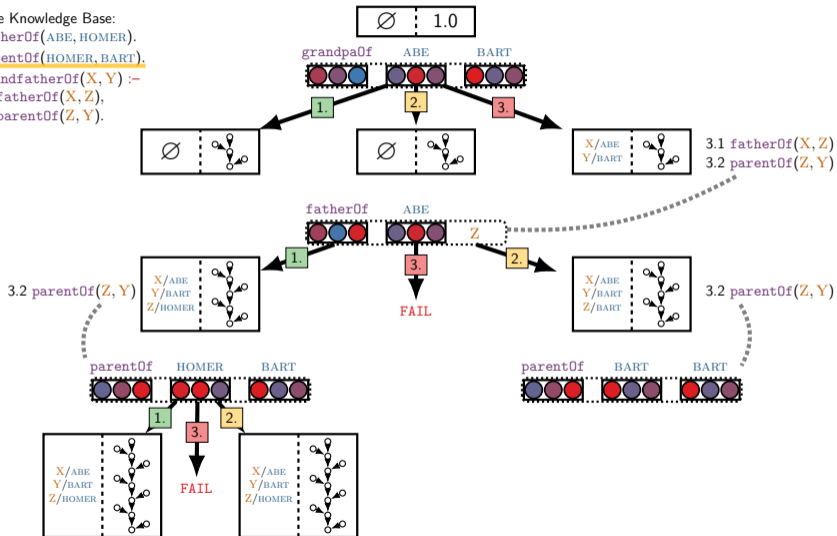
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

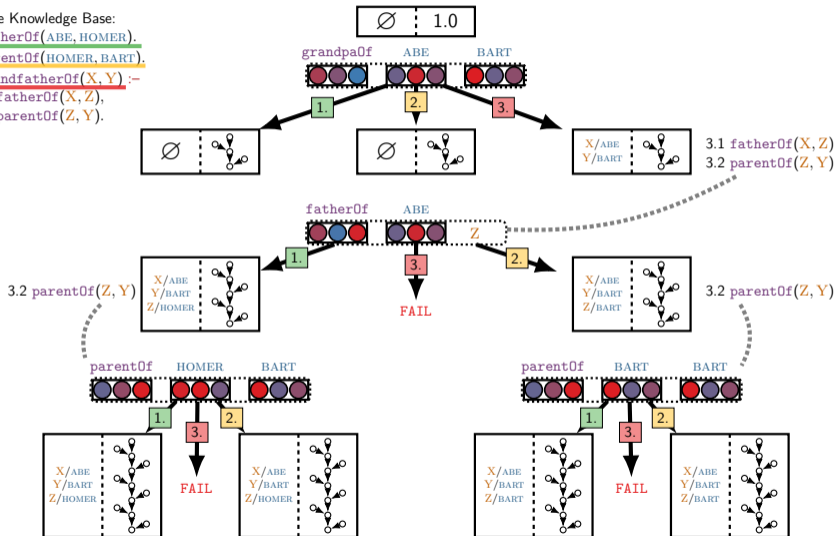
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. `grandfatherOf(X, Y) :-`
`fatherOf(X, Z),`
`parentOf(Z, Y).`



Differentiable Prover

Example Knowledge Base:

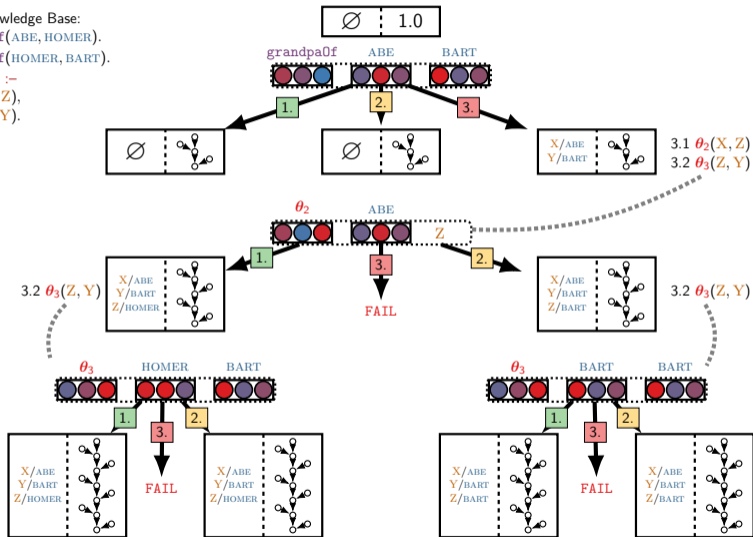
1. fatherOf(ABE, HOMER).
2. parentOf(HOMER, BART).
3. grandfatherOf(X, Y) :-
 fatherOf(X, Z),
 parentOf(Z, Y).



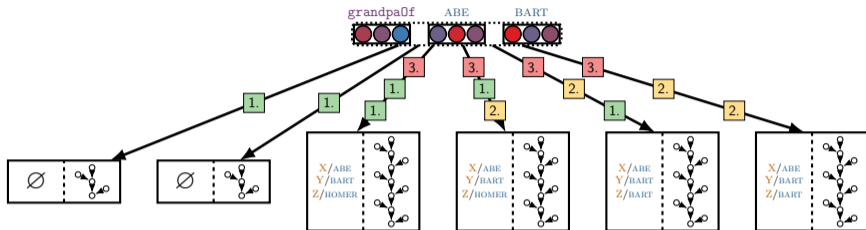
Neural Program Induction

Example Knowledge Base:

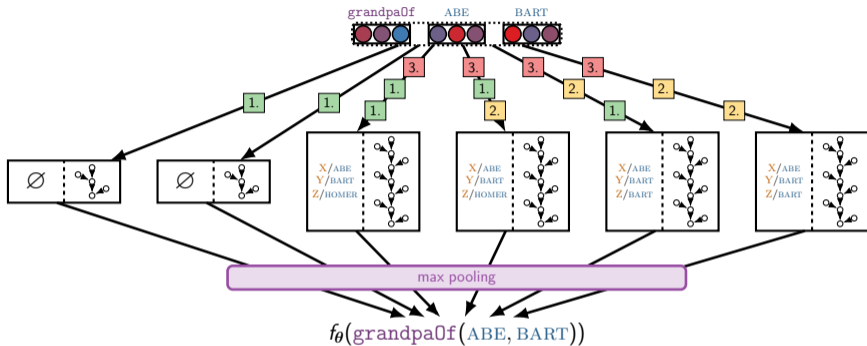
1. `fatherOf(ABE, HOMER).`
2. `parentOf(HOMER, BART).`
3. $\theta_1(X, Y) :-$
 $\theta_2(X, Z),$
 $\theta_3(Z, Y).$



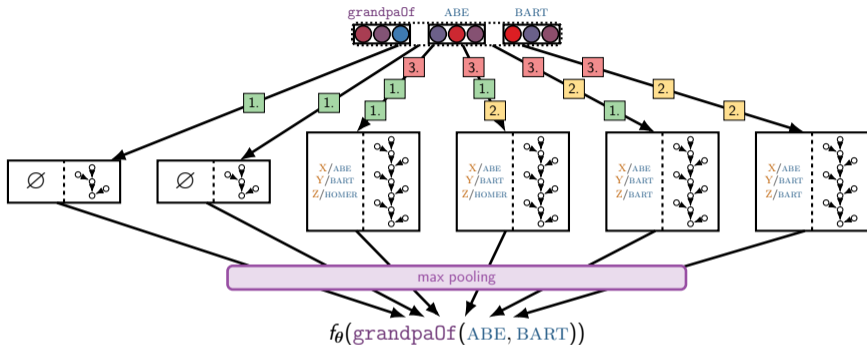
Training Objective



Training Objective

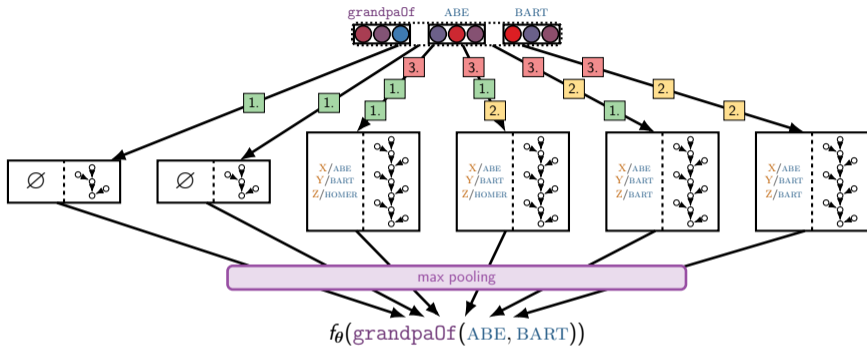


Training Objective



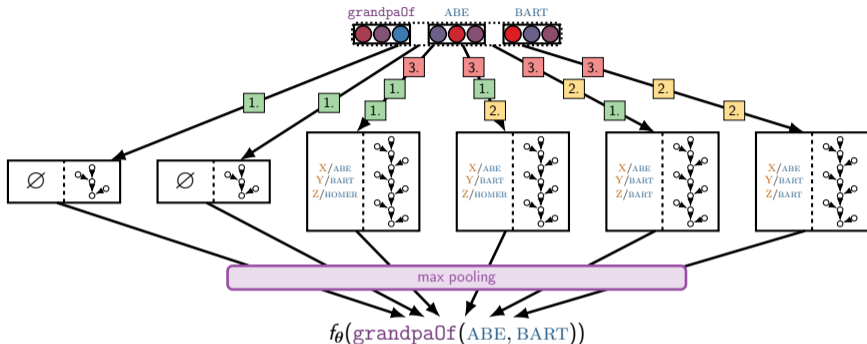
- Loss: negative log-likelihood w.r.t. target proof success

Training Objective



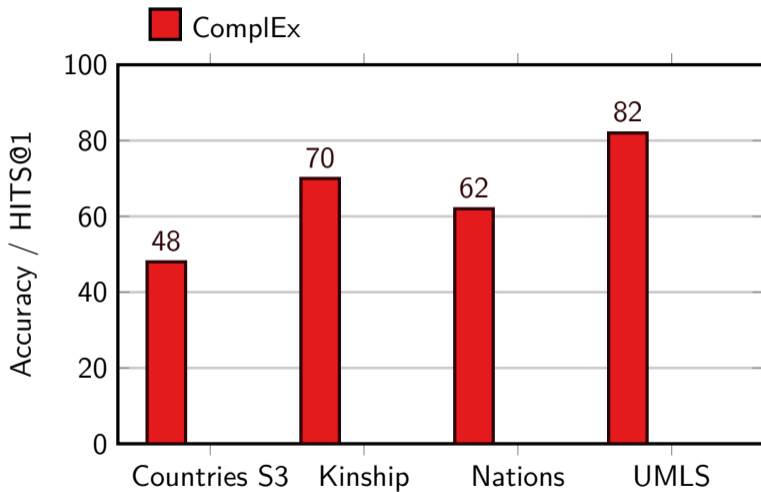
- Loss: negative log-likelihood w.r.t. target proof success
- Trained end-to-end using stochastic gradient descent

Training Objective

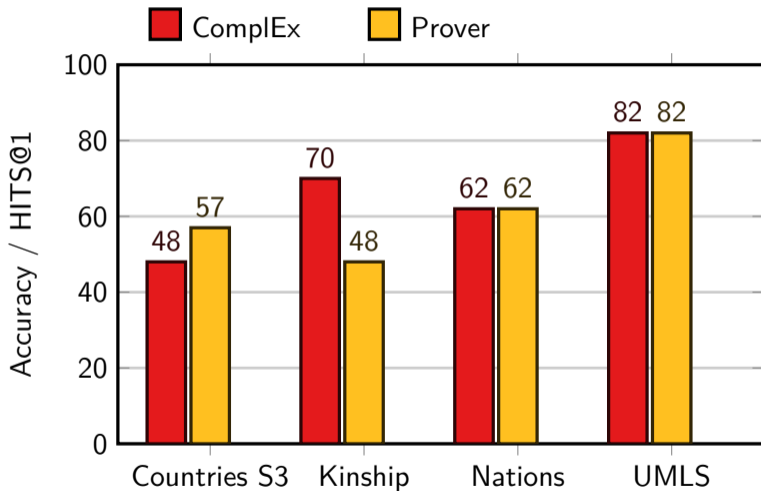


- Loss: negative log-likelihood w.r.t. target proof success
- Trained end-to-end using stochastic gradient descent
- Vectors are **learned such that proof success is high for known facts** and low for sampled negative facts

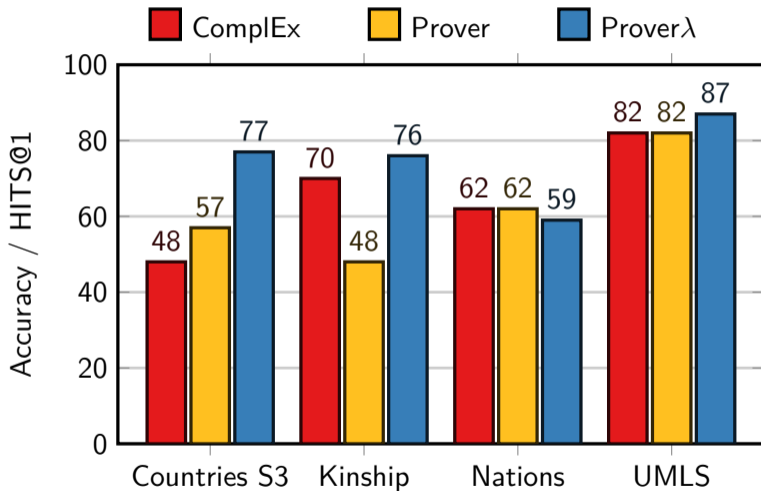
Results



Results



Results



Examples of Induced Rules

`locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y).`

`interacts_with(X, Y) :- interacts_with(X, Z), interacts_with(Z, Y).`

`derivative_of(X, Y) :- derivative_of(X, Z), 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



UNIVERSITY OF
OXFORD

End-to-End Differentiable Proving

Tim Rocktäschel¹ and Sebastian Riedel^{1,2}
¹University of Oxford, ²University College London, ³Bloomsbury AI



Motivation

Logic-based Expert Systems

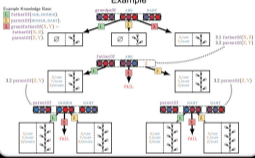
- No training data
- Interpretable
- No generalization beyond what is manually defined in rules

Representation Learning

- Behavior is learned from input-output examples
- Achieves strong generalization
- Needs a lot of training data
- Generally not interpretable

Can we get the best of both worlds?

Example



Neural Inductive Logic Programming

- Architecture allows us to induce rules of predefined structure
- We can, for instance, incorporate the inductive bias of a transitivity relationship in the knowledge base $\theta(x,y) > \theta(x,z), \theta(z,y)$
- θ are vector representations for unknown predicates
- They can be learned like all other vector representations
- They can be decoded at test time by finding the closest known relation using the RBF kernel
- Rule confidence is minimum GDP similarity over all decodings
- Confidence is an upper bound on the proof success that can be achieved when applying the induced rule

Differentiable Backward Chaining

- Neural network for proving queries to a knowledge base
- Proof success is differentiable with respect to vector representations of symbols
- Learn vector representations of symbols using SGD
- Make use of provided rules in soft proofs
- Induce interpretable first-order logic rules using SGD

Recursion

- Iterate through all rules in the knowledge base and unify goal with rule heads


 - $\text{and}(\{G, A, R\} = \{R^1 \mid R^1 \in \text{and}(\{R, A, \text{unify}(H, G, R)\} \text{ for } H = R \in \mathcal{R})$
 - Recursively prove subgoals in rule body

 - $\text{and}(\{L, \dots, \text{FAIL}\}) = \text{FAIL}$
 - $\text{and}(\{L, \dots, A, \dots\}) = \text{FAIL}$
 - $\text{and}(\{L, \dots, S, \dots\}) = S$
 - $\text{and}(\{G, A, R\} = \{R^1 \mid R^1 \in \text{and}(\{G, A, R^1\} \text{ for } R^1 \in \text{and}(\text{reduct}(G, S, L), L = S))$

Results


Knowledge Base	Metric	Model	
		CompEx	NTP
Countries	AUC-PR	96.37 ± 0.04	96.83 ± 15.4
	F1	393.89 ± 0.0	51.84 ± 1.4
S5	AUC-PR	48.44 ± 0.3	58.88 ± 17.8
	F1	77.26 ± 17.8	77.26 ± 17.8
Kinship	MRR	0.81	0.80
	F1	0.70	0.48
Nations	MRR	0.75	0.75
	F1	0.62	0.58
UMLS	MRR	0.86	0.83
	F1	0.82	0.82

Proof States and Modules



- Proof state $S = (p, g)$ is a tuple consisting of
 - S : Substitution set (variable bindings)
 - g : Neural network calculating real-valued proof success
- Modules map upstream proof state to a list of new proof states
 - Extending the substitution set (adding variable bindings)
 - Extending the neural network (adding nodes to comp. graph)

Training Objective



$NTP(\text{proof}(\{A, B, C, D\}))$

- Loss: $L_f(\theta, \{A, B, C, D\}) = -\gamma \log NTP_f(\theta, \{A, B, C, D\}) - (1 - \gamma) \log(1 - NTP_f(\theta, \{A, B, C, D\}))$
- NTP_f variant: SciA neural link prediction model (CompEx) as auxiliary task

Induced Rules

Knowledge Base: Example of induced rules per the confidence

Confidence	Rule
0.99	$\text{and}(\{A, B, C, D\}) \rightarrow \text{and}(\{A, B, C, D, E\})$
0.99	$\text{and}(\{A, B, C, D, E\}) \rightarrow \text{and}(\{A, B, C, D, E, F\})$
0.99	$\text{and}(\{A, B, C, D, E, F\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y\})$
0.99	$\text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y\}) \rightarrow \text{and}(\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\})$

Unification

- Update substitution set S_i by creating new variable bindings
- Compare vector representations of non-variables symbols using a Fossil Bana Function kernel (extending neural net S_i)

$$S_{i+1} = \left\{ \begin{array}{l} \text{and}(\{L, \dots, R\}) = S \\ \text{and}(\{L, \dots, R, A\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y\}) = \text{FAIL} \\ \text{and}(\{L, \dots, R, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}) = \text{FAIL} \end{array} \right.$$

Limitations and Future Work

- Scale to larger knowledge bases (beyond 10k facts)
 - Hierarchical attention for unification with facts
- Reinforcement learning for pruning proof tree
- Train jointly with RNNs that encode natural language statements which can then be used in proofs
- Learn to prove mathematical theorems
- Incorporate commonsense knowledge for Visual QA

Email: tim.rocktaschel@cs.ox.ac.uk s.riedel@cs.ucl.ac.uk Twitter: [@_rockt](https://twitter.com/_rockt) [@riedelcastr0](https://twitter.com/riedelcastr0)