# End-to-End Differentiable Proving

## Reading Notes

The NTP neuralizes itself in two parts.

### What can this NTP do?

Use knowledge from KB like a pro-log prover

Even if a KB with corrupted ground atom
e.g.,[grandpaof] is a corrupted version of [grandfatherof]

According to authors words, the 4 aims of this NTP is :

1. Neural network for proving queries to a knowledge base
2. Proof success differentiable w.r.t. vector representations of symbols
3. Learn vector representations of symbols end-to-end from proof success
4. Make use of provided rules in soft proofs
5. Induce interpretable rules end-to-end from proof success
   In one line of his own word : **Let's neuralize Prolog's Backward Chaining using a Radial Basis Function kernel for unifying vector representations of symbols!**

### What is the structure of the model?

Three modules:
Unification module-get unification representation
Then use OR and AND module which connect with backward chaining rules

### Hard concepts which I have come across when reading

To me there are 3 difficult points for understanding. The dynamic module net construction, the gradient transition at training time and the gernalizability of the method.
The first and the second points are problems about the dynamic construction at running time.
*Why the individual module net should be a static object while the net structure could change dramatically?*
Answer,though the solution structure are various, the querying parts of the module can be viewed as a static function. Therefore it can be a net.
*What is the meaning of the net which create by each module?/ What does the network learn?*
Answer, the information of the KB is encoded into the net.

The third problem is about the problem express in the experiment part.
*Is the experiment well supported to the conclusion*

Answer, yes. It provide a net model to represent a KB. The experiment support that the query into the KB can efficiently solved by it. And from the structure we can construct a multi-hop reasoning path( which means interpretable).

Besides, there are serval implementation problems need to be clarified. I try answering it after reading the incomplete code from github:

1. How to get the vector representation of a term? Is it learned from training(by autoencoder)? No, just use word2vec.
2. What is the measure of similarity between vectors? Sigmoid(L1 (A,B))

**unsolved problems after reading codes**
I can't find the parameters to train in the code.
The unify module seems never used within or/and module.
The return of module are only subsitutions.

## The most valuable idea to me

The unification module is interesting. Using vector representation for comparing is popular in CV but I never expect it in a theorem prover.
However the efficiency of OR and AND modules are questionable. Though they are differentiable, traditional methods may be more efficient and guaranteed and does no harm to create vector representation in unification module.

**Draft Notes which is helpful for me at reading time. they starts below:**

# Abstract

Object:
Knowledge Base
Methods:
dense vector representation of symbols.
replace symbolic unification with a differentiable computation on vector representations of symbols.
Results:
A network can infer facts from in incomplete knowledge base.
e.g.,
(i) place representations of similar symbols in close proximity in a vector space, (ii) make use of such similarities to prove queries, (iii) induce logical rules, and (iv) use provided and induced logical rules for multi-hop reasoning.

# Background

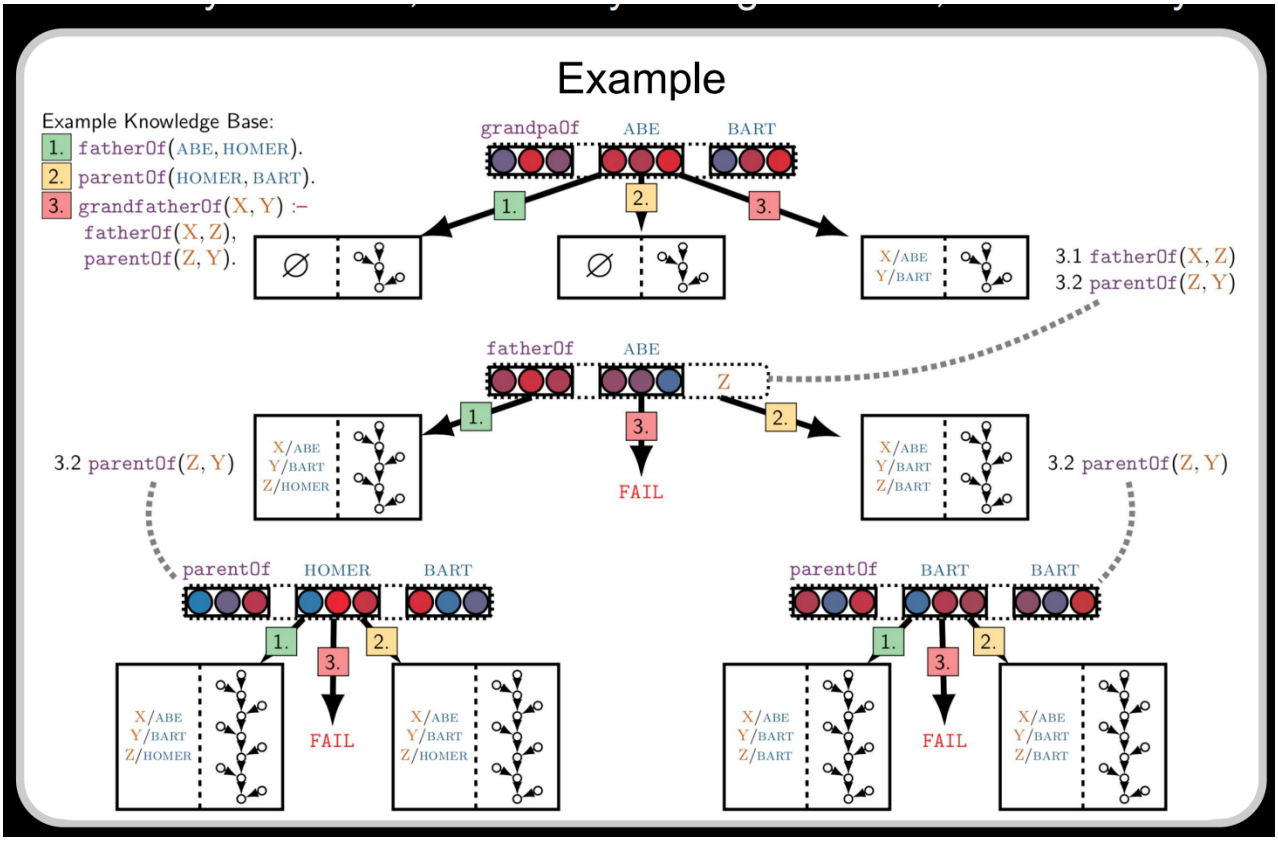
Introduction of the KB which used in this paper.
They only consider function-free first-order logic rules.

Use rule and AND OR operation to do backward chaining.

## Differentiable prover(construction the net)

End to End network
Each module takes as inputs discrete objects (atoms and rules) and a proof state, and returns a list of new proof states
A proof state S = (ψ, ρ) is a tuple consisting of the substitution set ψ constructed in the proof so far and a neural network ρ that outputs a real-valued success score of a (partial) proof.

The substitution set is kinds of alias set which give terms their alias in the proof. e.g., In the the application of module 3, X has substitution ABE




From my perspective, the net can ouput partial proof success score for all input. e.g.:

step 0:

1,2,3 are three rules in Knowledge Base
Use unification module to process the input so that it can be a unified vector

Use OR module to create query into database.
1 and 2 output S = (ψ, ρ), ψ = [], ρ = net, the input of net is the query

3 output S = (ψ, ρ), ψ = [X/ABE,Y/BART], ρ = net, the input of net is the query. This will create two new query.

step 1:

deal with new query with module

step n: no new query generate.

This paper neuralize the backward chaining by replace module with neural net.（I'll check this in code）

All symbol are in vector representation, so that it may be aware of similar symbol.

## Unification Module

if unify(A,B) not equal FAIL, proving process will continue.
This module works as a equal notation in theorem prover. It can check if the right hand side and the goal are equal. :

order matters, *i.e.*, if arguments match a line, subsequent lines are not evaluated.

1. $\texttt{unify}_{\boldsymbol{\theta}}([\,],[\,],S) = S$
2. $\texttt{unify}_{\boldsymbol{\theta}}([\,],\_,\_) = \texttt{FAIL}$
3. $\texttt{unify}_{\boldsymbol{\theta}}(\_,[\,],\_) = \texttt{FAIL}$
4. $\texttt{unify}_{\boldsymbol{\theta}}(h:\mathbf{H}, g:\mathbf{G}, S) = \texttt{unify}_{\boldsymbol{\theta}}(\mathbf{H},\mathbf{G},S') = (S'_\psi, S'_\rho)$ where

$$S'_\psi = \left\{ \begin{array}{ll} S_\psi \cup \{h/g\} & \text{if } h \in \mathcal{V} \\ S_\psi \cup \{g/h\} & \text{if } g \in \mathcal{V}, h \notin \mathcal{V} \\ S_\psi & \text{otherwise} \end{array} \right\}, \quad S'_\rho = \min\left(S_\rho, \left\{ \begin{array}{ll} \exp\left(\frac{-\|\boldsymbol{\theta}_h - \boldsymbol{\theta}_g\|_2}{2\mu^2}\right) & \text{if } h, g \notin \mathcal{V} \\ 1 & \text{otherwise} \end{array} \right\}\right)$$

## OR Module

search for rule in KB, create multiple AND Module:

1. $\texttt{or}^{\mathfrak{K}}_{\boldsymbol{\theta}}(\mathbf{G}, d, S) = [S' \mid S' \in \texttt{and}^{\mathfrak{K}}_{\boldsymbol{\theta}}(\mathbb{B}, d, \texttt{unify}_{\boldsymbol{\theta}}(\mathbf{H},\mathbf{G},S)) \text{ for } \mathbf{H} \texttt{ :- } \mathbb{B} \in \mathfrak{K}]$

## AND Module

First apply substitution:

tions to variables in an atom if possible. This is realized via

1. $\text{substitute}([\,],\_) = [\,]$
2. $\text{substitute}(g:\mathbf{G},\psi) = \left\{ \begin{array}{ll} x & \text{if } g/x \in \psi \\ g & \text{otherwise} \end{array} \right\} : \text{substitute}(\mathbf{G},\psi)$

For example, $\text{substitute}([\texttt{fatherOf}, \mathrm{X}, \mathrm{Z}], \{\mathrm{X}/\mathrm{Q}, \mathrm{Y}/i\})$ results in $[\texttt{fatherOf}, \mathrm{Q}, \mathrm{Z}]$.

Then apply AND:

This module is implemented as

1. $\text{and}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\_, \_, \texttt{FAIL}) = \texttt{FAIL}$
2. $\text{and}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\_, 0, \_) = \texttt{FAIL}$
3. $\text{and}_{\boldsymbol{\theta}}^{\mathfrak{K}}([\,], \_, S) = S$
4. $\text{and}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\mathrm{G} : \mathbb{G}, d, S) = [S'' \mid S'' \in \text{and}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\mathbb{G}, d, S') \text{ for } S' \in \text{or}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\text{substitute}(\mathrm{G}, S_\psi), d-1, S)]$

where the first two lines define the failure of a proof, either because of an upstream unification

### Success score(aggregation)

The parameterized rules are used as given rules. The vector representation of it optimized in training time too.

ill success score of proving a goal G using a

$$\text{ntp}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\mathrm{G}, d) = \underset{\substack{S \in \text{or}_{\boldsymbol{\theta}}^{\mathfrak{K}}(\mathrm{G}, d, (\varnothing, 1)) \\ S \neq \texttt{FAIL}}}{\arg\max} S_\rho$$

imum proof depth and the initial proof state i

## Optimization(How to pass the gradient)

There two kinds of loss, basic loss and auxiliary loss.
The corrupted atoms are generated by resampling at training time.

### Objective-Basic loss

A typical log likelihood loss:

$$\mathcal{L}_{\text{ntp}_{\boldsymbol{\theta}}^{\mathfrak{K}}} = \sum_{([s,i,j],y) \in \mathcal{T}} -y \log(\text{ntp}_{\boldsymbol{\theta}}^{\mathfrak{K}}([s,i,j],d)_\rho) - (1-y)\log(1 - \text{ntp}_{\boldsymbol{\theta}}^{\mathfrak{K}}([s,i,j],d)_\rho)$$

The unification success score is set to zero to avoid early abort which resulting in no parameter updates.
All training atoms are ground atoms.

### Neural link prediction-Auxiliary loss

> At the beginning of training all subsymbolic representations are initialized randomly. When unifying a goal with all facts in a KB we consequently get very noisy success scores in early stages of training.

This is the reason for Auxiliary Loss in training time.
The form of the Auxiliary Loss is similar to formalization.

$$\mathcal{L}_{\text{ntp}\lambda_{\boldsymbol{\theta}}^{\mathfrak{K}}} = \mathcal{L}_{\text{ntp}_{\boldsymbol{\theta}}^{\mathfrak{K}}} + \sum_{([s,i,j],y) \in \mathcal{T}} -y \log(\text{complex}_{\boldsymbol{\theta}}(s,i,j)) - (1-y)\log(1 - \text{complex}_{\boldsymbol{\theta}}(s,i,j))$$

### limitation on optimization

The limitation is the neural network is used to representing the KB and the prover module. In this paper, the NTP have no early abort except unification abort.

> NTPs as described above suffer from severe computational limitations since the neural network is representing all possible proofs up to some predefined depth. In contrast to symbolic backward

## Optimization methods

### Batch proving

batch version unification model

### K-max Gradient Approximation

in order to deal with the lack of gradient problem from max function, a K max heuristic method for gradient computation is proposed.
Only K unification successes are keeped after unification process.