Data-driven Optimization for Inductive Generalization

Nham Le and Arie Gurfinkel

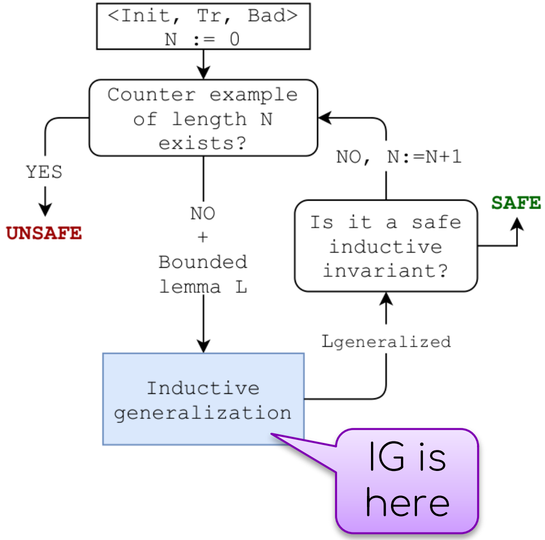
Xujie Si

What are we trying to optimize

Big picture: Symbolic Model Checking

Modern SMCs share the common basis: IC3-style algorithms

Inductive generalization (IG): the key to the efficiency of modern IC3-style Symbolic Model Checkers



A. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011
N. Eén, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability. FMCAD 2011

A Typical Inductive Generalization Query

and ~~(x_3)~~
(x_1)
(x_6 = 1)
(x_9 - x_10 >=41)
(x_5 = 1)

inductive?
YES

and ~~(x_1)~~
~~(x_6 = 1)~~
(x_9 - x_10 >=41)
(x_5 = 1)

inductive?
NO
YES

and (x_1)
~~(x_9 - x_10 >=41)~~
~~(x_5 = 1)~~

inductive?
NO
YES

and (x_1)
(x_9 - x_10 >=41)

Problem:
Inductive checks are expensive!

Our goal

We want a heuristics that:

Can check dropping multiple literals at the same time

Can be learned based on past behavior

Can generalize to unseen literals

On the road to our goal

Is there something to learn?

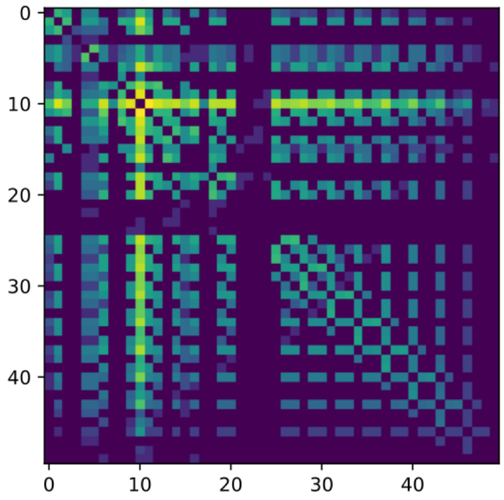
Representation learning of symbolic formulas

Learning for inductive generalization

Are the learned heuristics useful?

Is there something to learn?

Literal co-occurrences in solving



PRODUCER_CONSUMMER_luke_2_e7_1068_e8_1019

Conjecture:

Some groups of literals may always be dropped or kept together

What if we plot the literal co-occurrences matrix?

how many times $\mathbf{lit}_i$ and $\mathbf{lit}_j$ are kept together?

There are strong signals!

Why Machine Learning in the first place?

We want to avoid hand-crafted heuristics that do not generalize well

We want a heuristic that applies to new, previously unseen, literals

Prior to ML, we have tried several hand-crafted heuristics using Boolean abstraction

- but they were not stable and do not extend to many benchmarks

Challenge 1: Representation learning

Literals are symbolic formulas

Machine learning algorithms/frameworks only work with fixed length vectors of real numbers

That is not a new problem!

Can we use existing techniques in PL+ML space?

CODE2VEC: **LEARNING DISTRIBUTED REPRESENTATIONS OF CODE**

Appeared in POPL'2019

Code2Inv

IR2VEC

(off the shelf) solution

$x_9 - x_{10} >= 41$

tokenization

x_9	-	x_{10}	>=	41

encoding

<VAR>	-	<VAR>	>=	<NUM>

+	[0.82, 0.86, 0.43, 0.56, 0.94]
-	[0.36, 0.46, 0.65, 0.94, 0.61]
*	[0.66, 0.48, 0.51, 0.79, 0.03]
/	[0.31, 0.01, 0.45, 0.91, 0.95]
=	[0.56, 0.47, 0.62, 0.02, 0.82]
<=	[0.20, 0.39, 0.55, 0.87, 0.90]
>=	[0.11, 0.50, 0.78, 0.91, 0.31]
<VAR>	[0.93, 0.84, 0.03, 0.94, 0.81]
<NUM>	[0.10, 0.97, 0.69, 0.24, 0.20]

embedding

```
[[0.93, 0.84, 0.03, 0.94, 0.81],
[0.36, 0.46, 0.65, 0.94, 0.61],
[0.93, 0.84, 0.03, 0.94, 0.81],
[0.11, 0.50, 0.78, 0.91, 0.31],
[0.10, 0.97, 0.69, 0.24, 0.20]]
```

Important semantics is lost!

```
x1 + 2*x3 + 7*x5 >= 10

x1 + 2*x3 + 7*x5 >= 14
```

Off the shelf solution

```
emb(<VAR> + <NUM>*<VAR> + <NUM>*<VAR> >= <NUM>)

emb(<VAR> + <NUM>*<VAR> + <NUM>*<VAR> >= <NUM>)
```

```
x1 + 2*x3 + 7*x5 >= 10

x4 + 7*x2 + 2*x8 >= 0
```

Off the shelf solution

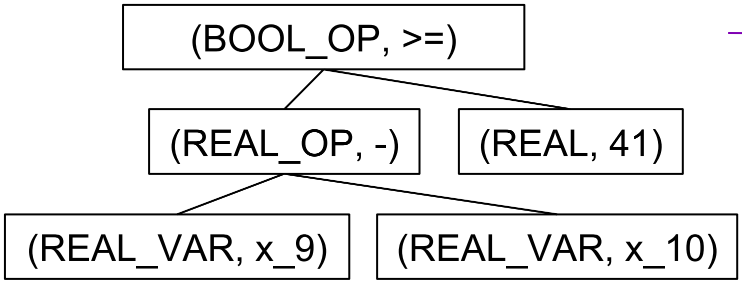
```
emb(<VAR> + <NUM>*<VAR> + <NUM>*<VAR> >= <NUM>)

emb(<VAR> + <NUM>*<VAR> + <NUM>*<VAR> >= <NUM>)
```

Inputs are different, but outputs are identical

Semantically important information is lost before learning!

BOOL_OP	[0.82, 0.86, 0.43, 0.56, 0.94]
REAL_	[illegible]
REAL_	[illegible]
REAL	[illegible]
BOOL_	[illegible]
INT_VAR	[illegible]
INT	[0.11, 0.50, 0.78, 0.91, 0.31]
…	…

>=	[0.56, 0.47, 0.62, 0.02, 0.82]
+	[0.70, 0.98, 0.65, 0.75, 0.49]
-	[0.11, 0.50, 0.78, 0.91, 0.31]
*	[0.35, 0.95, 0.43, 0.62, 0.50]
…	[0.35, 0.08, 0.60, 0.98, 0.01]



Naïve variable Embedding

Positional Embedding

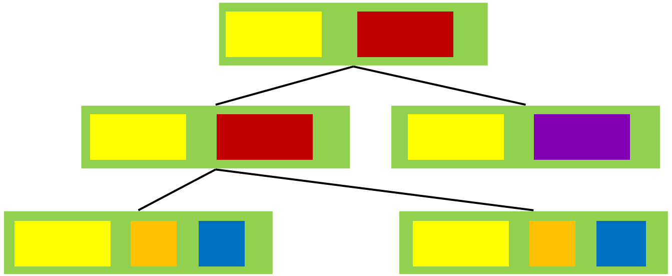
Constant Embedding

Kind Embedding

Op Embedding

"x_0"	[0.60, 0.71, 0.56, 0.97, 0.17]
"x_1"	[0.95, 0.59, 0.47, 0.83, 0.87]
"x_2"	[0.14, 0.53, 0.07, 0.26, 0.45]
"x_3"	[0.66, 0.32, 0.09, 0.07, 0.41]
…	[0.34, 0.84, 0.61, 0.21, 0.75]

Our solution



Positional Embedding

To be useful for machine learning algorithms, we want:

Each absolute position **t** is mapped to a fixed length vector **PE(t)**

Each entry in the vector should be in a small range

If two positions differ by **k**, **PE(t)** and **PE(t+k)** should differ by a linear transformation **Tr** that only depends on **k**

Positional Embedding

Embedding using sine and cosine!

Not trivial, was a huge breakthrough in Natural Language Processing!

Formally:

Position **t** is converted into a vector **PE** of length **d**

Each entry **i** in the vector **PE** is

$$\mathrm{PE}^d(t)_i = \begin{cases} \sin(\omega_k \cdot t) & \text{if } i = 2k \\ \cos(\omega_k \cdot t) & \text{if } i = 2k+1 \end{cases}$$

$$\text{where } \omega_k = 10000^{-2k/d}$$

Ashish Vaswani et al. Attention is all you need. (NIPS'17)

Constant Embedding

What we want:

Each number ρ is mapped to a fixed length vector

Numbers that are vastly different should be easily distinguishable

Each entry in the vector should be in a small range

Constant Embedding

Scientific notation + one hot encoding!

Example

CE(42) = [4.2 0 0 0 1 0] with MAX_E = 2

CE(42) = [4.2 0 0 1 0 0 0 0] with MAX_E = 3

$\rho = s * 10^e$ in the scientific notation

ρ is converted into a vector of length 2(n+1)

First entry: s

The rest 2*n entries: one hot encoding for e between [-MAX_E, +MAX_E]

(out of range e's are mapped to either –MAX_E or MAX_E)

Recap

Need to convert formulas to a structure of fix-length vectors of numbers (list of vectors or tree of vectors)

Off-the-shelf solution abstracts too much semantic information

Our solution retains positional, value, and kind information

*Spoiler: Our solution shows a difference in practice

Challenge 2:
Learning to generalize

How to formulate the learning problem?

What neural network architecture should we use?

IG as a tagging process

```
and (x_3)
    (x_1)
    (x_6 = 1)
    (x_9 - x_10 >=41)
    (x_5 = 1)
```

→ ITERDROP →

```
and (x_1)
    (x_9 - x_10 >=41)
```

ITERDROP can be viewed as a tagging process:
Given two tags 0 and 1, which literal is tagged 1, which is tagged 0?

```
and (x_3)
    (x_1)
    (x_6 = 1)
    (x_9 - x_10 >=41)
    (x_5 = 1)
```

→ ITERDROP →

```
0
1
0
1
0
```

The Lemma tagging problem

A datapoint (x, y) in the dataset

input x: a lemma represented as an ordered list of literals

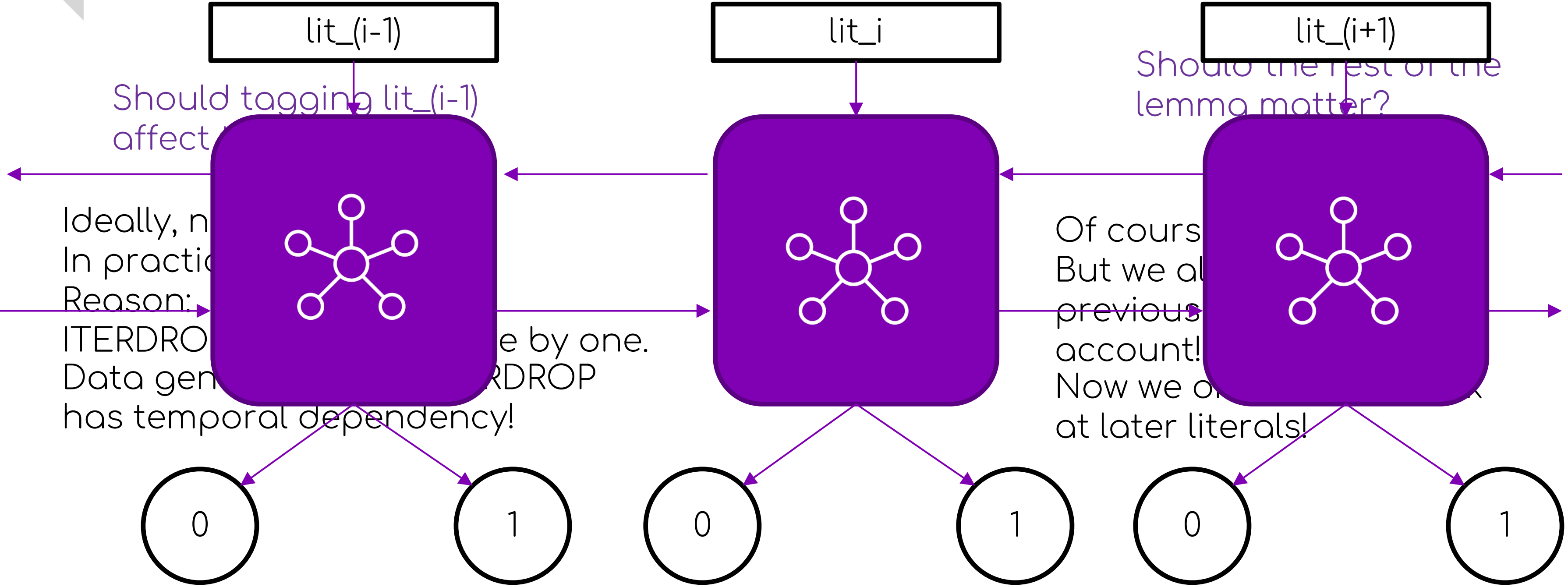
output y: a binary mask corresponding to ITERDROP's result

($|x| = |y|$)

Learning problem

Train a tagger M: $x \rightarrow \{0,1\}^{|x|}$ s.t $M(x) \sim y$ for all datapoints (x, y)

What neural network architecture should we use?



Should tagging lit_(i-1) affect

Should the rest of the lemma matter?

Ideally, n
In practic
Reason:
ITERDRO e by one.
Data gen RDROP
has temporal dependency!

Of cours
But we al
previous
account!
Now we o
at later literals!

At tagging lit_i

(we use BiLSTM in our implementation)

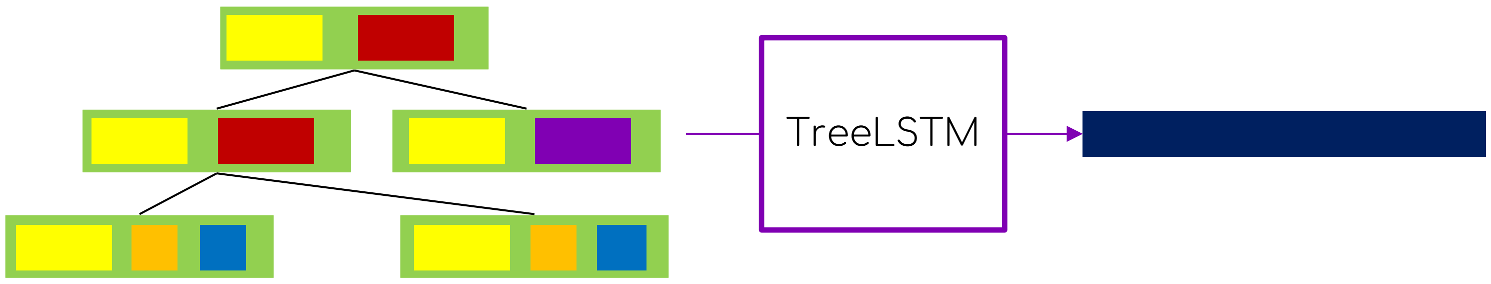
This is Bidirectional Recurrent Neural Net!

What about the trees in Representation Learning?

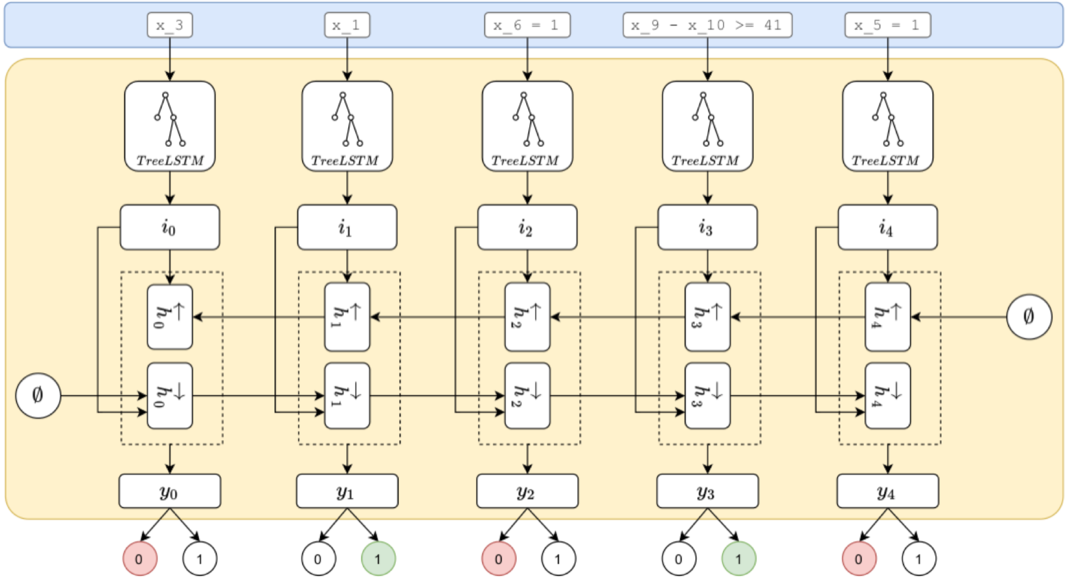
Literals are represented as **trees of vectors**

Inputs to the Bidirectional RNN are **single vectors**

Solution: Feed the tree of vectors through a TreeLSTM



Full Model



x_3
x_1
x_6 = 1
x_9 - x_10 >= 41
x_5 = 1
TreeLSTM
TreeLSTM
TreeLSTM
TreeLSTM
TreeLSTM
i_0
i_1
i_2
i_3
i_4
$h_0^\uparrow$
$h_1^\uparrow$
$h_2^\uparrow$
$h_3^\uparrow$
$h_4^\uparrow$
$\emptyset$
$\emptyset$
$h_0^\downarrow$
$h_1^\downarrow$
$h_2^\downarrow$
$h_3^\downarrow$
$h_4^\downarrow$
y_0
y_1
y_2
y_3
y_4
0 1
0 1
0 1
0 1
0 1

How is the model used?

XDROP, a drop-in replacement for ITERDROP

XDROP is only one of the many ways to use the neural network!

and ~~(x_3)~~
(x_1)
~~(x_6 = 1)~~
(x_9 - x_10 >=41)
~~(x_5 = 1)~~

0
1
0
1
0

inductive?
YES

and ~~(x_1)~~
~~(x_9 - x_10 >=41)~~

inductive?
NO

and (x_1)
(x_9 - x_10 >=41)

ROPEY: A SMC using XDROP

Core SMC is based on SPACER, written in C++

Model inferencing is written in PyTorch

Communication is done through gRPC

Empirical evaluation

Online learning:

How well does a model trained on 10 minutes of solving X guide the rest of solving X?

Transfer learning:

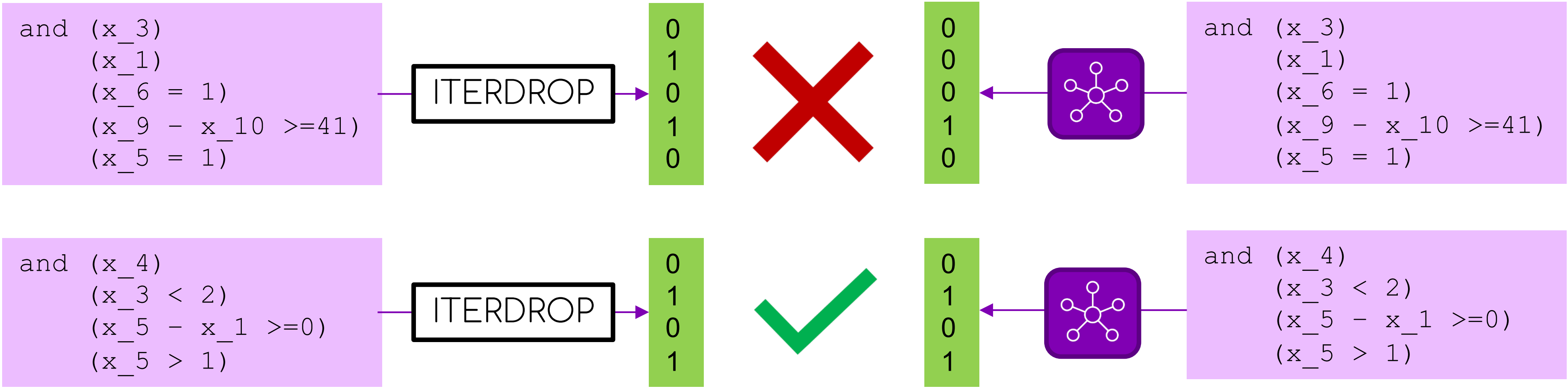
How well does a model trained on solving X to completion guide the solving of its variants X_1, X_2, etc. ?

(exact formal definition and dataset are in the paper)

Metrics

Perfect prediction ratio (PPR):

How often do M and ITERDROP return the same exact answer?



Perfect prediction ratio: 0.5

Not all instances are the same!

Instances with too few IG queries (e.g., < 10) are:

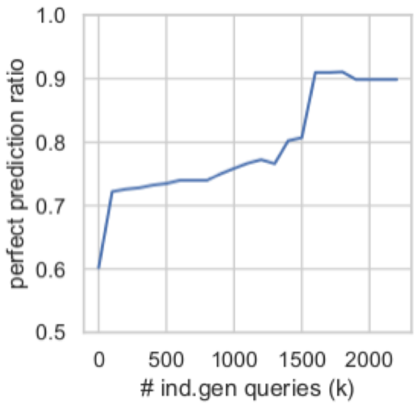
Very hard to train

Subjected to noise in measurement

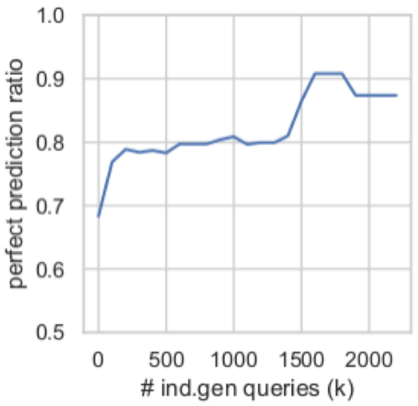
Solution: Plot PPR for all instances with at least 100 IG queries, 200 queries, etc.

Predictive Power Result

Online learning



Transfer learning



Running time

SPACER's running time is easy to measure

ROPEY's running time has multiple components:

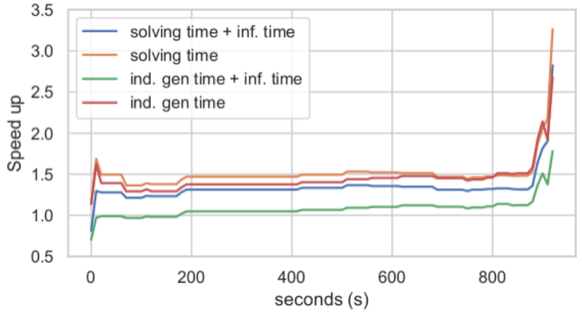
- SMC solving time
- Model inferencing time (GPU dependent)
- Data parsing time
- gRPC communication time

inferencing time can be improved by better engineering and hardware (GPU/TPU)

Not all instances are the same! (again)

	All	Non-trivial
solving + inf. time	0.81560	1.25385
solving time	1.14085	1.69792
ind. gen time	1.13570	1.63041
ind. gen + inf. time	0.70519	0.91891

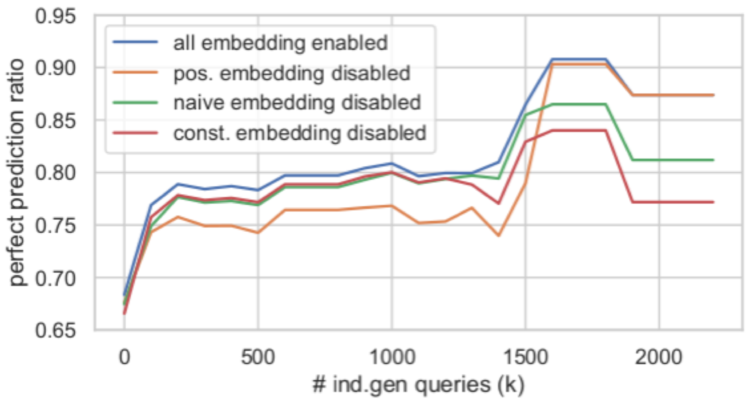
Small instances are subjected to noise!
Plot running time improvement for instances that are solved by SPACER in under 10 seconds, 20 seconds, 30 seconds, etc.
(Instances that takes more than 10 second to solved are called non-trivial)



How about timed out instances?
Use the time needed to reach the same depth explored by SPACER

Do Constant and Positional Embedding make a difference?

(Have you tried turning it off and on again?)



Conclusion

A data-driven approach to improve inductive generalization

Learned neural nets show promising predictive power

The predictive power translates to meaningful improvement in running time over the state-of-the-art SMC

Future work

Explore other ways to use the neural network

Explore other neural architecture, e.g.,Transformer

Better engineering for ROPEY

Thank you!