Deep Learning and Logic

…

William W Cohen

Google AI/Carnegie Mellon University

joint work with

Fan Yang, Zhilin Yang, Kathryn Rivard Mazaitis

Clean
understandable
elegant models

Complexity of
real-world
phenomena

⇒

complex models

⇒

lots of programming or data

How did we get here?

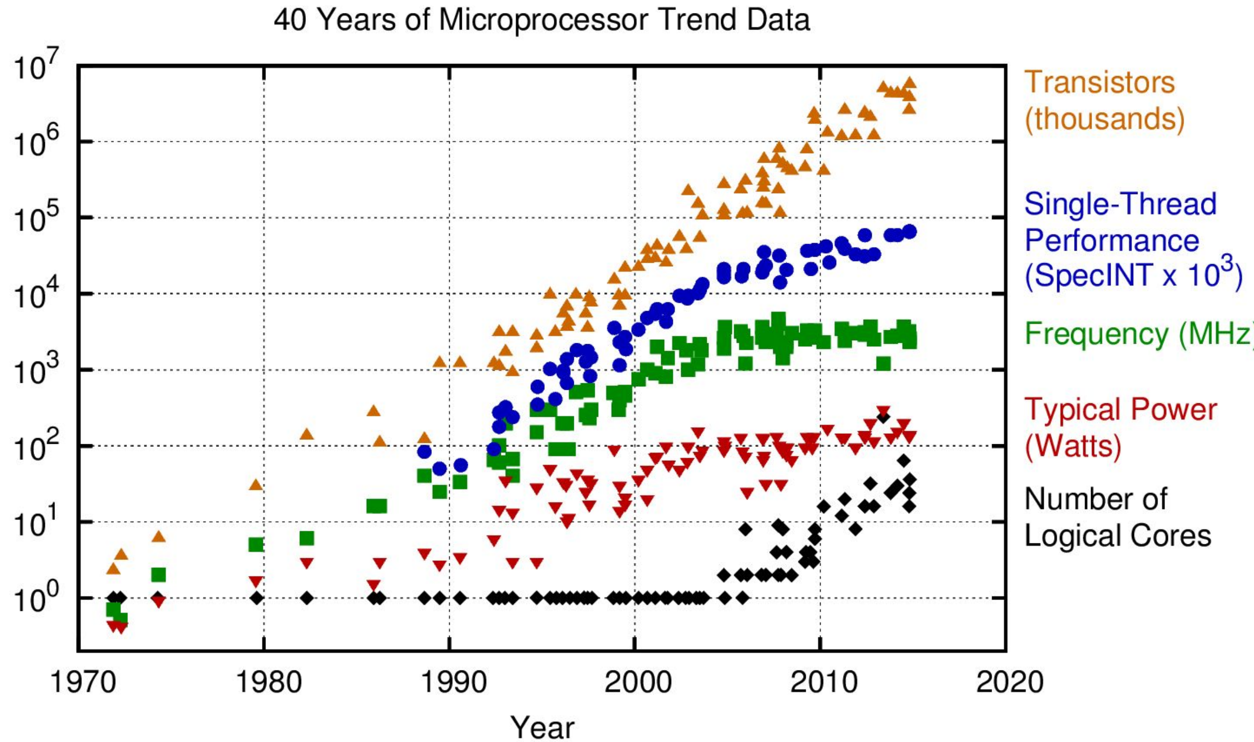
Complexity of
real-world
phenomena

⇒

complex models

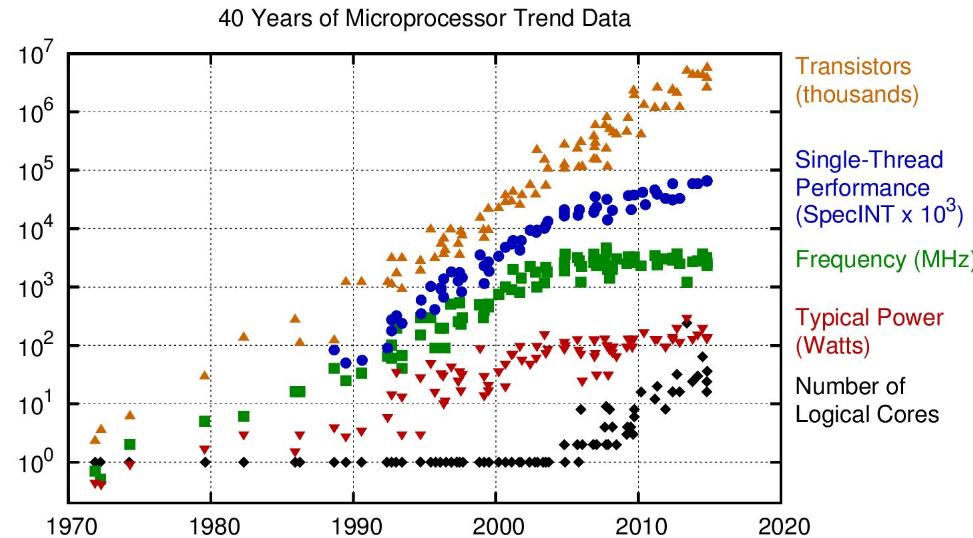
⇒

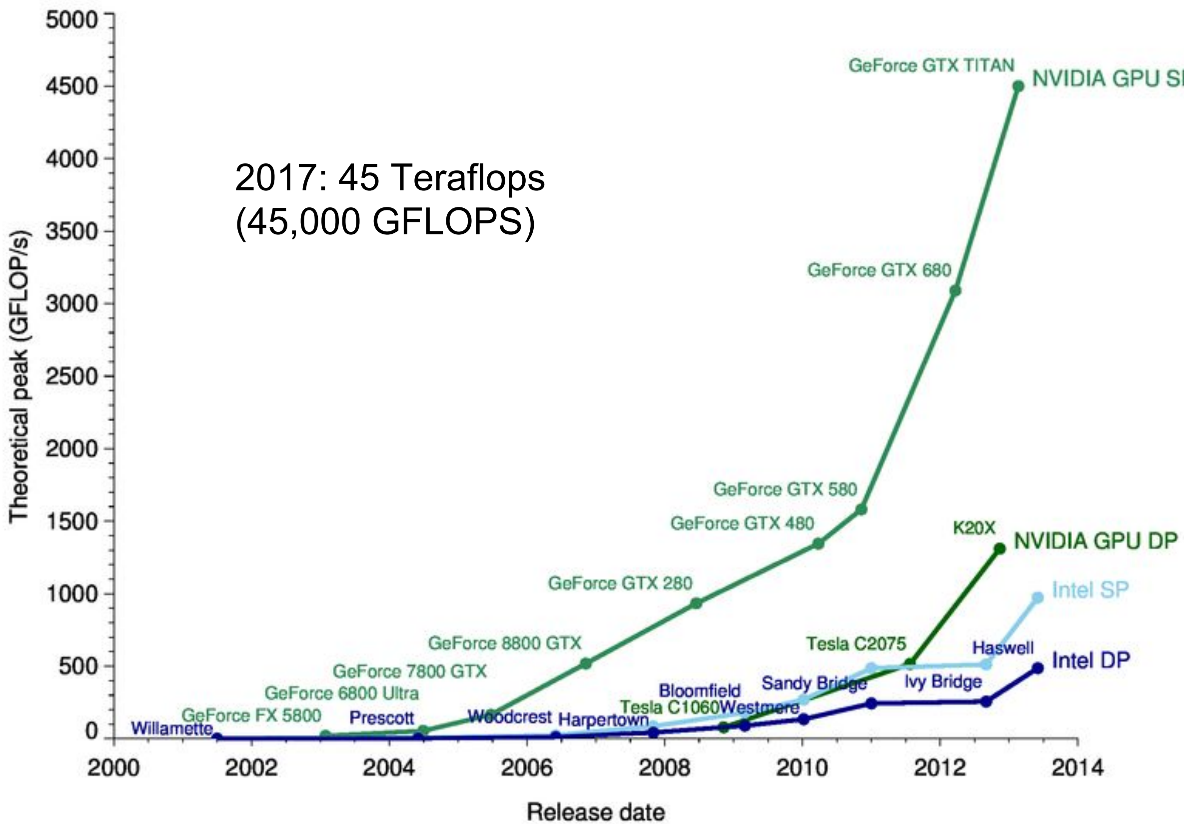
lots of programming or data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

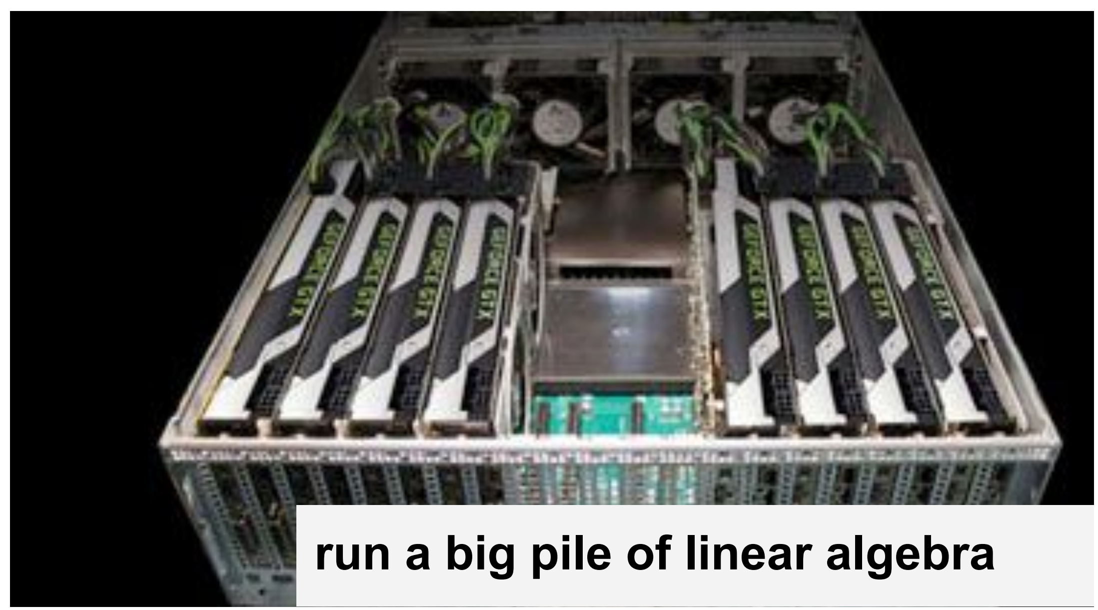
How did we get here?





2017: 45 Teraflops
(45,000 GFLOPS)

How did we get here?



Deep Learning and Logic:

Learnable Probabilistic Logics That Run On GPUs

Tensorlog:
Key ideas and background

Probabilistic Deductive DBs

```
1. uncle(X,Y):-child(X,W),brother(W,Y).
2. uncle(X,Y):-aunt(X,W),husband(W,Y).
3. status(X,tired):-child(W,X),infant(W).
```

child(liam,eve)	0.99
child(dave,eve)	0.99
child(liam,bob)	0.75
husband(eve,bob)	0.9
infant(liam)	0.7
infant(dave)	0.1
aunt(joe,eve)	0.9
brother(eve,chip)	0.9

Horn clauses (rules)

ground unit clauses (facts)

weight for each fact

Probabilistic Deductive DBs

1. uncle(X,Y):-child(X,W),brother(W,Y).
2. uncle(X,Y):-aunt(X,W),husband(W,Y).
3. ~~status(X,tired):-child(W,X),infant(W).~~

status(X,tired) :- child(W,X), infant(W), **weighted(r3).**

child(liam,eve)	0.99
child(dave,eve)	0.99
child(liam,bob)	0.75
husband(eve,bob)	0.9
infant(liam)	0.7
infant(dave)	0.1
aunt(joe,eve)	0.9
brother(eve,chip)	0.9
weighted(r3)	**0.98**

We use this trick to weight rules

special fact *only* appearing in *this* rule

Probabilistic Deductive KGs (Knowledge Graphs)

```
1. uncle(X,Y):-child(X,W),brother(W,Y).
2. uncle(X,Y):-aunt(X,W),husband(W,Y).
3. status(X,tired):-child(W,X),infant(W).
```

child(liam,eve)	0.99
child(dave,eve)	0.99
child(liam,bob)	0.75
husband(eve,bob)	0.9
infant(liam)	0.7
infant(dave)	0.1
aunt(joe,eve)	0.9
brother(eve,chip)	0.9

Assumptions:

- (*Only* parameters are weights for facts)
- Predicates are **unary or binary**
- Rules have **no function symbols** or constants

Neural implementations of logic

```
uncle(X,Y):-child(X,W),brother(W,Y).
uncle(X,Y):-aunt(X,W),husband(W,Y).
status(X,tired):-child(W,X),infant(W).
```

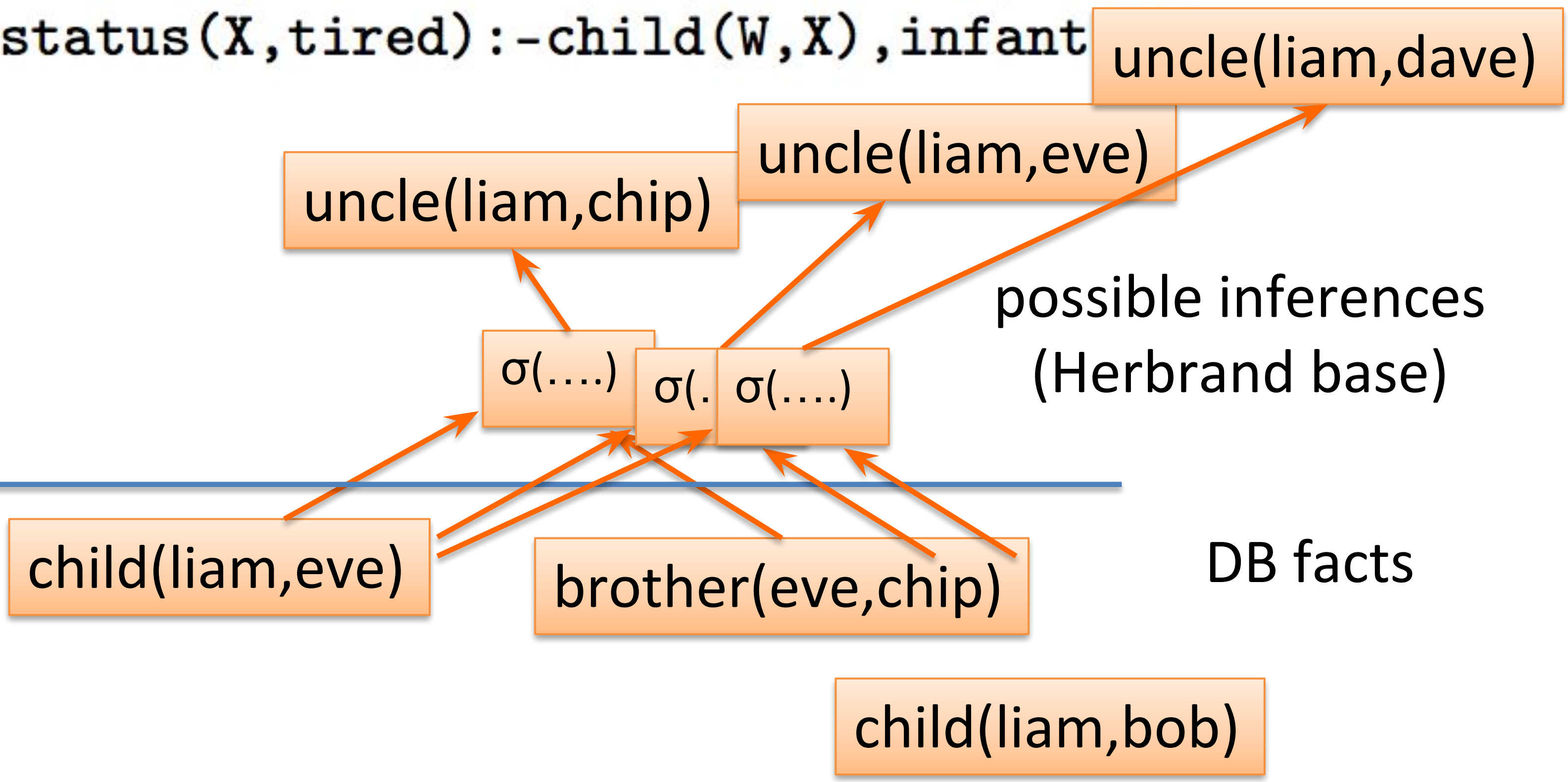
KBANN idea (1991): convert every DB fact, and every possible inferable fact, to a neuron.

Similar "grounding strategies" are used by many other soft logics: Markov Logic Networks, Probabilistic Soft Logic, …

A neuron for every *possible* inferable fact is "too many" --- i.e., bigger than the DB.

Reasoning in PrDDBs/PrDKGs

```
uncle(X,Y):-child(X,W),brother(W,Y).
uncle(X,Y):-aunt(X,W),husband(W,Y).
status(X,tired):-child(W,X),infant
```



usual approach: “grounding” the rules

Reasoning in PrDDBs/PrDKGs

explicit grounding does not scale!

```
uncle(X,Y):-child(X,W),brother(W,Y).
uncle(X,Y):-aunt(X,W),husband(W,Y).
status(X,tired):-child(W,X),infant(W).
```

Example: inferring family relations like "uncle"

- N people
- N^2 possible "uncle" inferences

- N = 2 billion → N^2 = 4 quintillion
- N = 1 million → N^2 = 1 trillion

A KB with 1M entities is *small*

Reasoning in TensorLog

- TensorLog uses a knowledge-graph specific trick to get scalability:
  - "reasoning" means answering a query like*: find all Y for which p(a,Y) is true* for some given predicate *p;*query entity *a;* and theory T and KG)
  - inferences for a logical theory can be encoded as a bunch of **functions**: for every *p, a,* a vector ***a*** encodes *a*, *and* the function $f_p(\boldsymbol{a})$ returns a **vector** encoding answers *y* (and confidences)
  -
  - actually we have functions for *p(a,Y)* and *p(Y,a)*…. called $f_{p:io}(a)$ and $f_{p:oi}(a)$

Reasoning in TensorLog

```
uncle(X,Y):-child(X,W),brother(W,Y).
uncle(X,Y):-aunt(X,W),husband(W,Y).
status(X,tired):-child(W,X),infant(W).
```

Example: inferring family relations like "uncle"

- N people
- N^2 possible "uncle" facts
- ~~N = 1 million → N^2 = 1 trillion~~

The vectors are size O(N) not O(N^2)

x is the nephew

x is the uncle

$f_1(\mathbf{x}) = \mathbf{Y}$

$f_2(\mathbf{x}) = \mathbf{Y}$

one-hot vectors

(0,0,0,**1**,0,0,0)

(0,0,**0.81**,0,0,**0.93**,0,0,0)

vectors encoding weighted **set** of DB instances

Reasoning in TensorLog

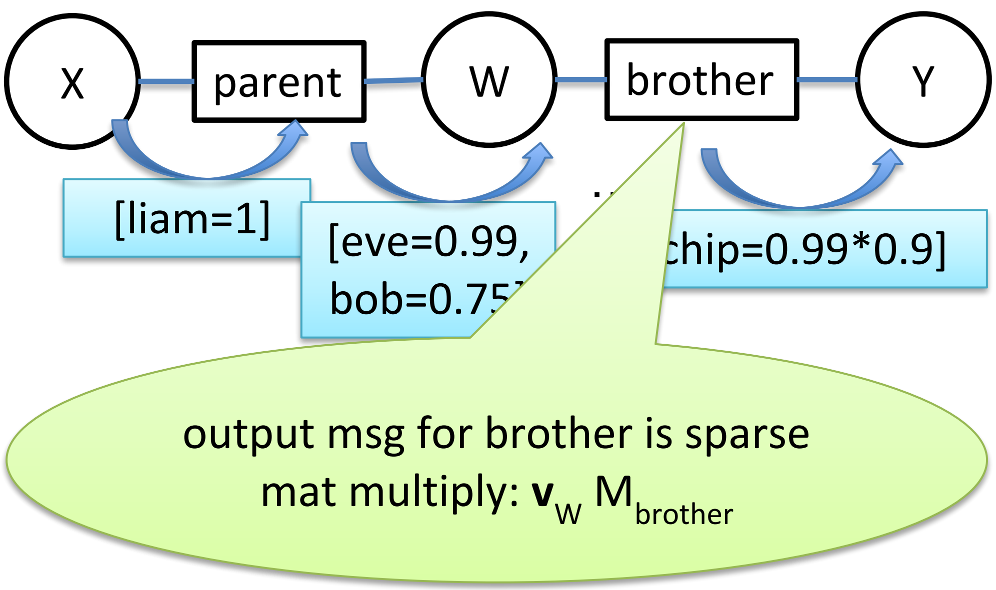
- TensorLog uses a knowledge-graph specific trick…functions from sets of entities to sets of entities
- Key idea: You can describe the reasoning process as a *factor graph*
- Example: Let’s start with some example one-rule theories

Reasoning via message-passing: example

```
child(liam,eve),0.99     infant(liam),0.7
child(dave,eve),0.99     infant(dave),0.1
child(liam,bob),0.75     aunt(joe,eve),0.9
husband(eve,bob),0.9     brother(eve,chip),0.9
```

Query: uncle(liam, Y) ?

uncle(X,Y):-parent(X,W),brother(W,Y)



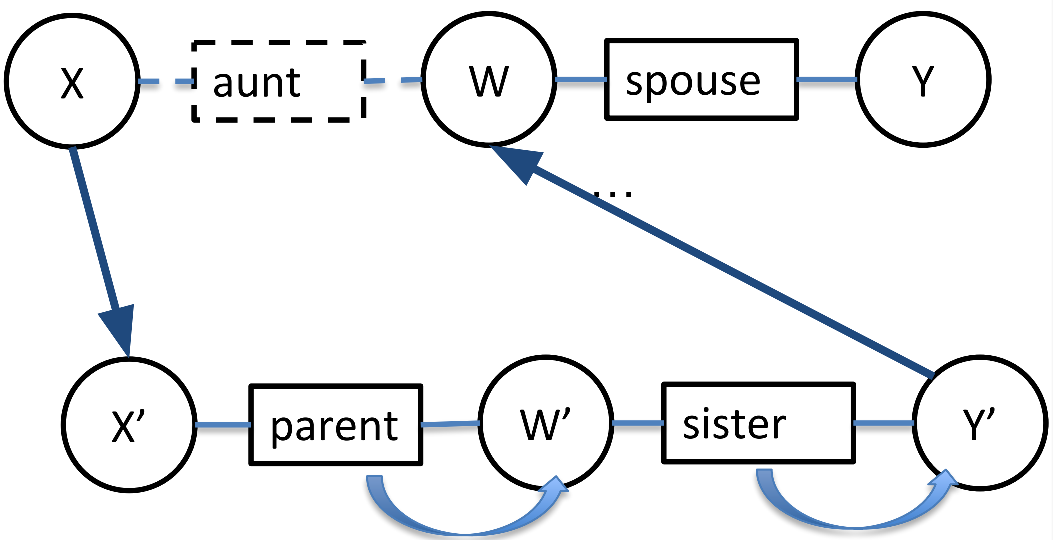
- Algorithm: build a **factor graph** with one **random variable** for each **logical variable**, encoding a distribution over DB constants, and one **factor** for each logical **literal**.
- **Belief propagation** on factor graph enforces the logical constraints of a proof, and gives a **weighted count** of number of proofs supporting each answer

Reasoning via message-passing: subpredicates

```
child(liam,eve),0.99      infant(liam),0.7
child(dave,eve),0.99      infant(dave),0.1
child(liam,bob),0.75      aunt(joe,eve),0.9
husband(eve,bob),0.9      brother(eve,chip),0.9
```

Query: uncle(liam, Y) ?

uncle(X,Y):-aunt(X,W),spouse(W,Y)
aunt(X,Y):-parent(X,W),sister(W,Y)



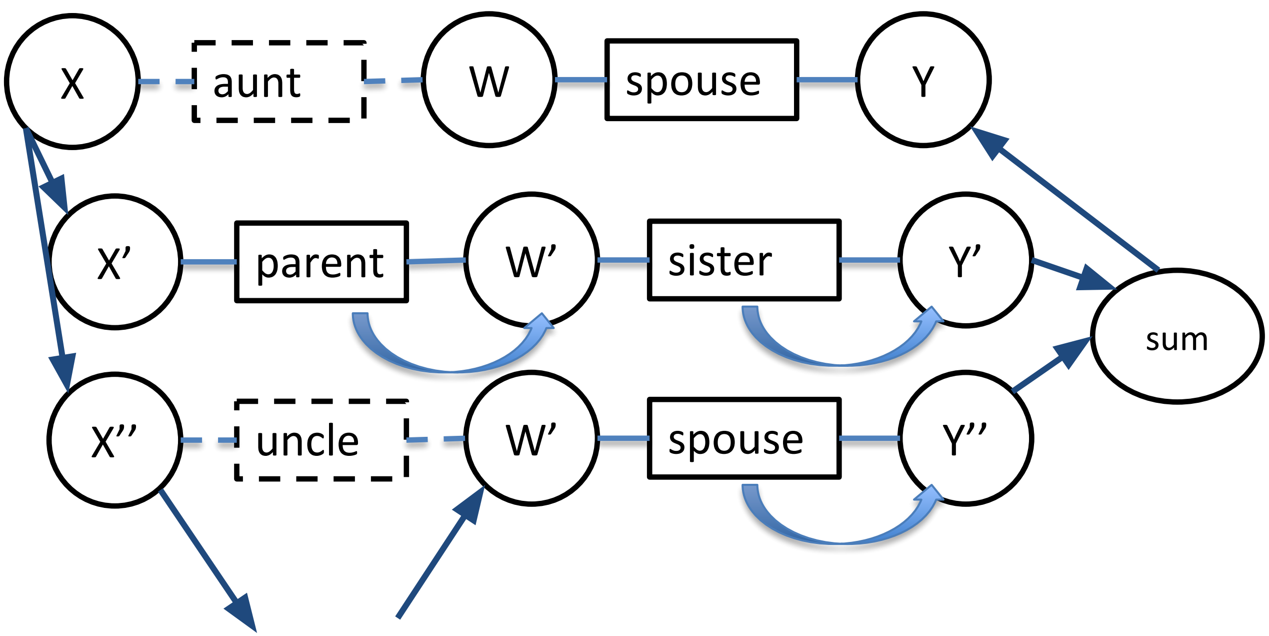
- Recursive predicate calls can be expanded in place in the factor graph
- Stop at a fixed maximum depth (and return count of zero proofs)

Reasoning via message-passing: subpredicates

Query: uncle(liam, Y) ?

```
child(liam,eve),0.99     infant(liam),0.7
child(dave,eve),0.99     infant(dave),0.1
child(liam,bob),0.75     aunt(joe,eve),0.9
husband(eve,bob),0.9     brother(eve,chip),0.9
```

uncle(X,Y):-aunt(X,W),spouse(W,Y)
aunt(X,Y):-parent(X,W),sister(W,Y)



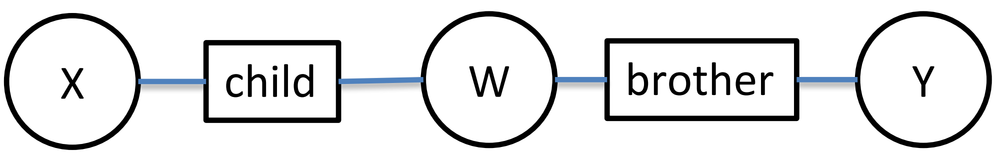
- Recursive predicate calls can be expanded in place in the factor graph
- **Multiple** clauses for the same predicate: **add** the proof counts for each clause

Reasoning via message-passing: key ideas

`child(liam,eve)`,0.99	`infant(liam)`,0.7
`child(dave,eve)`,0.99	`infant(dave)`,0.1
`child(liam,bob)`,0.75	`aunt(joe,eve)`,0.9
`husband(eve,bob)`,0.9	`brother(eve,chip)`,0.9

Query: uncle(liam, Y) ?

uncle(X,Y):-child(X,W),brother(W,Y)



General case for p(c,Y):

- initialize the evidence variable X to a one-hot vector for c
- wait for BP to converge
- read off the message **y** that would be sent from the output variable Y.
    - un-normalized probability
- **y**[d] is the **weighted number of proofs supporting** p(c,d)

Reasoning via message-passing: key ideas

```
child(liam,eve),0.99    infant(liam),0.7
child(dave,eve),0.99    infant(dave),0.1
child(liam,bob),0.75    aunt(joe,eve),0.9
husband(eve,bob),0.9    brother(eve,chip),0.9
```

Special case:

- If all clauses are *polytrees* (~= every free variable has one path of dependences linking it to a bound variable) then BP converges in *linear time* and will result in a *fixed sequence of messages* being passed
- Only a few linear algebra operators are used in these messages:
  - vector-matrix multiplication
  - Hadamard product
  - multiply **v1** by L1 norm of **v2**
  - vector sum
  - (normalization)

Rule	r1: uncle(X,Y):- parent(X,W), brother(W,Y)	r2: uncle(X,Y):- aunt(X,W), husband(W,Y)	r3: status(X,T):- assign_tired(T), parent(X,W), infant(W),any(T,W)
Function	$g^{r1}_{\texttt{io}}(\vec{u}_c)$	$g^{r2}_{\texttt{io}}(\vec{u}_c)$	$g^{r3}_{\texttt{io}}(\vec{u}_c)$
Operation sequence defining function	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{parent}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{brother}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{aunt}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{husband}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$	$\mathbf{v}_{2,W} = \mathbf{u}_c\mathbf{M}$ $\mathbf{v}_{3,W} = \mathbf{v}_{\texttt{infant}}$ $\mathbf{W} = \mathbf{v}_{2,W} \circ \mathbf{v}_{3,W}$ $\mathbf{v}_{1,T} = \mathbf{v}_{\texttt{assign\_tired}}$ $\mathbf{v}_{4,T} = \mathbf{v}_W\mathbf{M}_{\texttt{any}}$ $\mathbf{T} = \mathbf{v}_{1,T} \circ \mathbf{v}_{4,T}$
Returns	$\mathbf{v}_Y$	$\mathbf{v}_Y$	$\mathbf{v}_T$

The result of this message-passing sequence produced by BP is just a function:
the function $f_{p:io}$(***a***) we were trying to construct!

Note on Semantics

The semantics are **proof-counting**, not model-counting: conceptually

- For each answer *a* to query *Q,* find all derivations d_a that prove *a*
- The weight of each d_a is product of weight w_f of each KG fact *f* used in that derivation
- The weight of *a* is the sum of the weights of all derivations

This is an **unnormalized stochastic logic program** (SLP) - Cussens and Muggleton, with weights computed efficiently (for this special case) by dynamic programming (even with exponentially many derivations)

Note on Semantics

Compare to **model-counting** where conceptually

- There is a distribution Pr(KG) over KGs
  - Tuple-independence: draw a KG by picking each fact f with probability w_f
- The probability of a fact f' is the probability T+KG' implies f', for a KG' is drawn from Pr(KG)

E.g.: ProbLog, Fuhr's Probabilistic Datalog (PD), ...

Tensorlog:
Learning Algorithms

Learning in TensorLog

Inference is now via a numeric function: $\mathbf{y} = g_{io}^{uncle}(\mathbf{u}_a)$

$\mathbf{y}$ encodes *{b:uncle(a,b)}* is true and **y**[b]=confidence in uncle(a,b)

Define *loss function* relative to target proof-count values **y*** for **x**, eg

$\text{loss}(g_{io}^{uncle}(\mathbf{u}_a), \mathbf{y^*}) = \text{crossEntropy}(\text{softmax}(g(\mathbf{x})),\mathbf{y^*})$

Minimize the loss with gradient-descent, ….

- To adjust weights for selected DB relations, e.g.: $\text{dloss}/\text{d}M_{brother}$

Key point: Learning is "**free**" in TensorLog

Inference is now via a numeric function: $\mathbf{y} = g_{io}^{uncle}(\mathbf{u}_a)$

y encodes *{b:uncle(a,b)}* is true and **y**[b]=confidence in uncle(a,b)

Define *loss function* relative to target proof-count values **y*** for **x**, eg

$loss(g_{io}^{uncle}(\mathbf{u}_a), \mathbf{y^*}) = crossEntropy(softmax(g(\mathbf{x})),\mathbf{y^*})$

Minimize the loss with gradient-descent, ...

- To adjust weights for selected DB relations, e.g.: $dloss/dM_{brother}$

- **Homegrown** implementation: SciPy implementation of operations, derivatives, and gradient descent optimization
- **Compilation** to TensorFlow expressions ⇒ TF derivatives, optimizers, ...

Tensorlog:
Experimental Results

Experiment: factual Q/A from a KB

WikiMovies dataset

who acted in the movie Wise Guys? ['Harvey Keitel', 'Danny DeVito', 'Joe Piscopo', …]
what is a film written by Luke Ricci? ['How to Be a Serial Killer']
…

Data: from Miller, Fisch, Dodge, Karami, Bordes, Weston "Key-Value Memory Networks for Directly Reading Documents"

- Questions: 96k train, 20k dev, 10k test
  Knowledge graph: 421k triples about 16k movies, 10 relations

- Subgraph/question embedding:
  - 93.5%
- Key-value memory network:
  - 93.9% "reading" the KG
  - 76.2% by reading text of articles

starred_actors	Wise Guys	Harvey Keitel
starred_actors	Wise Guys	Danny DeVito
starred_actors	Wise Guys	Joe Piscopo
starred_actors	Wise Guys	Ray Sharkey
directed_by	Wise Guys	Brian De Palma
has_genre	Wise Guys	Comedy
release_year	Wise Guys	1986
…		

TensorLog model

relations in DB = 9

who acted in the movie Wise Guys? ['Harvey Keitel', 'Danny DeVito', 'Joe Piscopo', …]
what is a film written by Luke Ricci? ['How to Be a Serial Killer']
…

```
answer(Question, Entity) :-
  mentions_entity(Question,Movie),
  starred_actors(Movie,Entity),
   feature(Question,F),weight_sa_io(F).
  % w_sa_f: weight for  starred_actors(i,o)
...
answer(Question, Movie) :-
  mentions_entity(Question,Entity),
  written_by(Movie,Entity),
  feature(Question,F),weight_wb_oi(F).
...
Total: 18 rules
```

starred_actors	Wise Guys	Harvey Keitel
starred_actors	Wise Guys	Danny DeVito
starred_actors	Wise Guys	Joe Piscopo
starred_actors	Wise Guys	Ray Sharkey
directed_by	Wise Guys	Brian De Palma
has_genre	Wise Guys	Comedy
release_year	Wise Guys	1986
…		
written_by	How to .. Killer	Luke Ricci
has_genre	How to .. Killer	Comedy
...		

TensorLog model

k = # relations in DB = 9

who acted in the movie Wise Guys? ['Harvey Keitel', 'Danny DeVito', 'Joe Piscopo', …]
what is a film written by Luke Ricci? ['How to Be a Serial Killer']
…

```
answer(Question, Entity) :-
  mentions_entity(Question,Movie),
  starred_actors(Movie,Entity),
  feature(Question,F),weight_sa_io(F).
  % w_sa_f: weight for  starred_actors(i,o)
...
answer(Question, Movie) :-
  mentions_entity(Question,Entity),
  written_by(Movie,Entity),
  feature(Question,F),weight_wb_oi(F).
...
Total: 18 rules
```

These weights are a linear classifier that says *which* rule to use to answer *which question*

Experiment: Factual Q/A with a KB

Method	Accuracy	Time per epoch
Subgraph/question embedding	93.5%	
Key-value memory network	93.9%	
TensorLog (1,000 training examples)	89.4%	6.1 sec
TensorLog (10,000 training examples)	94.8%	1.7 min
TensorLog (96,182 training examples)	**95.0%**	49.5 min

- KG is about 420k movie facts + 850k facts about the questions (mentions_entity/2, features/2)

Joint entity-linking and QA

proposed extension

```
answer(Question,Answer) :-
    classification(Question,aboutActedIn),
    mentionsEntity(Question,Entity), actedIn(Answer,Entity).
answer(Question,Answer) :-
    classification(Question,aboutDirected),
    mentionsEntity(Question,Entity), directed(Answer,Entity).
answer(Question,Answer) :-
    classification(Question,aboutProduced),
    mentionsEntity(Question,Entity), produced(Answer,Entity).
...
mentionsEntity(Question,Entity) :-
    containsNGram(Question,NGram), matches(NGram,Name),
    possibleName(Entity,Name), popular(Entity).
classification(Question,Y) :-
    containsNGram(Question,NGram), indicatesLabel(NGram,Y).
matches(NGram,Name) :-
    containsWord(NGram,Word), containsWord(Name,Word), important(Word).
```

Experiment: Relational Learning Benchmarks

Task	ProPPR	TensorLog
CORA (13k facts, 10 rules)	AUC-ROC 83.2	AUC-ROC **97.6**
UMLS (5k facts, 226 rules)	Accuracy 49.8	Accuracy **52.5**
Wordnet (276k facts)		
Hypernym (46 rules)	Accuracy **93.4**	Accuracy 93.3
Hyponym (46 rules)	Accuracy 92.1	Accuracy **92.8**

Theories all learned using ISG (Wang et al, CIKM 2014) and then fixed

Experiment: Scalability of Inference

Friends and Smokers			Grid Transitive Closure		
Name	# Entities	# Facts	Name	# Entities	# Facts
FS100	400	5,060	GR10	100	784
FS1k	4,000	48,260	GR25	625	5,329
FS10k	40,000	480,260	GR50	2,500	21,904
FS100k	400,000	4800,260	GR100	10,000	88,804
FS500K	2,000,000	24,000,260	GR200	40,000	357,604

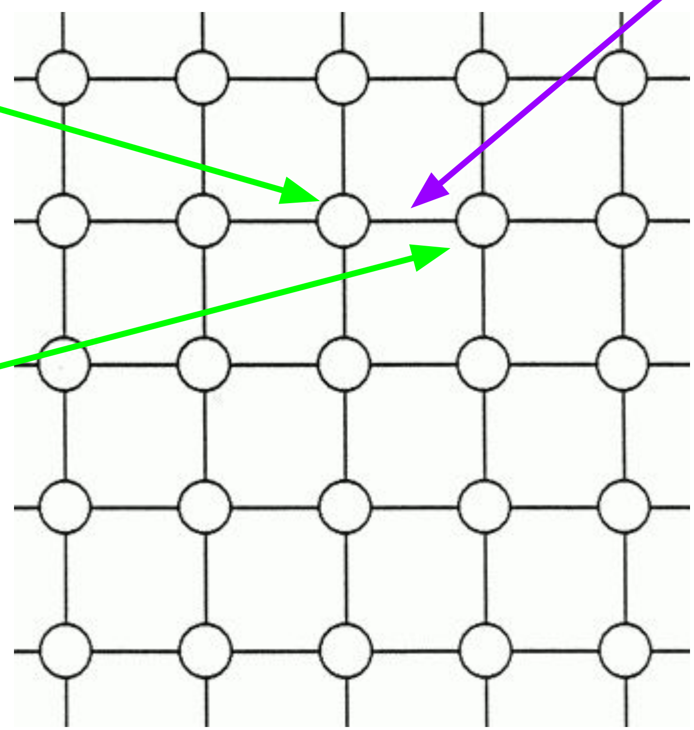
shallow inference task

deeply recursive inference task

Experiment: Scalability of Inference

edge(cell_2_3, cell_2_4) 0.2
...

cell_2_3



cell_2_4

path(X,Y) :- edge(X,Y)
path(X,Z) :- edge(X,Z),path(Z,Y)

Experiment: Scalability of Inference

Graph	ProPPR	SciPy			Tensorflow CPU		Tensorflow GPU	
		none	b=25	b=250	b=25	b=250	b=25	b=250
FS100	1392.8	73.08						
FS1k	1310.6	71.40						
FS10k	*1190.8*	68.39						
FS100k	*236.1*	33.19						
FS500k	*178.4*	12.16						
GR10	43.1	75.1						
GR25	83.8	68.8						
GR50	108.1	47.2						
GR100	117.3	11.3						
GR200	*116.6*	0.9						

shallow

recursive

- Queries per second: machine with one GPU
  - eg on query -? path(cell_2_4,Y)
- bold is best TensorLog performer - ProPPR italicized if it “wins”

Experiment: Scalability of Inference

Graph	ProPPR	SciPy			Tensorflow CPU		Tensorflow GPU		
		none	b=25	b=250	b=25	b=250	b=25	b=250	
FS100	1392.8	73.08	1247.64	–					shallow
FS1k	1310.6	71.40	1183.65	**3635.62**					
FS10k	*1190.8*	68.39	551.53	**907.64**					
FS100k	*236.1*	33.19	93.33	**99.44**					
FS500k	*178.4*	12.16	16.72	**17.10**					
GR10	43.1	75.1	**567.6**	–					recursive
GR25	83.8	68.8	325.8	1264.2					
GR50	108.1	47.2	99.4	466.0					
GR100	117.3	11.3	12.3	108.6					
GR200	*116.6*	0.9	1.0	8.5					

- Queries per second: machine with one GPU
- bold/italics is best performer
- b=25 means that 25 queries are done in **parallel** (as a “minibatch”)
- minibatch paralellization gives large - up to 10x - speedup on **one** core

Experiment: Scalability of Inference

Graph	ProPPR	SciPy			Tensorflow CPU		Tensorflow GPU	
		none	b=25	b=250	b=25	b=250	b=25	b=250
FS100	1392.8	73.08	1247.64	–	202.29	–		
FS1k	1310.6	71.40	1183.65	**3635.62**	143.34	926.22		
FS10k	*1190.8*	68.39	551.53	**907.64**	44.34	237.26		
FS100k	*236.1*	33.19	93.33	**99.44**	5.32	24.81		
FS500k	*178.4*	12.16	16.72	**17.10**	–	–		
GR10	43.1	75.1	**567.6**	–	250.3	–		
GR25	83.8	68.8	325.8	1264.2	174.9	1159.4		
GR50	108.1	47.2	99.4	466.0	67.9	466.8		
GR100	117.3	11.3	12.3	108.6	10.1	88.2		
GR200	*116.6*	0.9	1.0	8.5	0.89	7.65		

shallow

recursive

- Queries per second: machine with one GPU
- bold/italics is best performer
- b=25 means that 25 queries are done in parallel (as a "minibatch")
- Compared **TensorFlow** and **homegrown** sparse matrix backends ...

Experiment: Scalability of Inference

Graph	ProPPR	SciPy			Tensorflow CPU		Tensorflow GPU		
		none	b=25	b=250	b=25	b=250	b=25	b=250	
FS100	1392.8	73.08	1247.64	–	202.29	–	452.53	–	shallow
FS1k	1310.6	71.40	1183.65	**3635.62**	143.34	926.22	198.99	1552.55	
FS10k	*1190.8*	68.39	551.53	**907.64**	44.34	237.26	67.95	314.10	
FS100k	*236.1*	33.19	93.33	**99.44**	5.32	24.81	11.06	37.72	
FS500k	*178.4*	12.16	16.72	**17.10**	–	–	–	–	
GR10	43.1	75.1	**567.6**	–	250.3	–	204.8	–	recursive
GR25	83.8	68.8	325.8	1264.2	174.9	1159.4	187.9	**1826.5**	
GR50	108.1	47.2	99.4	466.0	67.9	466.8	85.5	**872.8**	
GR100	117.3	11.3	12.3	108.6	10.1	88.2	19.3	**191.2**	
GR200	*116.6*	0.9	1.0	8.5	0.89	7.65	1.6	**16.4**	

- Queries per second: machine with one GPU (Titan X, 12Gb)
- bold/italics is best performer
- b=25 means that 25 queries are done in **parallel** (as a "minibatch")
- Tested TensorFlow and hand-constructed sparse matrix backends
- Tested TensorFlow with GPU: **only 1.5-2x faster for inference** and then only on deeper models

Experiment: Scalability of Learning

Graph	SciPy		Tensorflow CPU		Tensorflow GPU		GPU Speedup	
	Time	Acc	Time	Acc	Time	Acc	*vs* SciPy	*vs* CPU
GR10	11.6	0.90	1.23	**0.85**	**0.97**	0.80	12.0	1.3
GR25	2544.7	0.98	24.88	**1.00**	**4.77**	**1.00**	533.3	5.2
GR50	>10000	–	296.0	0.95	**30.7**	**0.97**	–	9.6
GR100	>10000	–	3203.6	0.98	**392.9**	**1.00**	–	8.2
GR200	>10000	–	–	–	–	–	–	–

- Task: learn grid transition weights so that transitive closure operations perform a particular navigational goal
  - Go from cell to closest “landmark” cell, like (10,10) or (30,50)
- Minibatch size of 25
- A 25 by 25 grid
- Learning is ***much faster*** with TensorFlow and with GPUs
  - Architected for learning/repeated passes over data with same code

Experiment: Robustness of Learning

Grid Size	Max Depth	# Graph Nodes		Acc	
		SciPy	TF	SciPy	TF
16	10	68	2696	**99.9**	97.2
18	12	80	3164	93.9	**96.9**
20	14	92	3632	25.2	**99.1**
22	16	104	4100	8.6	**98.4**
24	18	116	4568	**2.4**	0.0

- Tune parameters on 16x16 grid task
- Run same parameters on larger grids (deeper inference, different architecture networks)
- Compare **homegrown gradient descent** and well-tuned Adagrad (Tensorflow implementation)

Adagrad is more robust and faster

Tensorlog: Extensions

Experiment: Learning Other Semantics

Inference is now via a numeric function: $\mathbf{y} = g_{io}^{uncle}(\mathbf{u}_a)$

$\mathbf{y}$ encodes *{b:uncle(a,b)}* is true and $\mathbf{y}$[b]=confidence in uncle(a,b)

Define *loss function* relative to target proof-count values $\mathbf{y}^*$ for $\mathbf{x}$, eg

$\text{loss}(g_{io}^{uncle}(\mathbf{u}_a), \mathbf{y}^*) = \text{crossEntropy}(\text{softmax}(g(\mathbf{x})),\mathbf{y}^*)$

softmax *normalizes* the proof counts **y**
so you learn a *conditional* distribution P(**y**|**x**)

- i.e. sum of **y**’s will be 1.0
- can rank people by confidence in being “Bob’s uncle” but can’t tell *how many uncles* Bob has

(but it’s great to optimize!)

Key point: flexibility is **free**

Adding logistic regression "on top" of TensorLog

Inference is now via a numeric function: $\mathbf{y} = g_{io}^{uncle}(\mathbf{u}_a)$

y encodes *{b:uncle(a,b)}* is true and **y**[b]=confidence in uncle(a,b)

Define *loss function* relative to target proof-count values **y*** for **x**, eg

$loss(g_{io}^{uncle}(\mathbf{u}_a), \mathbf{y^*}) = crossEntropy(sigmoid(g(\mathbf{x}) + \mathbf{b}), \mathbf{y^*})$

Alternative: convert weighted proofcounts to an *arbitrary distribution* - e.g. with a biased sigmoid - and assess loss relative to that. *Loss function changes, learning still "free".*

Then you can learn to match an arbitrary target distribution.

Example: alternative semantics

Recall proof-counting was compared to **model-counting** systems (eg ProbLog2) where conceptually

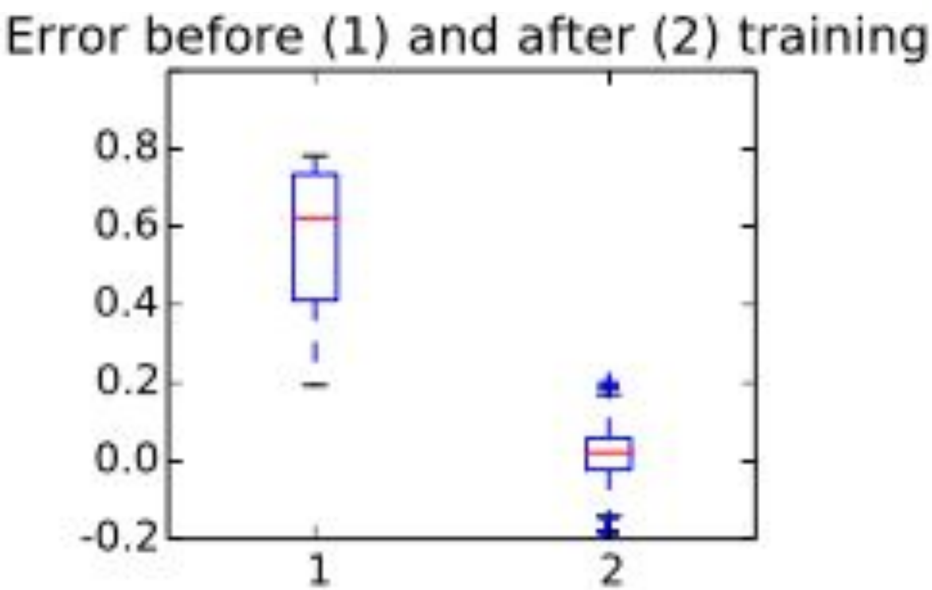
- There is a distribution Pr(KG) over KGs
  - Tuple-independence: draw a KG by picking each fact *f* with probability w_f
- The probability of a fact *f'* is the probability T+KG' implies *f'*, for a KG' is drawn from Pr(KG)

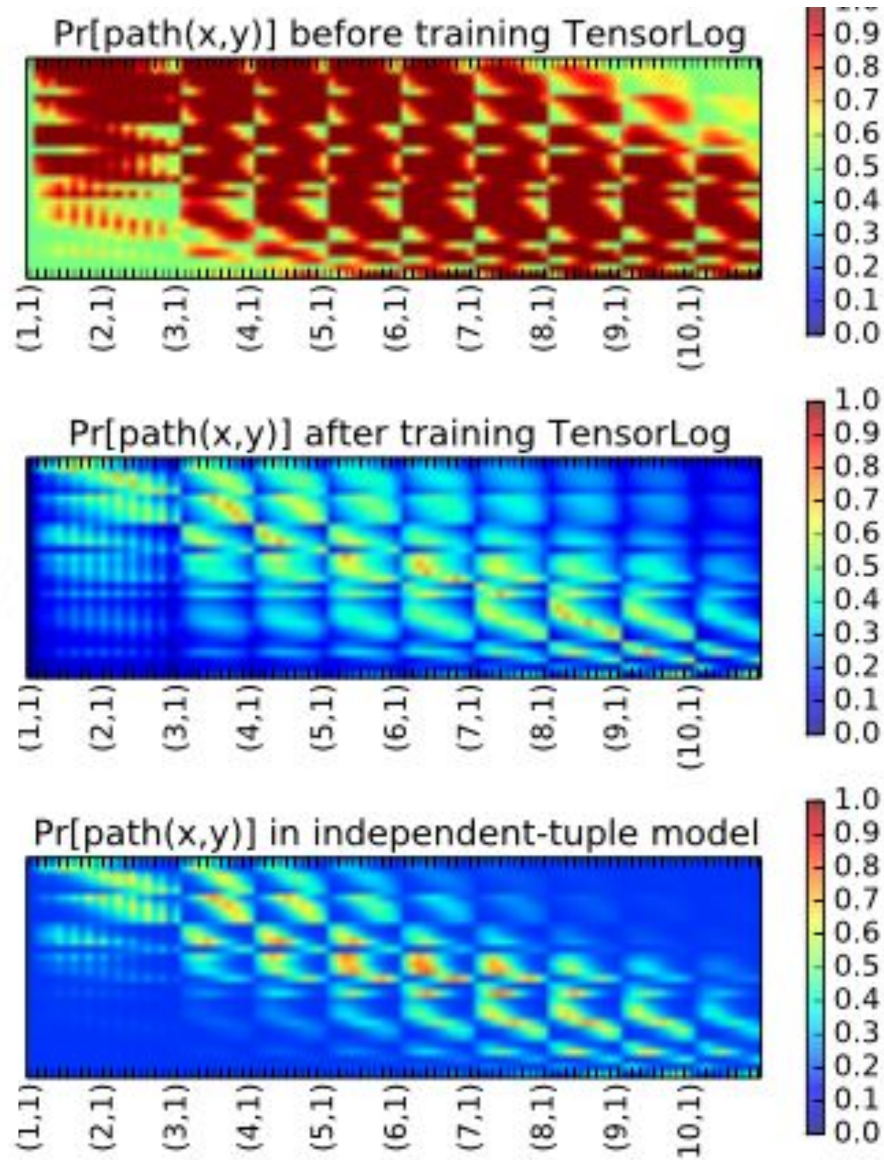
Experiments: for grid world, estimate Pr(path(a,b)) using a sample of 1M random KG/grids drawn from the tuple-independence model

Experiment: Learning Alternate Semantics

Experiment: learn grid-transition weights to approximate ProbLog2's inference weights.

Error drops by factor of 10x.



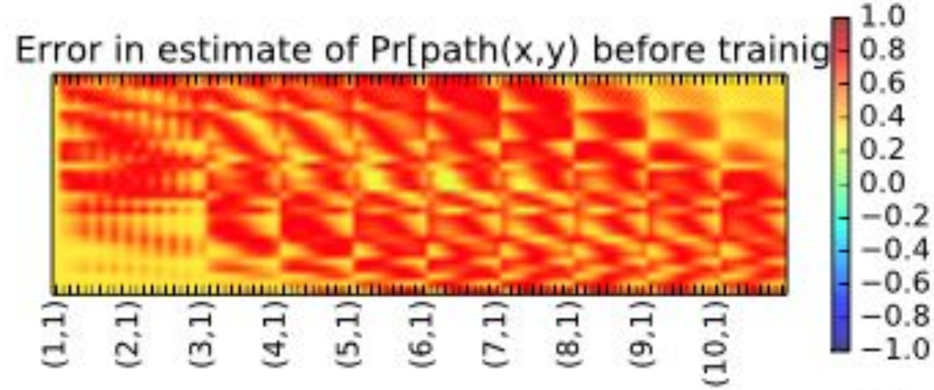


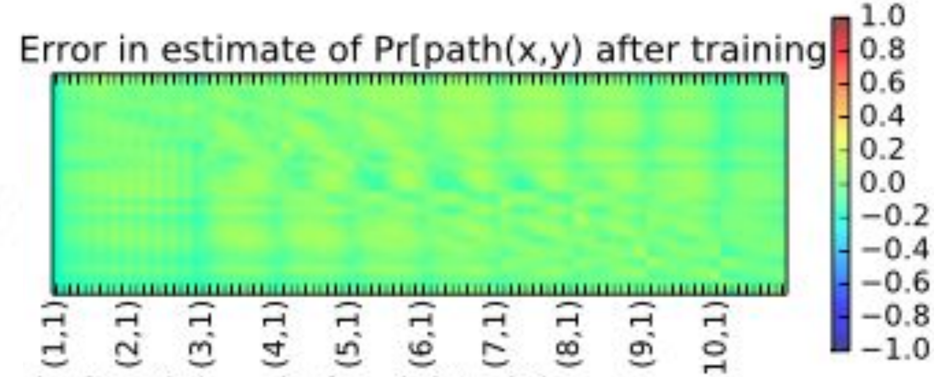
Er

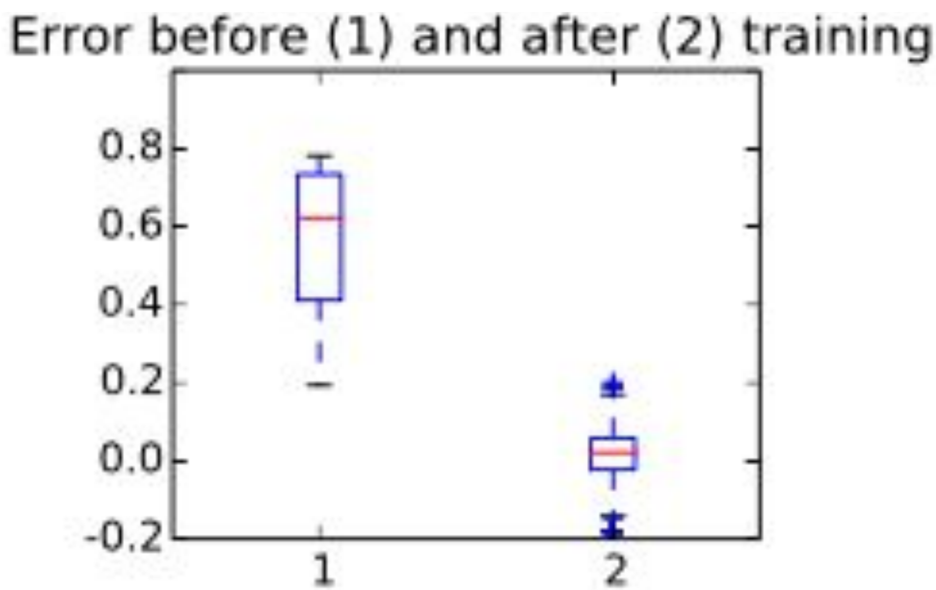
Experiment: Learning Alternate Semantics

Experiment: learn grid-transition weights to approximate ProbLog2's inference weights.

Error drops by factor of 10x.

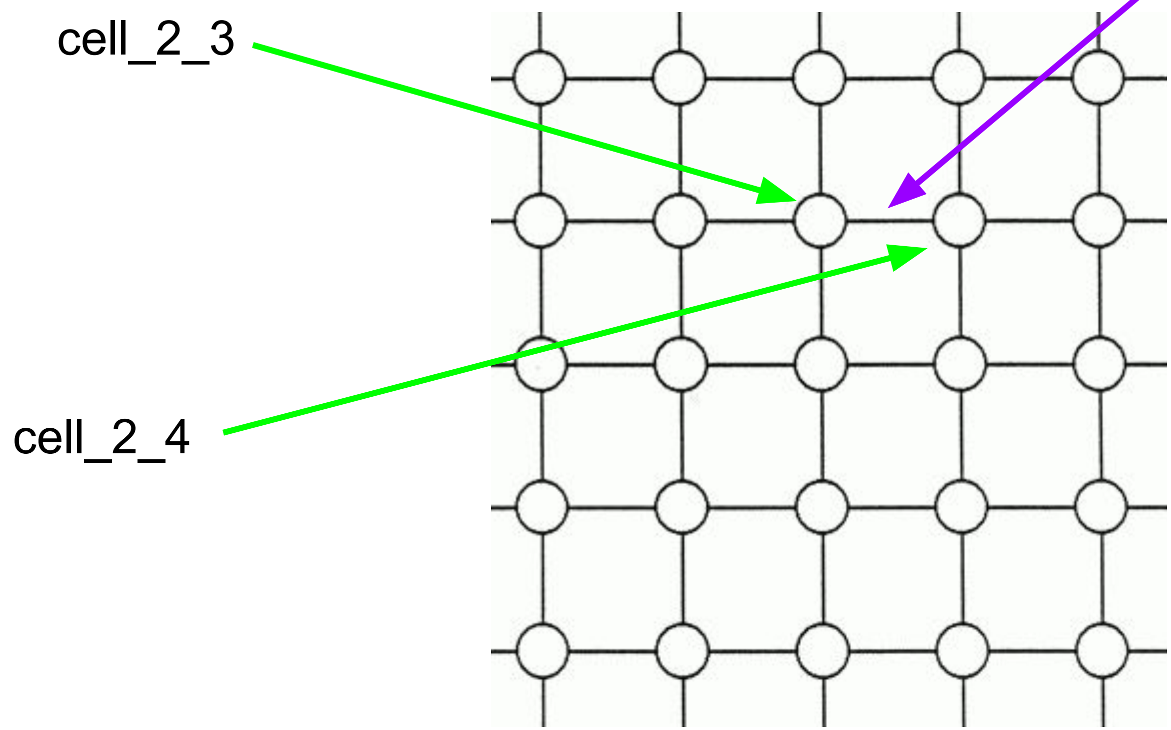






Experiment: Learning Representations

edge(cell_2_3, cell_2_4) ~~0.2~~
...



Replace learnable weight 0.2 with a function of *learned representations* of cell_2_3 and cell_2_4.

Each cell *i* has a *learned vector representation* $\mathbf{e}_i$

path(X,Y) :- edge(X,Y)
path(X,Z) :- edge(X,Z),path(Z,Y)

Experiment: Learning Representations

Experiment: learn a neural model for grid-transition weights.

edge(cell1, cell2) =
$$\underbrace{\log(1 + \exp(}_{\text{makes edge score positive}}\mathrm{sum\_d}\ (\mathbf{e}_1[d] - \mathbf{e}_2[d]))\quad *\ M[\mathrm{cell1},\mathrm{cell2}]$$

Averaged over 10 trials, 10x10 grid, 100 epochs.

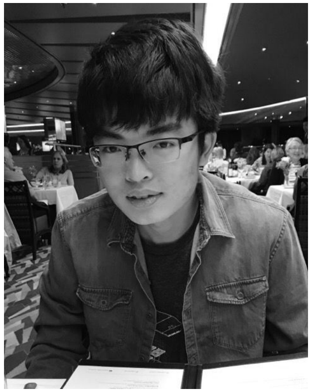
- Accuracy **97.8%**
- Accuracy of baseline: **85.8%** (one weight per edge)

Manhattan distance in embedding space, but **directional**: want weights to encourage transitions **toward** the target cell.

0,1 mask so only grid edges are considered

Tensorlog: Extension (Neural ILP)

Fang Yang, Zhilin Yang



Learning rules for TensorLog

Given only examples:
- uncle(liam,Y): Y should be {“bob”}
- aunt(liam,Y):Y should be {“mary, “sue”}
- …

Learn full model (parameters **and rules**)

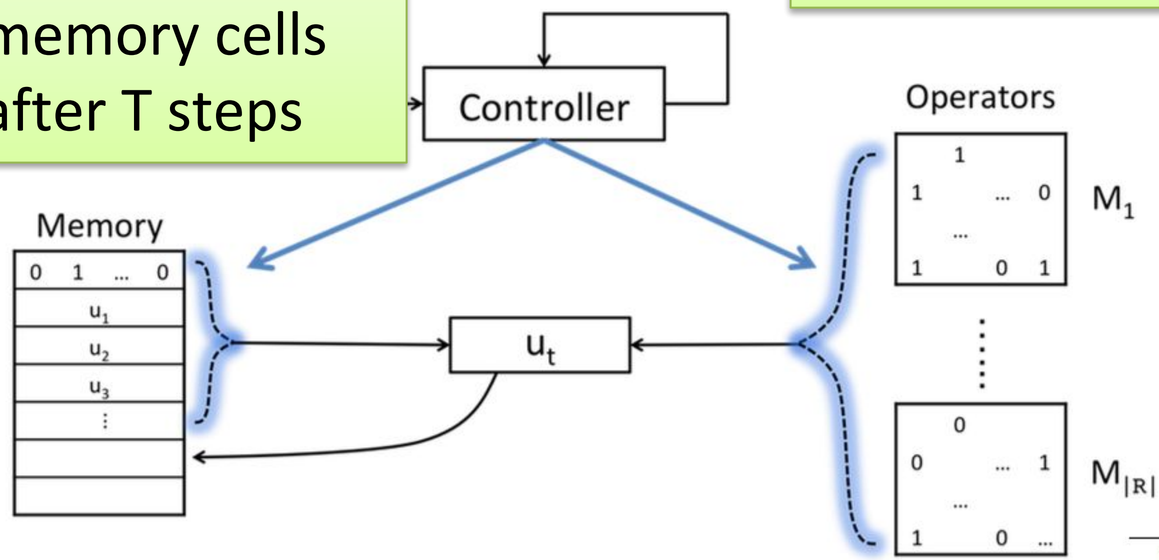
- Basic idea:
  – TensorLog programs are compiled to a sequence of *differentiable operators*
  – Each operator is applied to a memory location ~= logical variable
- Learn sequence with a neural controller

Function	$g^{r1}_{\mathtt{io}}(\vec{u}_c)$
Operation sequence defining function	$\mathbf{v}_{1,W} = \mathbf{u}_c \mathbf{M}_{\mathtt{parent}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W \mathbf{M}_{\mathtt{brother}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$
Returns	$\mathbf{v}_Y$

Learning rules for TensorLog

Final output is attention over memory cells after T steps

LSTM controller: reads p,a at each time step in computing Y : p(a,Y)



Current status: chain rules only, hard KB

New memory cell allocated at each time step: contents are formed by attention over ops and previous memory cells

Function	$g^{r1}_{\mathtt{io}}(\vec{u}_c)$
Operation sequence defining function	$\mathbf{v}_{1,W} = \mathbf{u}_c \mathbf{M}_{\mathtt{parent}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W \mathbf{M}_{\mathtt{brother}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$
Returns	$\mathbf{v}_Y$

Statistical relational learning

	ISG		Neural LP	
	T=2	T=3	T=2	T=3
UMLS	43.5	43.3	92.0	**93.2**
Kinship	59.2	59.0	**90.2**	90.1

WikiMovies with natural language queries.

Knowledge base	`directed_by(Blade Runner,Ridley Scott)`
	`written_by(Blade Runner,Philip K. Dick)`
	`starred_actors(Blade Runner,Harrison Ford)`
	`starred_actors(Blade Runner,Sean Young)`

Questions	What year was the movie `Blade Runner` released? Who is the writer of the film `Blade Runner`?

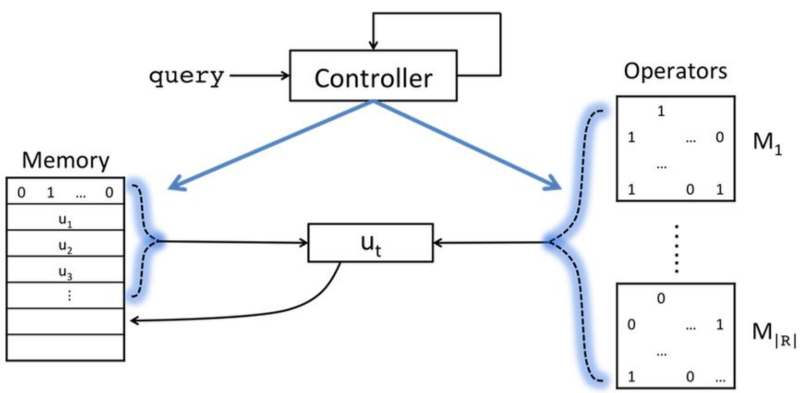
Model	Accuracy
Key-Value Memory Network	93.9
Neural LP	**94.6**

Results for Neural Inductive Logic Programming

Node+LinkFeat	94.3	**87.0**	34.7
DistMult	94.2	57.7	**40.8**
Neural LP	**94.5**	83.7	36.2

Recovering rules for Neural ILP

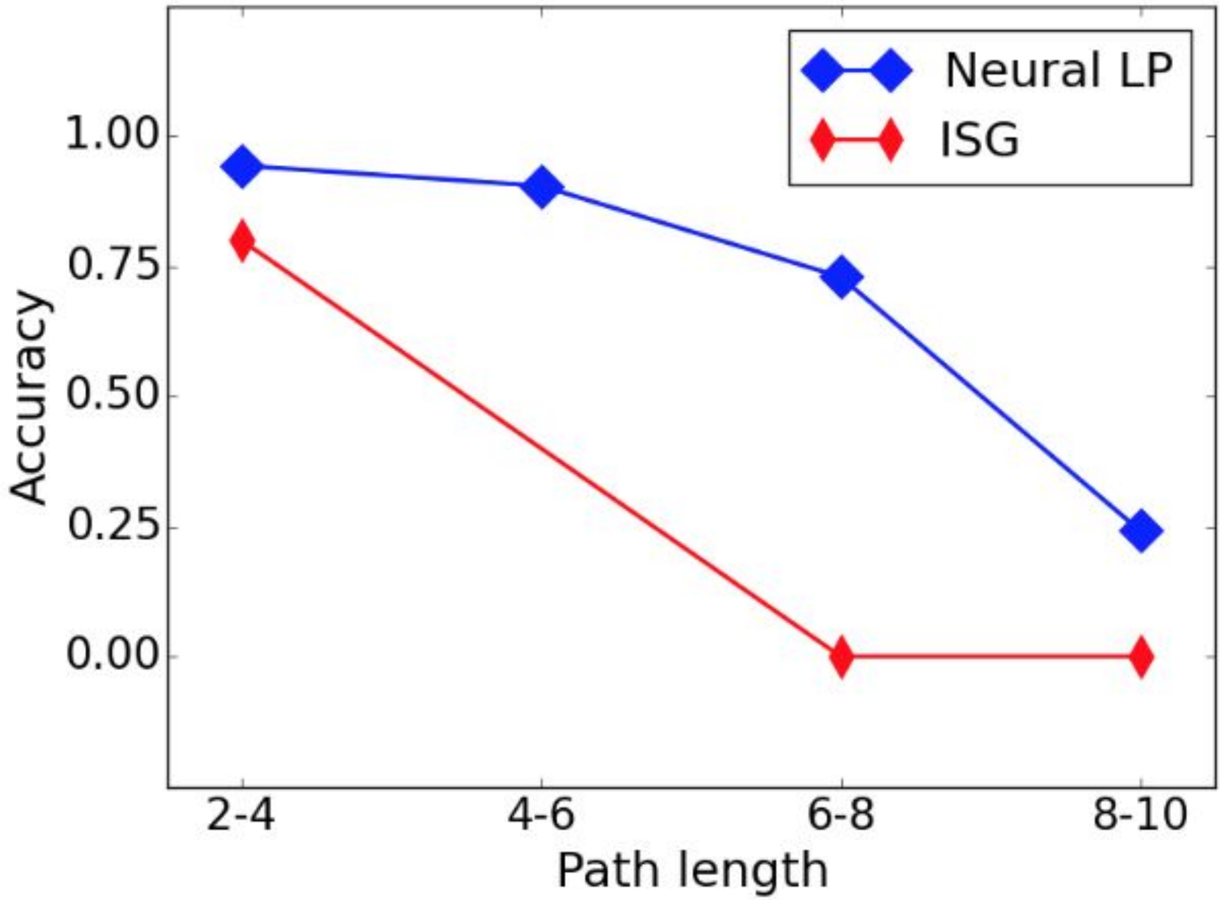
1.00	`partially_contains(C,A)` ← `contains(B,A)` ∧ `contains(B,C)`
0.45	`partially_contains(C,A)` ← `contains(A,B)` ∧ `contains(B,C)`
0.35	`partially_contains(C,A)` ← `contains(C,B)` ∧ `contains(B,A)`
1.00	`marriage_location(C,A)` ← `nationality(C,B)` ∧ `contains(B,A)`
0.35	`marriage_location(B,A)` ← `nationality(B,A)`
0.24	`marriage_location(C,A)` ← `place_lived(C,B)` ∧ `contains(B,A)`
1.00	`film_edited_by(B,A)` ← `nominated_for(A,B)`
0.20	`film_edited_by(C,A)` ← `award_nominee(B,A)` ∧ `nominated_for(B,C)`



Function	$g^{r1}_{\texttt{io}}(\vec{u}_c)$
Operation sequence defining function	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{parent}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{brother}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$
Returns	$\mathbf{v}_Y$

Results for Neural Inductive Logic Programming

Synthetic task: learning specific long paths in grid, like "NE-NE-S-S"



Where to next?

William Cohen
Google AI

TensorLog model

who acted in the movie Wise Guys? ['Harvey Keitel', 'Danny DeVito', 'Joe Piscopo', …]
what is a film written by Luke Ricci? ['How to Be a Serial Killer']
…

```
answer(Question, Entity) :-
  mentions_entity(Question,Movie),
  starred_actors(Movie,Entity),
   feature(Question,F),weight_sa_io(F).
  % w_sa_f: weight for  starred_actors(i,o)
...
answer(Question, Movie) :-
  mentions_entity(Question,Entity),
  written_by(Movie,Entity),
  feature(Question,F),weight_wb_oi(F).
...
Total: 18 rules
```

starred_actors	Wise Guys	Harvey Keitel
starred_actors	Wise Guys	Danny DeVito
starred_actors	Wise Guys	Joe Piscopo
starred_actors	Wise Guys	Ray Sharkey
directed_by	Wise Guys	Brian De Palma
has_genre	Wise Guys	Comedy
release_year	Wise Guys	1986
…		
written_by	How to .. Killer	Luke Ricci
has_genre	How to .. Killer	Comedy
...		

TensorLog model

Is this the best interface to give Google programmers to build models?
Problems:

- Hard to predict what will happen in the compiled model (what does the BP stage do to construct a model?)
- Hard to quantify over *relations* (do second order reasoning)
- Awkward to swap back and forth between TensorFlow and TensorLog (declarative vs functional)

Proposal: language for *compilation target for Tensorlog*

```
answer(Question, Entity) :-
  mentions_entity(Question,Movie
  starred_actors(Movie,Entity),
   feature(Question,F),weight_sa_i
  % w_sa_f: weight for  starred_ac
...
answer(Question, Movie) :-
  mentions_entity(Question,Entity
  written_by(Movie,Entity),
  feature(Question,F),weight_wb_
...
```

Neural Query Language: 1st-order

-1: go “backwards” mode oi

```
answer =
  question.mentions_entity().starred_actors().if_exists(
          question.feature() & nq.one(‘starred_actors’).indicates(-1))
 | question.mentions_entity().directed_by().if_exists(
          question.feature() & nq.one(‘directed_by’).indicates(-1))
 |
….
```

“features that indicate the ‘starred_actors’ KG relation”

“features that indicate the ‘directed_by’ KG relation”

x.**if_exists**(y): **return vector x multiplied by sum of weights in y**
… a soft version of *return x iff y is non-empty else empty set*

```
answer(Question, Entity) :-
  mentions_entity(Question,Movie),
  starred_actors(Movie,Entity),
  feature(Question,F),
  indicates(F,’starred_actors’).
...
answer(Question, Movie) :-
  mentions_entity(Question,Entity),
  written_by(Movie,Entity),
  feature(Question,F),
  indicates(F,’written_by’)
...
```

Neural Query Language: 1st-order

```
answer =
  question.mentions_entity().starred_actors(+1).if_exists(
          question.feature() & nq.one('starred_actors').indicates_rel(-1)).if_exists(
               question.feature() & nq.one('forward').indicates_dir(-1)))
 | question.mentions_entity().starred_actors(-1).if_exists(
          question.feature() & nq.one('starred_actors').indicates_rel(-1)).if_exists(
               question.feature() & nq.one('backward').indicates_dir(-1)))

....
```

```
answer(Question, Entity) :-
  mentions_entity(Question,Movie),
  starred_actors(Movie,Entity),
  feature(Question,F),
  indicates_rel(F,'starred_actors'),
  indicates_dir(F,'forward').
...
```

NQL: semantics in Tensorflow

variable/expression output x	a vector encoding a weighted set (localist representation)
nq.one('bob','person') x.jump_to('bob','person')	v_bob, one hot vector for entity 'bob'
nq.all('person') x.jump_to_all('person')	k-hot vector for set off all elements of type 'person' i.e. a ones vector
nq.none('person') x.jump_to_none('person')	k-hot vector for empty set of elements of type 'person' i.e. a zeros vector
x.r() x.follow('r')	x.dot(M_r) where M_r is sparse matrix for r and x a k-hot vector
x | y x + y	x + y
x & y x * y	x * y Hadamard aka component-wise product
x.filtered_by('r','bob') x.weighted_by('r','bob')	x * v_bob.dot(M_r') M_r' is transpose
x.if_exists(y) x.weighted_by_sum(y)	x * y.sum()

Neural Query Language: 2nd-order

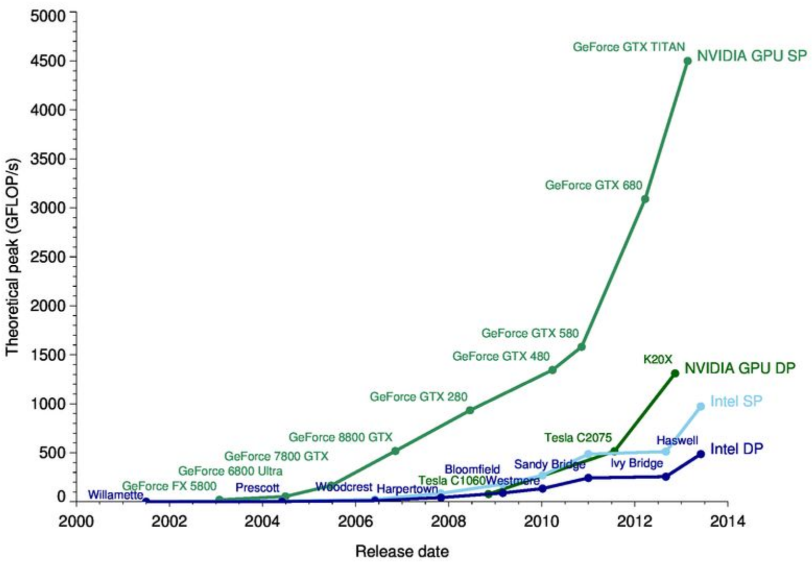
```
def kg_relation(question):
  return question.features().feat2rel()  % classify relation

def answer(question):
  return question.mentions_entity().follow(kg_relation(question))
```

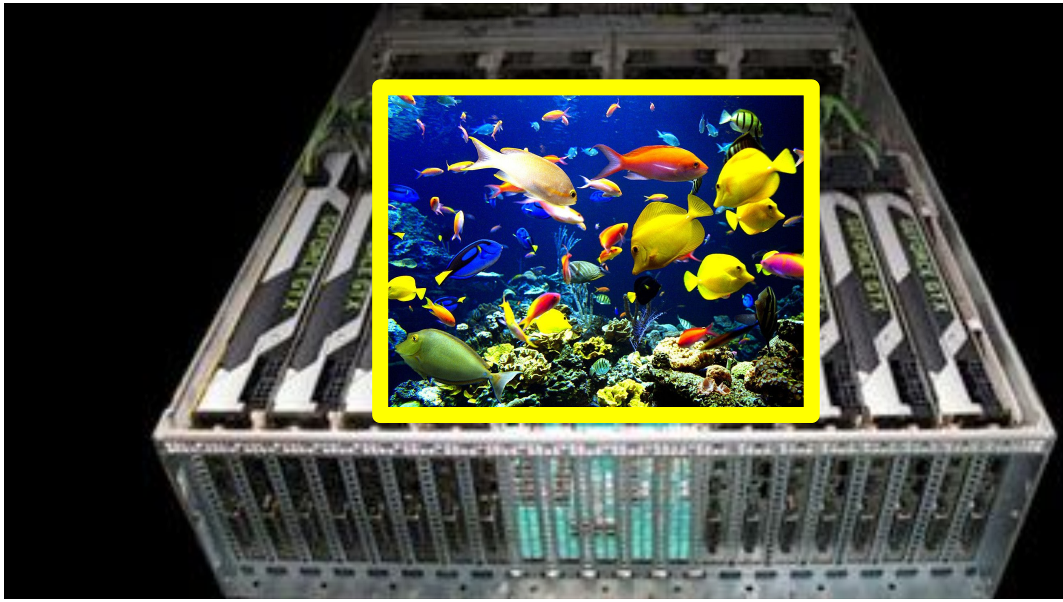
```
                                     verb(t37,starred_in)
starred_in(tom_hanks,the_post) →  subject(t37,tom_hanks)
                                     object(t37,the_post)
```

x.follow(g) == (x.subject(-1) & g.verb(-1)).object()

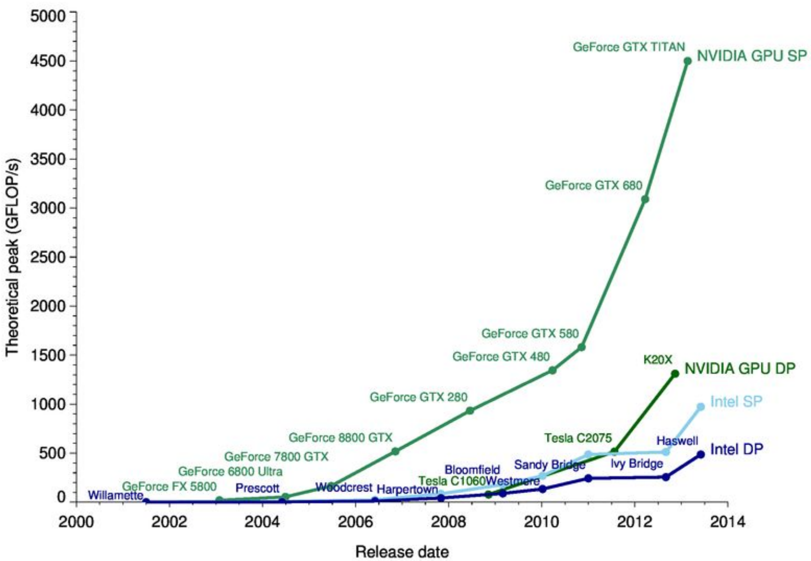
x={tom_hanks} g={starred_in}: (tom_hanks is sub) & (starred_in is verb) → object

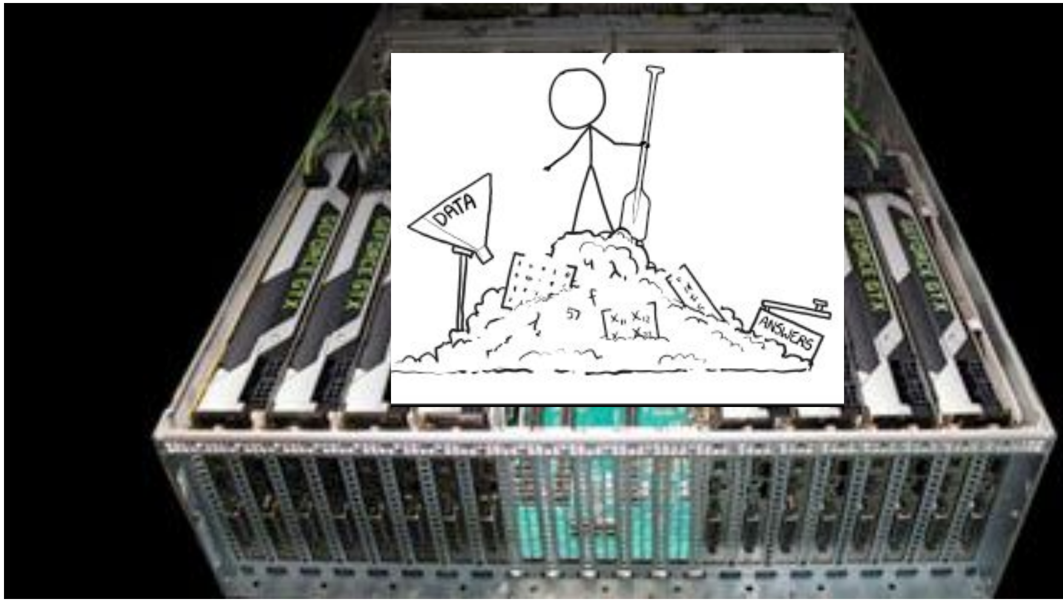
Conclusions and Wrap-Up



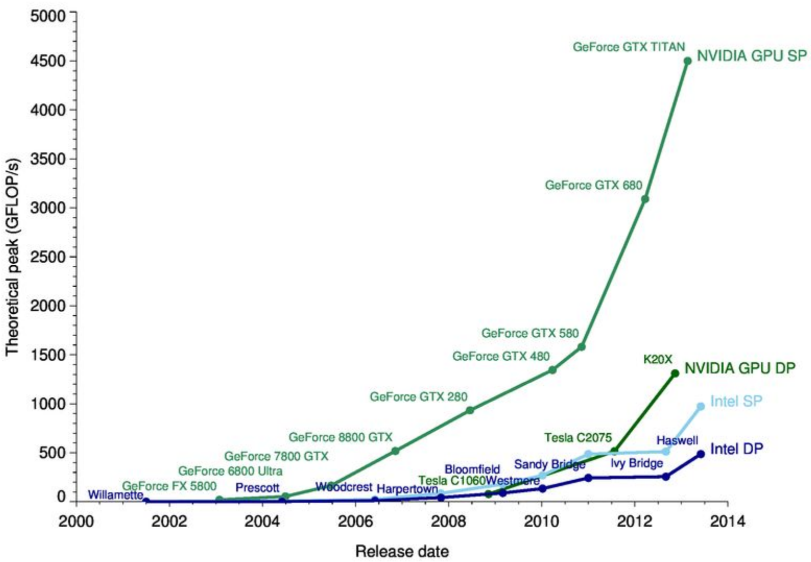


Conclusions and Wrap-Up





Conclusions and Wrap-Up



How should logic and logic programming approaches to AI be integrated with “neural” / “deep” / GPU-based approaches to AI?

Conclusions and Wrap-Up

How should logic and logic programming approaches to AI be integrated with “neural” / “deep” / GPU-based approaches to AI?

TensorFlow tries to answer this in one way:

- Scalable - but restricted - declarative subset of Prolog
- Very efficient for learning and inference
- Combinable with neural methods:
    - Eg: Logistic regression model “on top” of proof counts (for tuple-independence)
    - Eg: Representation learning “underneath” (to define edge weights)