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

- Trained end-to-end
- Strong generalization
- Needs a lot of training data

First-order Logic Expert Systems

- Rules manually defined
- No generalization
- 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
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

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
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**
- Rules manually defined
- No generalization
- 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
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

<table>
<thead>
<tr>
<th>Neural Networks</th>
<th>First-order Logic Expert Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Trained end-to-end</td>
<td>- Rules manually defined</td>
</tr>
<tr>
<td>- Strong generalization</td>
<td>- No generalization</td>
</tr>
<tr>
<td>- Needs a lot of training data</td>
<td>- No/little training data</td>
</tr>
<tr>
<td>- Not interpretable</td>
<td>- Interpretable</td>
</tr>
</tbody>
</table>

**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
## Combining Deep and Symbolic Reasoning

<table>
<thead>
<tr>
<th>Neural Networks</th>
<th>First-order Logic Expert Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Trained end-to-end</td>
<td>• No/little training data</td>
</tr>
<tr>
<td>• Strong generalization</td>
<td>• Interpretable</td>
</tr>
</tbody>
</table>

- Neural Networks: Trained end-to-end and strong generalization.
- First-order Logic Expert Systems: No/little training data and 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) . . .

- Inductive Logic Programming, e.g.,
  - 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 (Holld¨obler, 1990; Komendantskaya 2011), SHRUTI (Shastri, 1992), Neural Prolog (Ding, 1995), CLIP++ (Franca et al. 2014), Lifted Relational Networks (Sourek et al. 2015)
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.,
  - Statistical Predicate Invention (Kok and Domingos, 2007)
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.,
- 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!
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!

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

Intuition:
Backward chaining translates a query into subqueries via rules,
\texttt{grandfatherOf}(\texttt{ABE}, \texttt{BART})
\texttt{fatherOf}(\texttt{ABE}, \texttt{Z}),
\texttt{parentOf}(\texttt{Z}, \texttt{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) → fatherOf(ABE, Z), parentOf(Z, BART)
Prolog’s Backward Chaining

Example Knowledge Base:

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

Intuition:

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

- It attempts this for all rules in the knowledge base and thus specifies a depth-first search.
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
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
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)
Example Knowledge Base:

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

Query

\texttt{grandfatherOf} \ ABE \ BART

\begin{align*}
\texttt{fatherOf} & \ ABE \ HOMER & \\
\text{FAIL} & \ SUCCESS & \ FAIL
\end{align*}
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)

State $t$

∅ SUCCESS

1. fatherOf(ABE, HOMER)
   - SUCCESS

FAIL SUCCESS FAIL
Example Knowledge Base:
1. \texttt{fatherOf(ABE, HOMER)}.
2. \texttt{parentOf(HOMER, BART)}.
3. \texttt{grandfatherOf(X, Y) :- fatherOf(X, Z), parentOf(Z, Y)}.

Query

\texttt{grandfatherOf(ABE, BART)}

State \(t\)

\[\emptyset\]  SUCCESS

1. \texttt{fatherOf(ABE, HOMER)}  \[\emptyset\]  \texttt{FAIL}

State \(t + 1\)

\[\emptyset\]  FAIL

FAIL  SUCCESS  FAIL
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).

State $t$:
- $\emptyset$, SUCCESS

State $t + 1$:
- $\emptyset$, FAIL

FAIL, FAIL, SUCCESS
Example Knowledge Base:
1. \texttt{fatherOf}(\texttt{ABE}, \texttt{HOMER}).
2. \texttt{parentOf}(\texttt{HOMER}, \texttt{BART}).
3. \texttt{grandfatherOf}(X, Y) :- \\
   \texttt{fatherOf}(X, Z), \\
   \texttt{parentOf}(Z, Y).

Query
\texttt{grandfatherOf}(\texttt{ABE}, \texttt{BART}).

State \( t \)

\[ \emptyset \quad \text{SUCCESS} \]

\[ \texttt{grandfatherOf}(X, Y) \]

\[ X / \texttt{ABE} \quad Y / \texttt{BART} \quad \text{SUCCESS} \]

State \( t + 1 \)
Unification Failure

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

Query

State $t$

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>SUCCESS</td>
</tr>
</tbody>
</table>

State $t + 1$

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X/ABE</td>
<td>Y/BART</td>
</tr>
<tr>
<td>FAIL</td>
<td></td>
</tr>
</tbody>
</table>
Neural Unification

Example Knowledge Base:
1. \texttt{fatherOf}(\texttt{ABE}, \texttt{HOMER}).
2. \texttt{parentOf}(\texttt{HOMER}, \texttt{BART}).
3. \texttt{grandfatherOf}(X, Y) :-
   \texttt{fatherOf}(X, Z),
   \texttt{parentOf}(Z, Y).

Query

\texttt{grandfatherOf}(\texttt{ABE}, \texttt{BART}).

State $t$

\[ \emptyset \quad 1.0 \]

State $t + 1$

\[ X/\texttt{ABE} \quad Y/\texttt{BART} \]
Neural 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)

\[
\min\left(1.0, \exp\left(-\|v_{\text{grandpaOf}} - v_{\text{grandfatherOf}}\|_2^2 / 2\mu^2\right)\right)
\]
Differentiable Prover

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

∅ ; 1.0

grandpaOf

ABE

BART
Differentiable Prover

Example Knowledge Base:
1. fatherOf(ABE, HOMER).
2. parent0f(HOMER, BART).
3. grandfather0f(X, Y) :-
   father0f(X, Z),
   parent0f(Z, Y).

∅: 1.0
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).

∅ : 1.0

∅ : 1.0

∅ : 1.0

X/ABE
Y/BART

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).

∅ : 1.0

grandpaOf

fatherOf ABE

X/ABE

Y/BART

FAIL

FAIL

FAIL
Differentiable Prover

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

FAIL
Example Knowledge Base:
1. \texttt{fatherOf}(ABE, HOMER).
2. \texttt{parentOf}(HOMER, BART).
3. \texttt{grandfatherOf}(X, Y) :-
   \texttt{fatherOf}(X, Z),
   \texttt{parentOf}(Z, Y).
Example Knowledge Base:

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

Differentiable Prover
Differentiable Prover

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

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).

∅ : 1.0

∅ : 1.0

FAIL

FAIL

FAIL

FAIL
Example Knowledge Base:
1. fatherOf(abe, homer).
2. parentOf(homer, bart).
3. grandfatherOf(X, Y) :- fatherOf(X, Z), parentOf(Z, Y).

Differentiable Prover

∅ : 1.0

Tim Rocktäschel
End-to-End Differentiable Proving
Differentiable Prover

Example Knowledge Base:

1. fatherOf(abe, homer).
2. parentOf(homer, bart).
3. grandfatherOf(X, Y) :-
   fatherOf(X, Z),
   parentOf(Z, Y).

∞

Test Cases:

1. grandpaOf(abe, bart).
2. X/abe Y/bart Z/homer.
3.1 fatherOf(X, Z)
3.2 parentOf(Z, Y)

θ

FAIL

∞

Test Cases:

1. X/abe Y/bart Z/bart.
2. parentOf(bart, bart).
3. fatherOf(abe, Z)

FAIL

∞

Test Cases:

1. X/abe Y/bart Z/homer.
2. parentOf(homer, bart).
3. fatherOf(abe, Z)

FAIL

∞

Test Cases:

1. parentOf(bart, bart).
2. X/abe Y/bart Z/bart.
3. fatherOf(abe, Z)

FAIL

∞

Test Cases:

1. parentOf(bart, bart).
2. parentOf(homer, bart).
3. fatherOf(abe, bart)

FAIL

∞
Neural Program Induction

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

∅ 1.0

FAIL
Neural Program Induction

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

$\emptyset$ 1.0

fail 1.0

1. fatherOf 2. parentOf 3.2 $\theta_3(Z, Y)$ 3.1 $\theta_2(X, Z)$ 3.2 $\theta_3(Z, Y)$

fail 1.0

fail 1.0

fail 1.0

fail 1.0

fail 1.0
Training Objective

\[ f_{\theta}(\text{grandpaOf}(\text{abe}, \text{bart})) \]

Max pooling

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.
Training Objective

\[ f_\theta(\text{grandpaOf}(\text{abe, bart})) \]

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.
Training Objective

\[ f_\theta(\text{grandpaOf}(\text{ABE}, \text{BART})) \]

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

Accuracy / HITS@1

Countries  S3  Kinship  Nations  UMLS

ComplEx

48  70  62  82
Results

Accuracy / HITS@1

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ComplEx</th>
<th>Prover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Countries</td>
<td>48</td>
<td>57</td>
</tr>
<tr>
<td>S3</td>
<td>70</td>
<td>62</td>
</tr>
<tr>
<td>Kinship</td>
<td>48</td>
<td>62</td>
</tr>
<tr>
<td>Nations</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>UMLS</td>
<td>82</td>
<td>82</td>
</tr>
</tbody>
</table>
Results

Accuracy / HITS@1

Countries: ComplEx 48, Prover 57, Proverλ 70
S3: ComplEx 77, Prover 70, Proverλ 76
Kinship: ComplEx 48, Prover 62, Proverλ 62
Nations: ComplEx 62, Prover 59, Proverλ 82
UMLS: ComplEx 82, Prover 82, Proverλ 87
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.

Future research:
- Scaling up to larger knowledge bases.
- Connecting to RNNs for proving with natural language statements.
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
- Learn 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:

Tim Rocktäschel  End-to-End Differentiable Proving
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