Neural Program Synthesis and Inductive Logic Programming

Richard Evans
richardevans@google.com
@LittleBimble

Overview

1. What is Neural Program Synthesis?
2. Learning logic programs
3. How to learn logic programs from examples
4. Differentiable Inductive Logic Programming
5. Experiments
6. Related Work

1. What is Neural Program Synthesis?

Learning Procedures from Examples

Given some input / output examples, learn
a **general procedure for transforming inputs into outputs**.

Learning Procedures from Examples

Given some input / output examples, learn
a **general procedure for transforming inputs into outputs**.

$$
\begin{aligned}
[] &\mapsto 0 \\
[2] &\mapsto 1 \\
[4, 3] &\mapsto 2 \\
[1, 2, 2] &\mapsto 3
\end{aligned}
$$

Learning Procedures from Examples

Given some input / output examples, learn
a **general procedure for transforming inputs into outputs**.

$$[[1]] \mapsto [[]]$$
$$[[4, 3]] \mapsto [[4]]$$
$$[[2, 3], [1]] \mapsto [[2], []]$$
$$[[1, 3, 2], [2, 4]] \mapsto [[1, 3], [2]]$$

Learning Procedures from Examples

We shall consider **three approaches**:

1. Symbolic program synthesis
2. Neural program induction
3. Neural program synthesis

Symbolic Program Synthesis (SPS)

Given some input/output examples, produce an **explicit human-readable program** that, when evaluated on the inputs, produces the outputs.

This uses a **symbolic search procedure** to find the program.

Symbolic Program Synthesis (SPS)

Input / Output Examples

$$[[1]] \mapsto [[]]$$
$$[[4, 3]] \mapsto [[4]]$$
$$[[2, 3], [1]] \mapsto [[2], []]$$
$$[[1, 3, 2], [2, 4]] \mapsto [[1, 3], [2]]$$

Explicit Program

```
map (reverse . tail . reverse)
```

```
def remove_last(x):
  return [y[0:len(y)-1] for y in x]
```

Symbolic Program Synthesis (SPS)

Input / Output Examples

$$[[1]] \mapsto [[]]$$
$$[[4,3]] \mapsto [[4]]$$
$$[[2,3],[1]] \mapsto [[2],[]]$$
$$[[1,3,2],[2,4]] \mapsto [[1,3],[2]]$$

Explicit Program

```
map (reverse . tail . reverse)
```

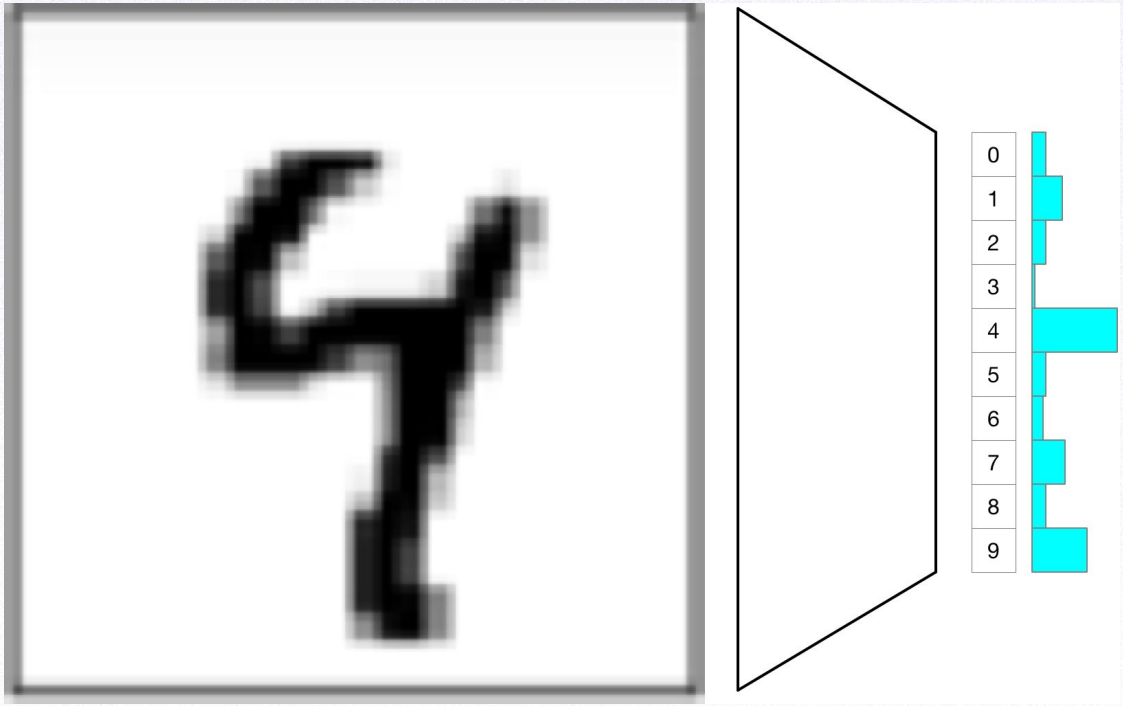
```
def remove_last(x):
  return [y[0:len(y)-1] for y in x]
```

Examples: MagicHaskeller, λ², Igor-2, Progol, Metagol

Symbolic Program Synthesis (SPS)

Data-efficient?	Yes
Interpretable?	Yes
Generalises outside training data?	Yes
Robust to mislabelled data?	Sometimes
Robust to ambiguous data?	No

Ambiguous Data



Neural Program Induction (NPI)

Given input/output pairs, a neural network learns a **procedure for mapping inputs to outputs**.

The network generates the output from the input directly, using a **latent representation of the program**.

Here, the general procedure is **implicit** in the weights of the model.

Neural Program Induction (NPI)

Examples:

Differentiable Neural Computers (Graves *et al.*, 2016)

Neural Stacks/Queues (Grefenstette *et al.*, 2015)

Learning to Infer Algorithms (Joulin & Mikolov, 2015)

Neural Programmer-Interpreters (Reed and de Freitas, 2015)

Neural GPUs (Kaiser and Sutskever, 2015)

Neural Program Induction (NPI)

Differentiable Neural Computers (Graves *et al.*, 2016):

- External memory
- Read/write to memory via differentiable attention mechanism

Neural Stacks/Queues (Grefenstette *et al.*, 2015)

- External stack
- Push/pop to stack via differentiable attention mechanism

Neural Program Induction (NPI)

Neural Programmer-Interpreters (Reed and de Freitas, 2015):

- It is given explicit traces of desired behaviour

Neural GPUs (Kaiser and Sutskever, 2015)

- A differentiable model of a cellular automaton

Neural Program Induction (NPI)

In all these cases, they provide a *differentiable model of the virtual machine*.

The network generates the output from the input directly, using a **latent representation of the program**.

Here, the general procedure is **implicit** in the weights of the model.

Neural Program Induction (NPI)

Data-efficient?	Not very
Interpretable?	No
Generalises outside training data?	Sometimes
Robust to mislabelled data?	Yes
Robust to ambiguous data?	Yes

The Best of Both Worlds?

	SPS	NPI	Ideally
Data-efficient?	Yes	Not always	Yes
Interpretable?	Yes	No	Yes
Generalises outside training data?	Yes	Not always	Yes
Robust to mislabelled data?	Not very	Yes	Yes
Robust to ambiguous data?	No	Yes	Yes

Neural Program Synthesis (NPS)

Given some input/output examples, produce an **explicit human-readable program** that, when evaluated on the inputs, produces the outputs.

Use an **optimisation procedure** (e.g. gradient descent) to find the program.

Neural Program Synthesis (NPS)

Given some input/output examples, produce an **explicit human-readable program** that, when evaluated on the inputs, produces the outputs.

Use an **optimisation procedure** (e.g. gradient descent) to find the program.

Examples: ∂ILP, RobustFill, Differentiable Forth, End-to-End Differentiable Proving

The Three Approaches

	Procedure is implicit	*Procedure is explicit*
Symbolic search		Symbolic Program Synthesis
Optimisation procedure	Neural Program Induction	Neural Program Synthesis

The Three Approaches

	Procedure is implicit	*Procedure is explicit*
Symbolic search		Symbolic Program Synthesis
Optimisation procedure	Neural Program Induction	**Neural Program Synthesis**

The Three Approaches

	SPS	NPI	NPS
Data-efficient?	Yes	Not always	Yes
Interpretable?	Yes	No	Yes
Generalises outside training data?	Yes	Not always	Yes
Robust to mislabelled data?	No	Yes	Yes
Robust to ambiguous data?	No	Yes	Yes

Overview

1. What is Neural Program Synthesis?
2. **Learning logic programs**
3. How to learn logic programs from examples
4. Differentiable Inductive Logic Programming
5. Experiments
6. Related Work

2. Learning Logic Programs

What Target Language to Synthesise Programs in?

The desiderata for human programming languages are different from the desiderata for program-synthesis languages...

The target language for program synthesis should be…

1. Pure (referentially transparent)
2. Expressive

Referential Transparency

A language is **referentially transparent** if $a = b$ implies $\varphi[a] = \varphi[b]$

Python and C++ are not referentially transparent. Haskell is.

Referential transparency is important for program synthesis because equivalent programs can be identified, and the search space can be drastically pruned.

Expressiveness

We are searching through the space of programs. Suppose the branching factor is *b* and the length of the shortest solution program is *n*. Then we are searching through b^n programs.

If we can minimise *n*, then we can keep the search space tractable.

Target Languages for Program Synthesis

Good target languages

Lambda calculus

Datalog

Bad target languages

Python

C++

Combinatory Logic

Datalog is a Good Target Language for Synthesis

1. It is pure / referentially transparent
2. It is highly expressive

The Expressiveness of Datalog

Analysis	Hand-coded	Datalog
context-insensitive Java	1975	30
context-sensitive Java	3451	33
context-insensitive C	1363	308
external lock analysis	n/a	42
SQL injection analysis	n/a	38

What is Datalog?

A Datalog program is a set of ground facts (the extensional database) together with a set of clauses (the intensional database).

Ground facts:

$$\{\text{edge}(a, b), \text{edge}(b, c), \text{edge}(c, a)\}$$

Clauses:

$$\begin{aligned} \text{connected}(X, Y) &\leftarrow \text{edge}(X, Y) \\ \text{connected}(X, Y) &\leftarrow \text{edge}(X, Z), \text{connected}(Z, Y) \end{aligned}$$

What is Datalog?

We shall deal with **definite clauses** only.

We don’t allow disjunctions or existentials in the head.

Executing Datalog Programs

The consequences of a set *R* of rules is computed by repeatedly applying the rules in R until no more consequences can be derived.

Executing Datalog Programs

$$\begin{array}{ll} \text{edge}(a,b) & \text{connected}(X,Y) \leftarrow \text{edge}(X,Y) \\ \text{edge}(b,c) & \text{connected}(X,Y) \leftarrow \text{edge}(X,Z), \text{connected}(Z,Y) \\ \text{edge}(c,a) & \end{array}$$

$$\begin{aligned} C_{R,1} &= \{\text{edge}(a,b), \text{edge}(b,c), \text{edge}(c,a)\} \\ C_{R,2} &= C_{R,1} \cup \{\text{connected}(a,b), \text{connected}(b,c), \text{connected}(c,a)\} \\ C_{R,3} &= C_{R,2} \cup \{\text{connected}(a,c), \text{connected}(b,a), \text{connected}(c,b)\} \\ C_{R,4} &= C_{R,3} \cup \{\text{connected}(a,a), \text{connected}(b,b), \text{connected}(c,c)\} \\ C_{R,5} &= C_{R,4} = \text{con}(R) \end{aligned}$$

Executing Datalog Programs

I have described naive bottom-up evaluation.

There are more efficient ways of executing Datalog:

- Semi-naive: keep track of what has been added to the database
- Magic sets: use knowledge of the query to only perform the subset of the inferences that are relevant to evaluating the query

Datalog vs Prolog

- Datalog is purely declarative
- Datalog programs always terminate

Learning Logic Programs

An **ILP problem** is a tuple $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ of ground atoms, where:

- $\mathcal{B}$ is a set of background assumptions, a set of ground atoms[8].
- $\mathcal{P}$ is a set of positive instances - examples taken from the extension of the target predicate to be learned
- $\mathcal{N}$ is a set of negative instances - examples taken outside the extension of the target predicate

Learning Logic Programs

An **ILP problem** is a tuple $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ of ground atoms, where:

- $\mathcal{B}$ is a set of background assumptions, a set of ground atoms[8].
- $\mathcal{P}$ is a set of positive instances - examples taken from the extension of the target predicate to be learned
- $\mathcal{N}$ is a set of negative instances - examples taken outside the extension of the target predicate

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3), ...\}$$

$$\begin{aligned} \mathcal{P} &= \{\text{even}(0), \text{even}(2), \text{even}(4), \text{even}(6), ...\} \\ \mathcal{N} &= \{\text{even}(1), \text{even}(3), \text{even}(5), \text{even}(7), ...\} \end{aligned}$$

Learning Logic Programs

An **ILP problem** is a tuple $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ of ground atoms, where:

- $\mathcal{B}$ is a set of background assumptions, a set of ground atoms[8].
- $\mathcal{P}$ is a set of positive instances - examples taken from the extension of the target predicate to be learned
- $\mathcal{N}$ is a set of negative instances - examples taken outside the extension of the target predicate

Given an ILP problem $(\mathcal{B}, \mathcal{P}, \mathcal{N})$, a **solution** is a set R of definite clauses such that

- $\mathcal{B}, R \models \gamma$ for all $\gamma \in \mathcal{P}$
- $\mathcal{B}, R \not\models \gamma$ for all $\gamma \in \mathcal{N}$

Learning Logic Programs

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3), ...\}$$

$$\begin{aligned} \mathcal{P} &= \{\text{even}(0), \text{even}(2), \text{even}(4), \text{even}(6), ...\} \\ \mathcal{N} &= \{\text{even}(1), \text{even}(3), \text{even}(5), \text{even}(7), ...\} \end{aligned}$$

$$\begin{aligned} even(X) &\leftarrow zero(X) \\ even(X) &\leftarrow even(Y), succ2(Y,X) \\ succ2(X,Y) &\leftarrow succ(X,Z), succ(Z,Y) \end{aligned}$$

Learning Logic Programs

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3), ...\}$$

$$\begin{aligned} \mathcal{P} &= \{\text{even}(0), \text{even}(2), \text{even}(4), \text{even}(6), ...\} \\ \mathcal{N} &= \{\text{even}(1), \text{even}(3), \text{even}(5), \text{even}(7), ...\} \end{aligned}$$

$$\begin{aligned} even(X) &\leftarrow zero(X) \\ even(X) &\leftarrow even(Y), succ2(Y,X) \\ succ2(X,Y) &\leftarrow succ(X,Z), succ(Z,Y) \end{aligned}$$

Learning Logic Programs

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3), ...\}$$

$$\begin{aligned} \mathcal{P} &= \{\text{even}(0), \text{even}(2), \text{even}(4), \text{even}(6), ...\} \\ \mathcal{N} &= \{\text{even}(1), \text{even}(3), \text{even}(5), \text{even}(7), ...\} \end{aligned}$$

$$\begin{aligned} even(X) &\leftarrow zero(X) \\ even(X) &\leftarrow even(Y), succ2(Y,X) \\ succ2(X,Y) &\leftarrow succ(X,Z), succ(Z,Y) \end{aligned}$$

Overview

1. What is Neural Program Synthesis?
2. Learning logic programs
3. **How to learn logic programs from examples**
4. Differentiable Inductive Logic Programming
5. Experiments
6. Related Work

3. How to Learn Logic Programs

Two Approaches to ILP

1. Bottom-up (Progol, Aleph)
2. Top-down (Metagol, ASPAL)

Existing bottom-up approaches are unable to learn recursive clauses or to generate invented predicates.

We shall focus on a particular top-down approach, a variant of ASPAL (Corapi et al).

The basic idea is to convert the induction problem into a SAT problem.

Converting ILP to SAT

- Define a rule template τ as a way of defining a set of clauses
- Define a program template as a set of rule templates
- For each rule template τ, generate the set $cl(\tau)$ of all clauses that satisfy the template
- Introduce a boolean flag for each generated clause, indicating whether it is “on” or “off”
- Now the induction problem has been transformed into a satisfiability problem: find an assignment to the flags such that the set of clauses that are “on” together entail the positive examples and do not entail the negative examples

Converting ILP to SAT

A **rule template** τ describes a range of clauses that can be generated. It is a pair (v, int) where:

- $v \in \mathbb{N}$ specifies the number of existentially quantified variables allowed in the clause
- $int \in \{0, 1\}$ specifies whether the atoms in the body of the clause can use intensional predicates ($int = 1$) or only extensional predicates ($int = 0$)

Converting ILP to SAT

Suppose τ = (v=0, int=0):

1. $q(X,Y) \leftarrow p(X,X), p(X,Y)$
2. $q(X,Y) \leftarrow p(X,X), p(Y,X)$
3. $q(X,Y) \leftarrow p(X,X), p(Y,Y)$
4. $q(X,Y) \leftarrow p(X,Y), p(X,Y)$
5. $q(X,Y) \leftarrow p(X,Y), p(Y,X)$
6. $q(X,Y) \leftarrow p(X,Y), p(Y,Y)$
7. $q(X,Y) \leftarrow p(Y,X), p(Y,X)$
8. $q(X,Y) \leftarrow p(Y,X), p(Y,Y)$

Converting ILP to SAT

Suppose τ = (v=1, int=1):

1. $q(X,Y) \leftarrow p(X,X), q(Y,X)$
2. $q(X,Y) \leftarrow p(X,X), q(Y,Y)$
3. $q(X,Y) \leftarrow p(X,X), q(Y,Z)$
4. $q(X,Y) \leftarrow p(X,X), q(Z,Y)$
5. $q(X,Y) \leftarrow p(X,Y), q(X,X)$
6. $q(X,Y) \leftarrow p(X,Y), q(X,Z)$
7. $q(X,Y) \leftarrow p(X,Y), q(Y,X)$
8. $q(X,Y) \leftarrow p(X,Y), q(Y,Y)$
9. $q(X,Y) \leftarrow p(X,Y), q(Y,Z)$
10. $q(X,Y) \leftarrow p(X,Y), q(Z,X)$
11. $q(X,Y) \leftarrow p(X,Y), q(Z,Y)$
12. $q(X,Y) \leftarrow p(X,Y), q(Z,Z)$
13. $q(X,Y) \leftarrow p(X,Z), q(Y,X)$
14. $q(X,Y) \leftarrow p(X,Z), q(Y,Y)$
15. $q(X,Y) \leftarrow p(X,Z), q(Y,Z)$
16. $q(X,Y) \leftarrow p(X,Z), q(Z,Y)$

Converting ILP to SAT

Let f_i^j indicate whether the *j*'th clause generated by the *i*'th template τ_i is to be included in the program R.

We insist that exactly one flag is turned on for each template τ_i:

$$\forall i \; \exists! j \; f_i^j = \top$$

Converting ILP to SAT

Let f_i^j indicate whether the *j*'th clause generated by the *i*'th template τ_i is to be included in the program R.

Let $cl(\tau_i)[j]$ be the *j*'th clause generated by the *i*'th template τ_i.

Then the rules R are just

$$R = \{cl(\tau_i)[j] \mid f_i^j = \top\}$$

Converting ILP to SAT

- Define a rule template τ as a way of defining a set of clauses
- Define a program template as a set of rule templates
- For each rule template τ, generate the set $cl(\tau)$ of all clauses that satisfy the template
- Introduce a boolean flag for each generated clause, indicating whether it is “on” or “off”
- Now the induction problem has been transformed into a satisfiability problem: find an assignment to the flags such that the set of clauses that are “on” together entail the positive examples and do not entail the negative examples

Restricting the Set of Rules

Restrictions we make without loss of generality:

- We insist that all clauses have exactly two atoms in the body
- We insist that each predicate is defined by exactly two clauses

Other restrictions:

- We do not allow any constants in rules
- We limit ourselves to nullary, unary, and binary predicates

Overview

1. What is Neural Program Synthesis?
2. Learning logic programs
3. How to learn logic programs from examples
4. **Differentiable Inductive Logic Programming**
5. Experiments
6. Related Work

4. Differentiable Logic Programming

The Approach

So far, I have described a simple approach to program synthesis that converts it into a satisfiability problem.

Next, we replace the non-differentiable operations on discrete values with differentiable operations on continuous values.

∂ILP

∂ILP uses a differentiable model of forward chaining inference.

The weights represent a probability distribution over clauses.

We use SGD to minimise the log-loss.

We extract a readable program from the weights.

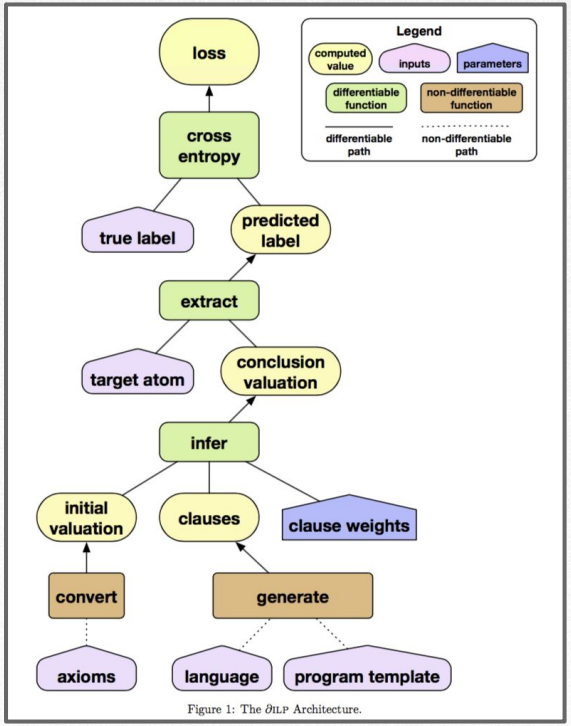


Figure 1: The ∂ILP Architecture.

∂ILP

A **valuation** is a vector in $[0,1]^n$

It maps each of *n* ground atoms to [0,1].

A valuation represents how likely it is that each of the ground atoms is true.

G	$\mathbf{a}_0$
$p(a)$	0.0
$p(b)$	0.0
$q(a)$	0.1
$q(b)$	0.3
$\bot$	0.0

∂ILP

Each clause *c* is compiled into a function on valuations:

$$F_c : [0,1]^n \to [0,1]^n$$

For example:

$$p(X) \leftarrow q(X)$$

G	$\mathbf{a_0}$	$\mathcal{F}_c(\mathbf{a_0})$
$p(a)$	0.0	0.1
$p(b)$	0.0	0.3
$q(a)$	0.1	0.0
$q(b)$	0.3	0.0
$\bot$	0.0	0.0

∂ILP

We combine the clauses' valuations using a weighted sum:

$$b_t = \sum_c F_c(a_t) \frac{e^{W[c]}}{\sum_{c'} e^{W[c']}}$$

We amalgamate the previous valuation with the new clauses' valuation:

$$a_{t+1} = a_t + b_t - a_t \cdot b_t$$

We unroll the network for *T* steps of forward-chaining inference, generating:

$$a_0, a_1, a_2, a_3, \ldots, a_T$$

∂ILP

∂ILP uses a differentiable model of forward chaining inference.

The weights represent a probability distribution over clauses.

We use SGD to minimise the log-loss.

We extract a readable program from the weights.

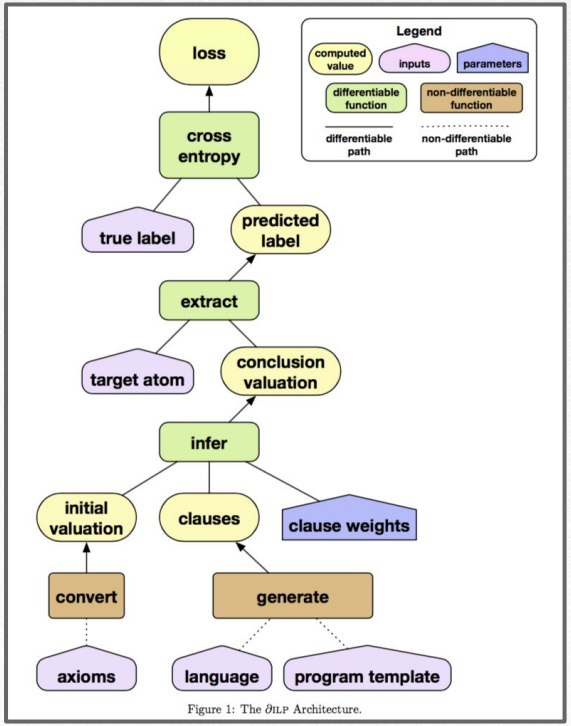


Figure 1: The ∂ILP Architecture.

∂ILP

Each clause *c* is compiled into a function on valuations:

$$F_c : [0,1]^n \to [0,1]^n$$

For example:

$$p(X) \leftarrow q(X)$$

G	$\mathbf{a_0}$	$\mathcal{F}_c(\mathbf{a_0})$
$p(a)$	0.0	0.1
$p(b)$	0.0	0.3
$q(a)$	0.1	0.0
$q(b)$	0.3	0.0
$\bot$	0.0	0.0

∂ILP

Assume that each clause has two atoms in the body. For example:

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$

We calculate, for each ground atom, the pairs of ground atoms that contribute to its truth:

$$r(a,a) : \{(p(a,a), q(a,a)), (p(a,b), q(b,a))\}$$
$$r(a,b) : \{(p(a,a), q(a,b)), (p(a,b), q(b,b))\}$$
$$r(b,a) : \{(p(b,a), q(a,a)), (p(b,b), q(b,a))\}$$
$$r(b,b) : \{(p(b,a), q(a,b)), (p(b,b), q(b,b))\}$$

∂ILP

Given our rule:

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$

We convert the pairs of atoms into pairs of indices:

k	γ_k	x_k
0	$\bot$	{}
1	$p(a,a)$	{}
2	$p(a,b)$	{}
3	$p(b,a)$	{}
4	$p(b,b)$	{}

k	γ_k	x_k
5	$q(a,a)$	{}
6	$q(a,b)$	{}
7	$q(b,a)$	{}
8	$q(b,b)$	{}

k	γ_k	x_k
9	$r(a,a)$	{(1,5), (2, 7)}
10	$r(a,b)$	{(1, 6), (2, 8)}
11	$r(b,a)$	{(3, 5), (4, 7)}
12	$r(b,b)$	{(3, 6), (4, 8)}

∂ILP

We convert:

$$
\begin{aligned}
r(a,a) &: \{(p(a,a), q(a,a)),\ (p(a,b), q(b,a))\} \\
r(a,b) &: \{(p(a,a), q(a,b)),\ (p(a,b), q(b,b))\} \\
r(b,a) &: \{(p(b,a), q(a,a)),\ (p(b,b), q(b,a))\} \\
r(b,b) &: \{(p(b,a), q(a,b)),\ (p(b,b), q(b,b))\}
\end{aligned}
$$

into:

k	γ_k	x_k
0	$\bot$	{}
1	$p(a,a)$	{}
2	$p(a,b)$	{}
3	$p(b,a)$	{}
4	$p(b,b)$	{}

k	γ_k	x_k
5	$q(a,a)$	{}
6	$q(a,b)$	{}
7	$q(b,a)$	{}
8	$q(b,b)$	{}

k	γ_k	x_k
9	$r(a,a)$	{(1,5), (2, 7)}
10	$r(a,b)$	{(1, 6), (2, 8)}
11	$r(b,a)$	{(3, 5), (4, 7)}
12	$r(b,b)$	{(3, 6), (4, 8)}

∂ILP

We convert:

k	γ_k	x_k
0	$\bot$	{}
1	$p(a,a)$	{}
2	$p(a,b)$	{}
3	$p(b,a)$	{}
4	$p(b,b)$	{}

k	γ_k	x_k
5	$q(a,a)$	{}
6	$q(a,b)$	{}
7	$q(b,a)$	{}
8	$q(b,b)$	{}

k	γ_k	x_k
9	$r(a,a)$	{(1,5), (2, 7)}
10	$r(a,b)$	{(1, 6), (2, 8)}
11	$r(b,a)$	{(3, 5), (4, 7)}
12	$r(b,b)$	{(3, 6), (4, 8)}

into a tensor of shape *n * w * 2*:

k	γ_k	$\mathbf{X}[k]$
0	$\bot$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
1	$p(a,a)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
2	$p(a,b)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
3	$p(b,a)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
4	$p(b,b)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$

k	γ_k	$\mathbf{X}[k]$
5	$q(a,a)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
6	$q(a,b)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
7	$q(b,a)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$
8	$q(b,b)$	$\begin{bmatrix}(0,0)\\(0,0)\end{bmatrix}$

k	γ_k	$\mathbf{X}[k]$
9	$r(a,a)$	$\begin{bmatrix}(1,5)\\(2,7)\end{bmatrix}$
10	$r(a,b)$	$\begin{bmatrix}(1,6)\\(2,8)\end{bmatrix}$
11	$r(b,a)$	$\begin{bmatrix}(3,5)\\(4,7)\end{bmatrix}$
12	$r(b,b)$	$\begin{bmatrix}(3,6)\\(4,8)\end{bmatrix}$

∂ILP

We split our tensor X into two matrices of shape *n* * *w* :

$$\mathbf{X}_1 = \mathbf{X}[:, :, 0] \qquad \mathbf{X}_2 = \mathbf{X}[:, :, 1]$$

We gather up the results:

$$\mathbf{Y}_1 = \text{gather}_2(\mathbf{a}, \mathbf{X}_1) \qquad \mathbf{Y}_2 = \text{gather}_2(\mathbf{a}, \mathbf{X}_2)$$

We take the element-wise product:

$$\mathbf{Z} = \mathbf{Y}_1 \odot \mathbf{Y}_2$$

Here, **Z** is of shape *n* * *w*. Now we take the max across the second dimension:

$$F_c(\mathbf{a}) = \mathbf{a}' \text{ where } \mathbf{a}'[k] = \max(\mathbf{Z}[k, :])$$

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
1	$p(a, a)$	1.0	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
2	$p(a, b)$	0.9	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
3	$p(b, a)$	0.0	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
4	$p(b, b)$	0.0	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
5	$q(a, a)$	0.1	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
6	$q(a, b)$	0.0	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
7	$q(b, a)$	0.2	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
8	$q(b, b)$	0.8	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	$[0 \quad 0]$	0.00
9	$r(a, a)$	0.0	$[1 \quad 2]$	$[5 \quad 7]$	$[1.0 \quad 0.9]$	$[0.1 \quad 0.2]$	$[0.1 \quad 0.18]$	0.18
10	$r(a, b)$	0.0	$[1 \quad 2]$	$[6 \quad 8]$	$[1.0 \quad 0.9]$	$[0 \quad 0.8]$	$[0 \quad 0.72]$	0.72
11	$r(b, a)$	0.0	$[3 \quad 4]$	$[5 \quad 7]$	$[0 \quad 0]$	$[0.1 \quad 0.2]$	$[0 \quad 0]$	0.00
12	$r(b, b)$	0.0	$[3 \quad 4]$	$[6 \quad 8]$	$[0 \quad 0]$	$[0 \quad 0.8]$	$[0 \quad 0]$	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

$$r(X, Y) \leftarrow p(X, Z), q(Z, Y)$$
$$r(a, b) \leftarrow p(a, b), q(b, b)$$

k	γ_k	$\mathbf{a}[k]$	$\mathbf{X}_1[k]$	$\mathbf{X}_2[k]$	$\mathbf{Y}_1[k]$	$\mathbf{Y}_2[k]$	$\mathbf{Z}[k]$	$F_c(\mathbf{a})[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	$p(a, a)$	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	$p(a, b)$	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	$p(b, a)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	$p(b, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	$q(a, a)$	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	$q(a, b)$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	$q(b, a)$	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	$q(b, b)$	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	$r(a, a)$	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	$r(a, b)$	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	$r(b, a)$	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	$r(b, b)$	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

Overview

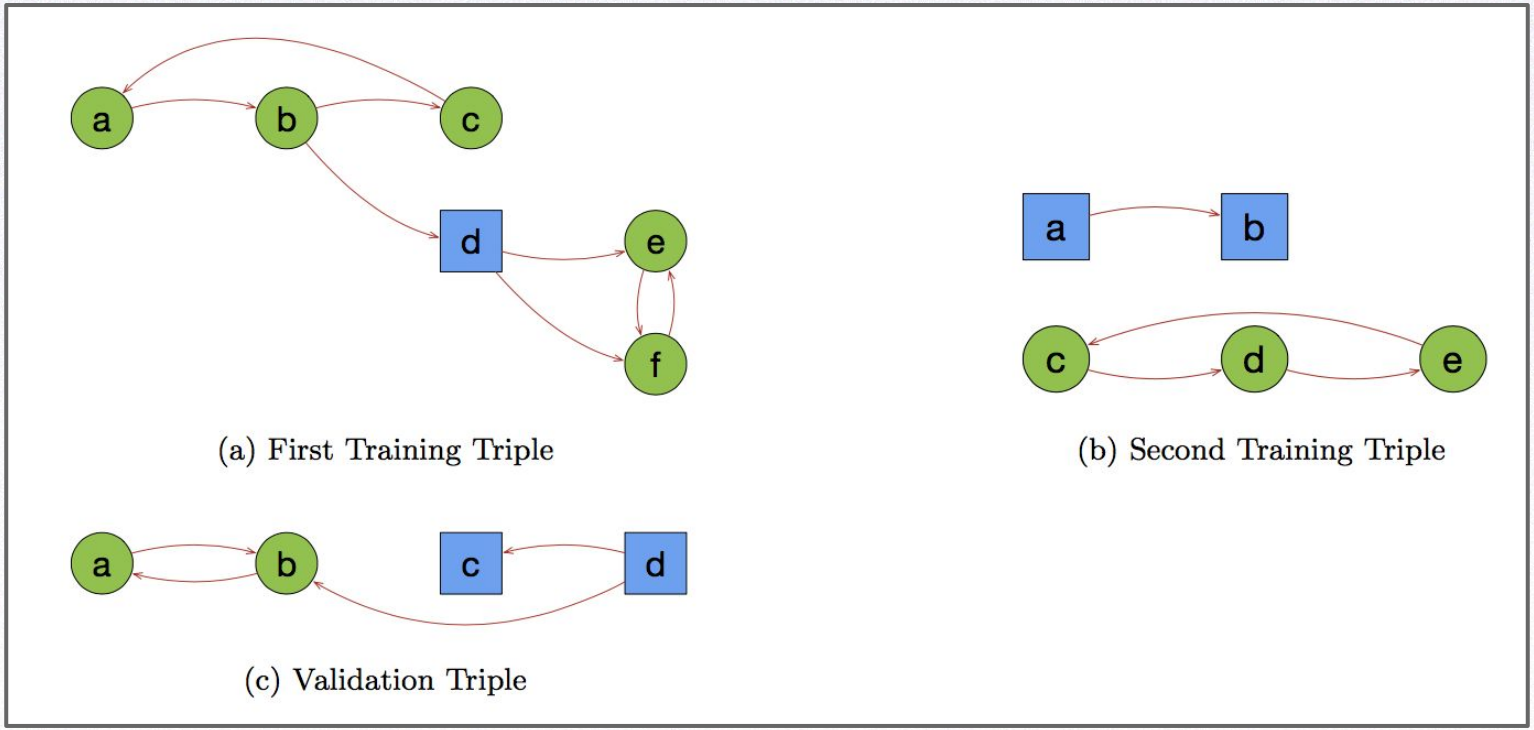
1. What is Neural Program Synthesis?
2. Learning logic programs
3. How to learn logic programs from examples
4. Differentiable Inductive Logic Programming
5. **Experiments**
6. Related Work

∂ILP Experiments

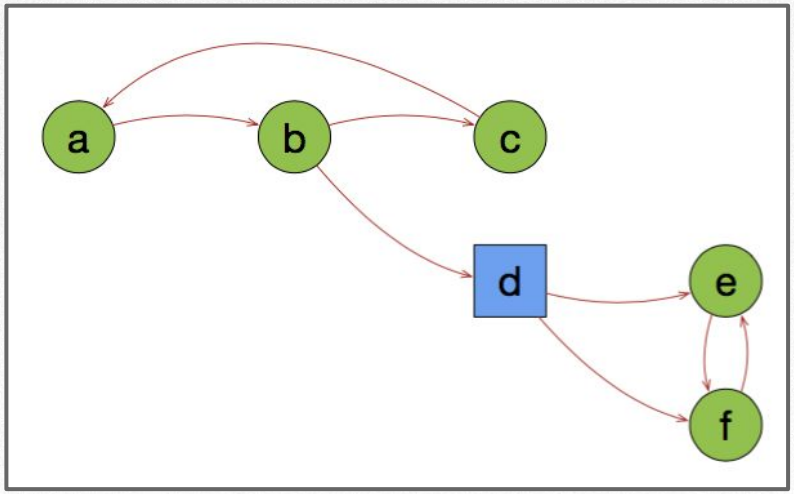
Domain	Task	$\lvert P_i \rvert$	Recursive	Metagol Performance	∂ILP Performance
Arithmetic	Predecessor	1	No	✓	✓
Arithmetic	Even / odd	2	Yes	✓	✓
Arithmetic	Even / succ2	2	Yes	✓	✓
Arithmetic	Less than	1	Yes	✓	✓
Arithmetic	Fizz	3	Yes	✓	✓
Arithmetic	Buzz	2	Yes	✓	✓
Lists	Member	1	Yes	✓	✓
Lists	Length	2	Yes	✓	✓
Family Tree	Son	2	No	✓	✓
Family Tree	Grandparent	2	No	✓	✓
Family Tree	Husband	2	No	✓	✓
Family Tree	Uncle	2	No	✓	✓
Family Tree	Relatedness	1	No	×	✓
Family Tree	Father	1	No	✓	✓
Graphs	Undirected Edge	1	No	✓	✓
Graphs	Adjacent to Red	2	No	✓	✓
Graphs	Two Children	2	No	✓	✓
Graphs	Graph Colouring	2	Yes	✓	✓
Graphs	Connectedness	1	Yes	×	✓
Graphs	Cyclic	2	Yes	×	✓

Table 2: A Comparison Between ∂ILP and Metagol on 20 Symbolic Tasks

Example Task: Graph Cyclicity



Example Task: Graph Cyclicity




```
cycle(X) ← pred(X, X).

pred(X, Y) ← edge(X, Y).

pred(X, Y) ← edge(X, Z), pred(Z, Y)
```

Example: Fizz-Buzz

1 ↦ 1
2 ↦ 2
3 ↦ Fizz
4 ↦ 4
5 ↦ Buzz
6 ↦ Fizz
7 ↦ 7
8 ↦ 8
9 ↦ Fizz
10 ↦ Buzz

11 ↦ 11
12 ↦ Fizz
13 ↦ 13
14 ↦ 14
15 ↦ Fizz+Buzz
16 ↦ 16
17 ↦ 17
18 ↦ Fizz
19 ↦ 19
20 ↦ Buzz

Example: Fizz

```
fizz(X) ← zero(X).

fizz(X) ← fizz(Y), pred1(Y, X).

pred1(X, Y) ← succ(X, Z), pred2(Z, Y).

pred2(X, Y) ← succ(X, Z), succ(Z, Y).
```

Example: Fizz

```
fizz(X) ← zero(X).

fizz(X) ← fizz(Y), pred1(Y, X).

pred1(X, Y) ← succ(X, Z), pred2(Z, Y).

pred2(X, Y) ← succ(X, Z), succ(Z, Y).
```

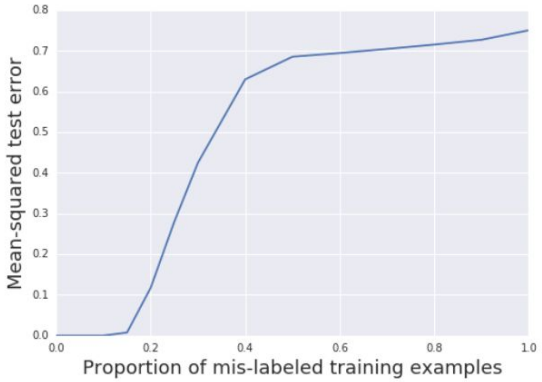
Example: Buzz

```
buzz(X) ← zero(X).

buzz(X) ← buzz(Y), pred3(Y, X).

pred3(X, Y) ← pred1(X, Z), pred2(Z, Y).
```

```
pred1(X, Y) ← succ(X, Z), pred2(Z, Y).

pred2(X, Y) ← succ(X, Z), succ(Z, Y).
```

Mis-labelled Data

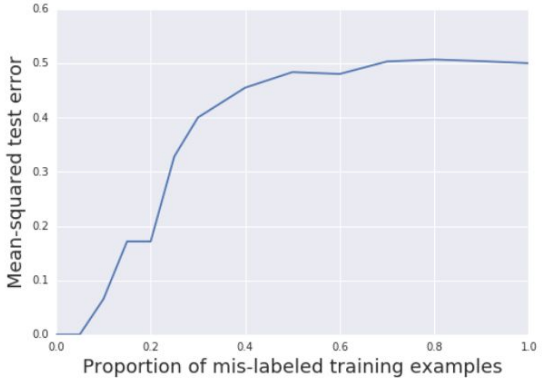
- If Symbolic Program Synthesis is given a single mis-labelled piece of training data, it **fails catastrophically**.
- We tested ∂ILP with mis-labelled data.
- We mis-labelled a certain proportion ρ of the training examples.
- We ran experiments for different values of ρ = 0.0, 0.1, 0.2, 0.3, ...



(a) Predecessor

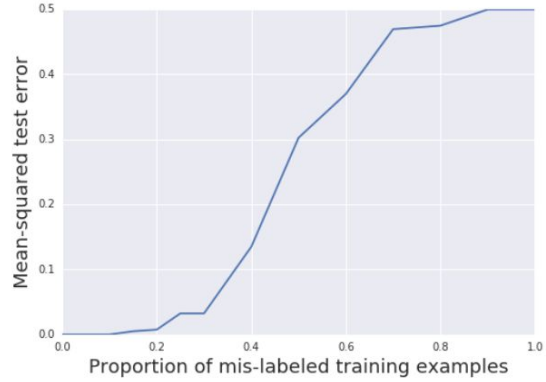
(b) Less-Than

(c) List: Member



(d) Family Tree: Son

(e) Graph: Connectedness



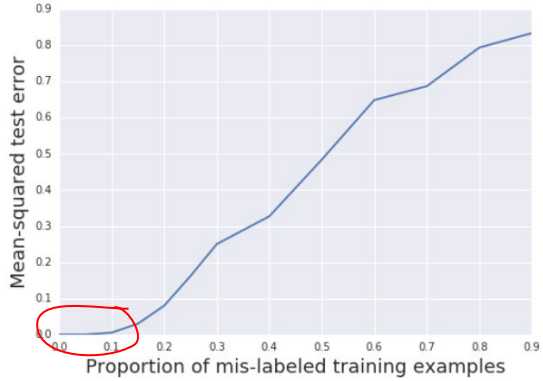
(f) Graph: Undirected Edge

(a) Predecessor

(b) Less-Than

(c) List: Member

(d) Family Tree: Son



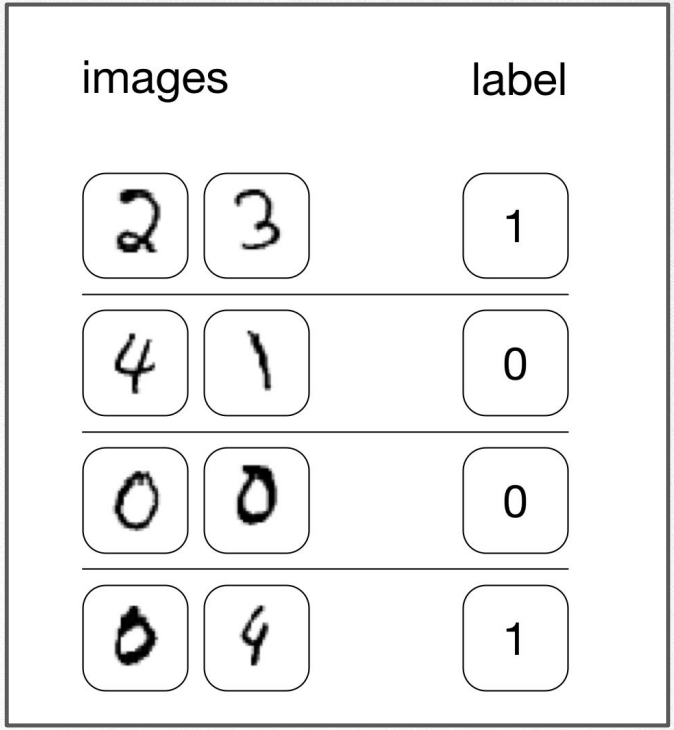
(e) Graph: Connectedness

(f) Graph: Undirected Edge

Example: Learning Rules from Ambiguous Data

Your system observes:

- a pair of images
- a label indicating whether the left image is *less than* the right image



Example: Learning Rules from Ambiguous Data

Your system observes:

- a pair of images
- a label indicating whether the left image is *less than* the right image

Two forms of generalisation:
It must decide if the relation holds for held-out images, and also *held-out pairs of digits*.

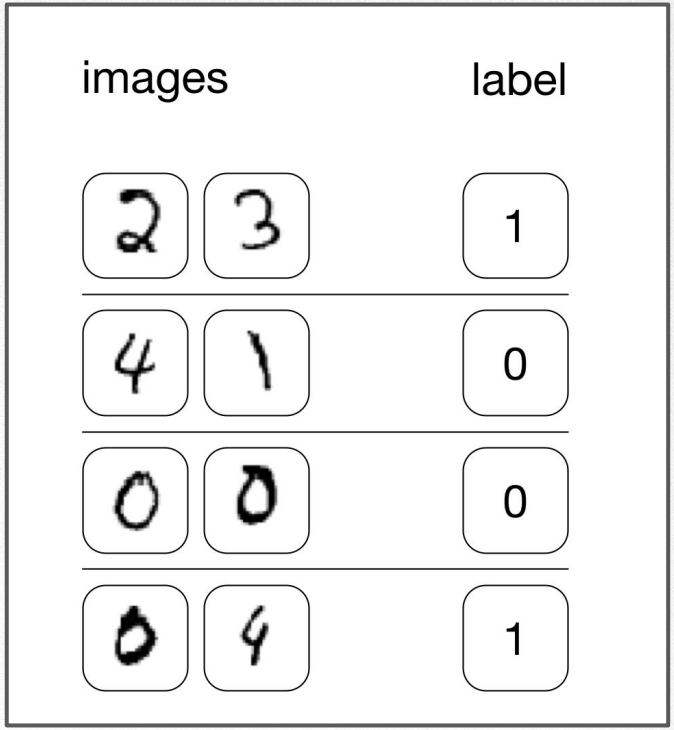
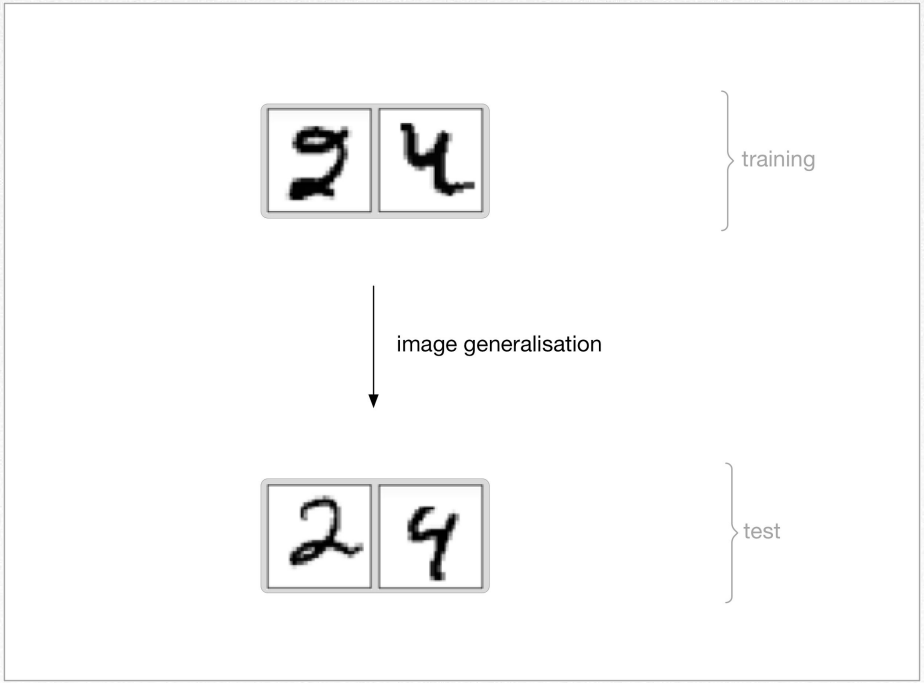
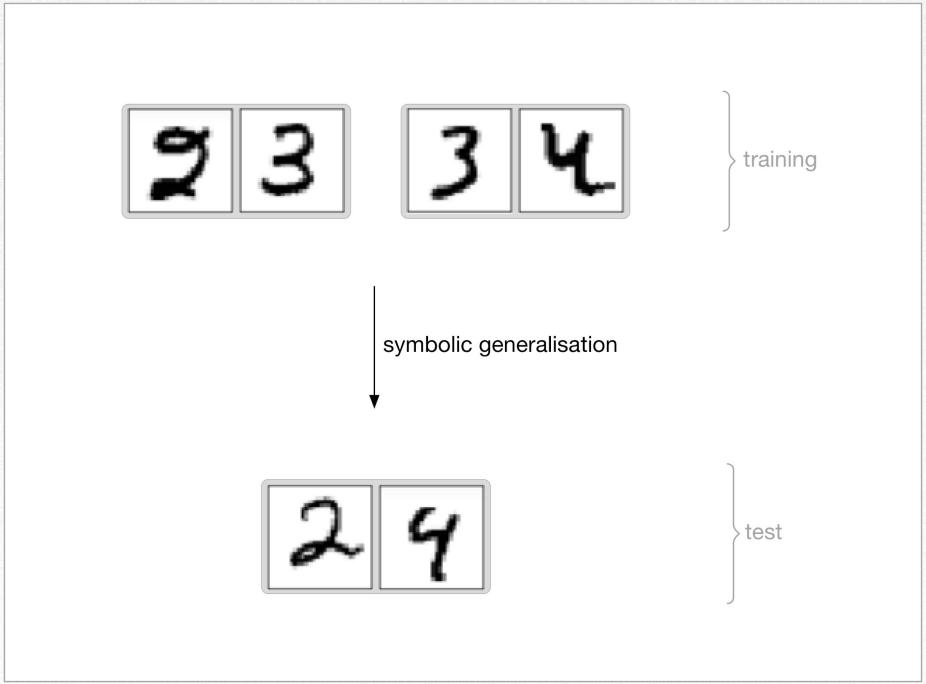


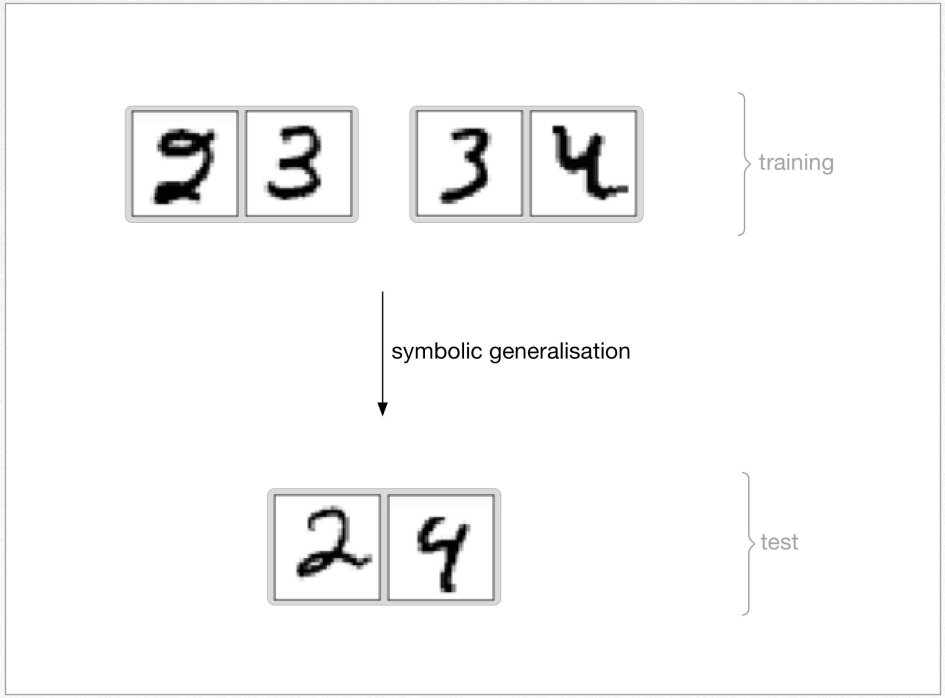
Image Generalisation



Symbolic Generalisation



Symbolic Generalisation



NB it has never seen *any* examples of 2 < 4 in training

Symbolic Generalisation

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5	1 < 6	1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7	2 < 8	2 < 9
				3 < 4	3 < 5	3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	7 < 9
									8 < 9

Symbolic Generalisation

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5	1 < 6	1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7	2 < 8	2 < 9
				3 < 4	3 < 5	3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	7 < 9
									8 < 9

Symbolic Generalisation

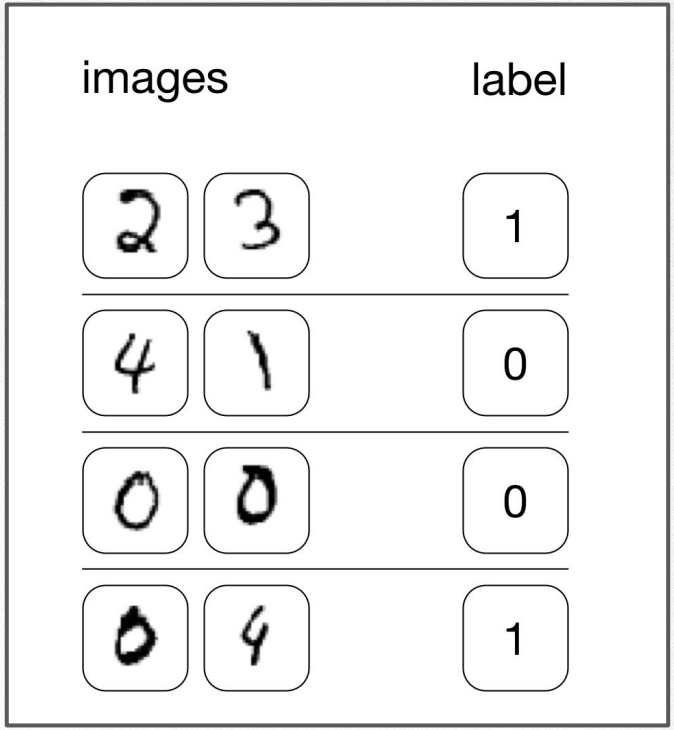
	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5		1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		2 < 9
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	
									8 < 9

Example: Less Than on MNIST Images

Your system observes:

- a pair of images
- a label indicating whether the left image is *less than* the right image

Two forms of generalisation:
It must decide if the relation holds for held-out images, and also *held-out pairs of digits*.



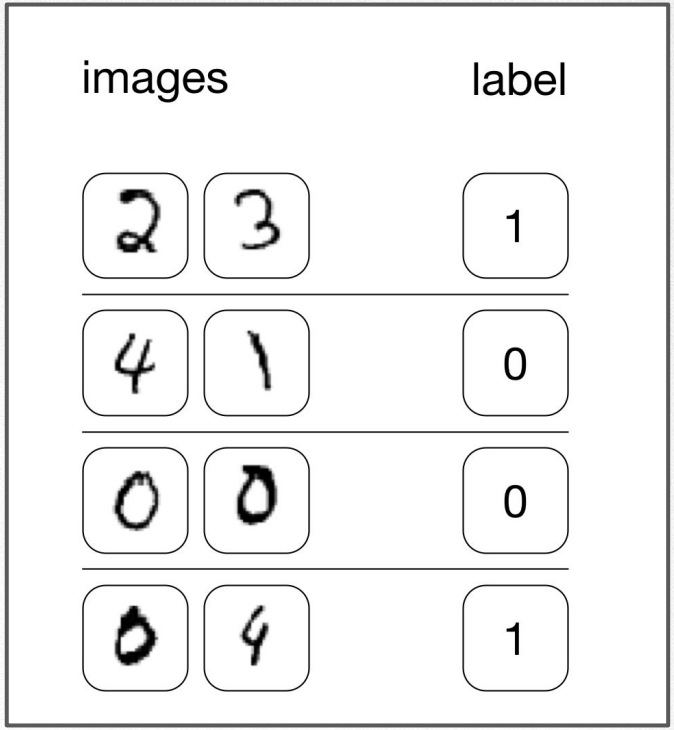
MLP Baseline

We created a baseline MLP to solve this task.

The output of the conv-net for the two images is a vector of (20) logits.

We added a hidden layer, produced a single output, and trained on cross-entropy loss.

The MLP baseline can solve this task easily.



Example: Less Than

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5	1 < 6	1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7	2 < 8	2 < 9
				3 < 4	3 < 5	3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	7 < 9
									8 < 9

Example: Less Than

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5	1 < 6	1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7	2 < 8	2 < 9
				3 < 4	3 < 5	3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	7 < 9
									8 < 9

Example: Less Than

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5		1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		2 < 9
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	
									8 < 9

Example: Less Than

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7	0 < 8	0 < 9
		1 < 2	1 < 3	1 < 4	1 < 5		1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		2 < 9
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	4 < 9
						5 < 6	5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	
									8 < 9

Example: Less Than

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7		0 < 9
		1 < 2		1 < 4	1 < 5		1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		2 < 9
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	
							5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	
									8 < 9

Example: Less Than

	0 < 1	0 < 2	0 < 3	0 < 4	0 < 5	0 < 6	0 < 7		0 < 9
		1 < 2		1 < 4	1 < 5		1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		2 < 9
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	
							5 < 7	5 < 8	5 < 9
							6 < 7	6 < 8	6 < 9
								7 < 8	
									8 < 9

Example: Less Than

	0 < 1	0 < 2		0 < 4	0 < 5	0 < 6	0 < 7		0 < 9
		1 < 2		1 < 4			1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	
							5 < 7	5 < 8	5 < 9
									6 < 9
								7 < 8	
									8 < 9

Example: Less Than

	0 < 1	0 < 2		0 < 4	0 < 5	0 < 6	0 < 7		0 < 9
		1 < 2		1 < 4			1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5	2 < 6	2 < 7		
				3 < 4		3 < 6	3 < 7	3 < 8	3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	
							5 < 7	5 < 8	5 < 9
									6 < 9
								7 < 8	
									8 < 9

Example: Less Than

	0 < 1			0 < 4	0 < 5	0 < 6	0 < 7		0 < 9
		1 < 2		1 < 4			1 < 7	1 < 8	1 < 9
			2 < 3	2 < 4	2 < 5		2 < 7		
				3 < 4		3 < 6			3 < 9
					4 < 5	4 < 6	4 < 7	4 < 8	
							5 < 7	5 < 8	5 < 9
									6 < 9
								7 < 8	

∂ILP Learning Less-Than

We made a slight modification to our original architecture:

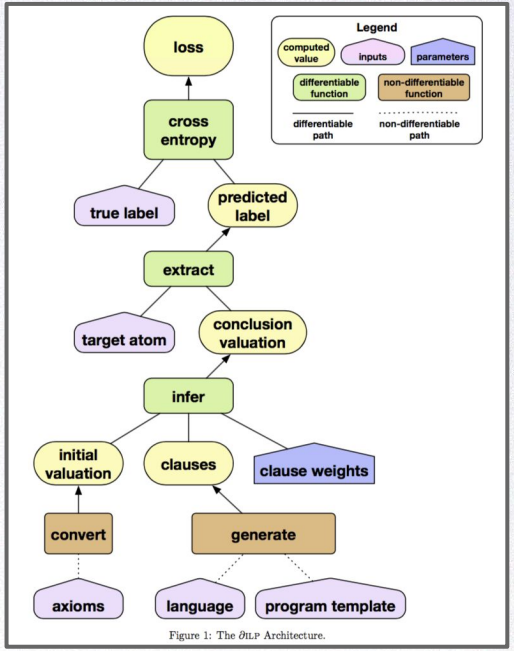


Figure 1: The ∂ILP Architecture.

∂ILP Learning Less-Than

We pre-trained a conv-net to recognise MNIST digits.

We convert the logits of the conv-net into a probability distribution over logical atoms.

Our model is able to solve this task.

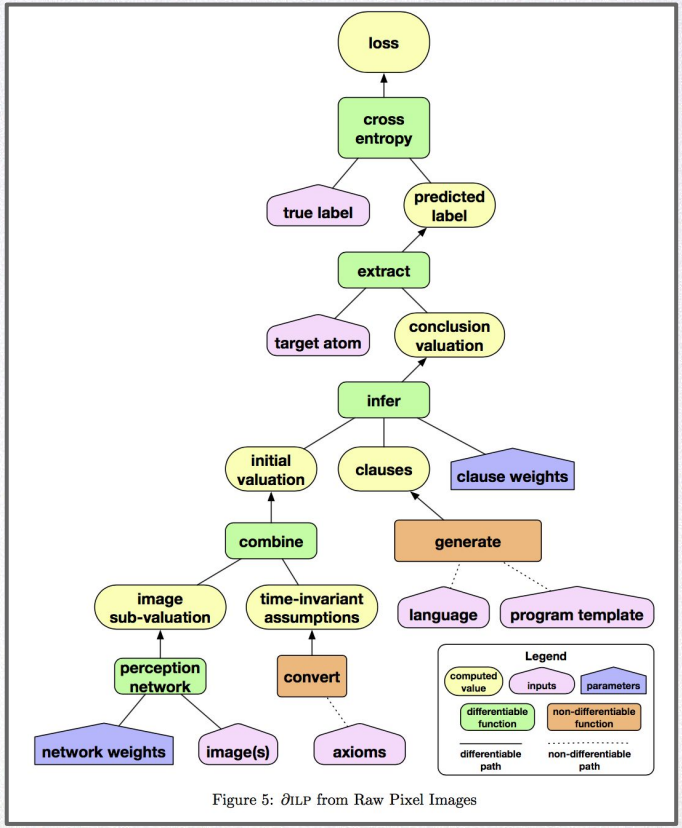
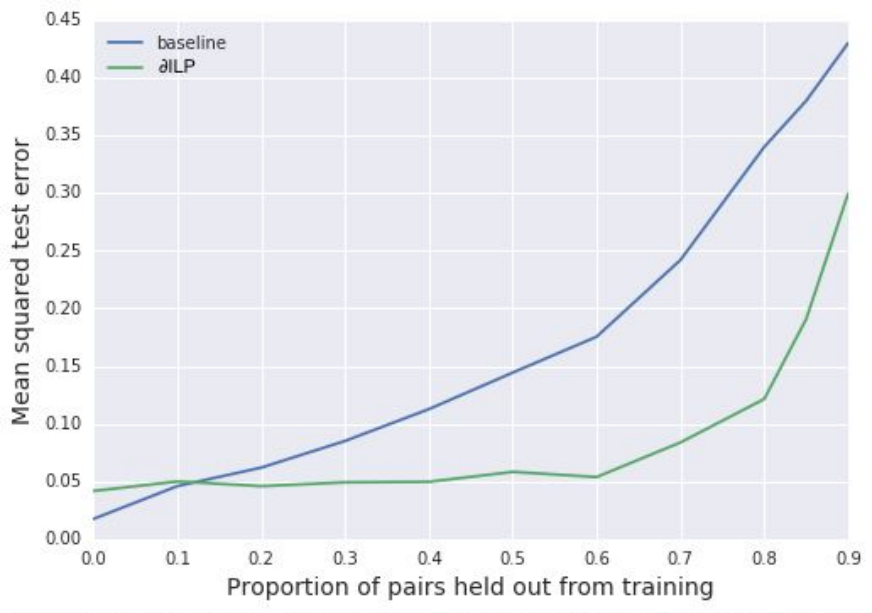


Figure 5: ∂ILP from Raw Pixel Images

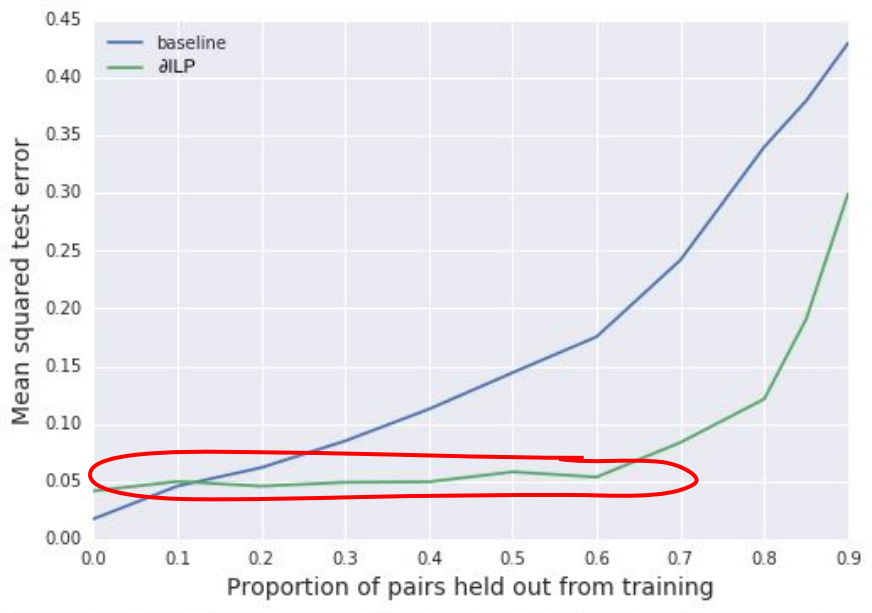
∂ILP Learning Less-Than

```
target() ← image2(X), pred1(X)

pred1(X) ← image1(Y), pred2(Y, X)

pred2(X, Y) ← succ(X, Y)

pred2(X, Y) ← pred2(Z, Y), pred2(X, Z)
```

Comparing ∂ILP with the Baseline



Comparing ∂ILP with the Baseline



Limitations

- Local minima!
- Memory use!

$$2 \cdot n \cdot t \cdot \sum_{i=1}^{|P_i|} |cl(\tau_i^1)| \cdot |cl(\tau_i^2)|$$

$$n = |G| \leq |P| \cdot |C|^2 + 1$$

Overview

1. What is Neural Program Synthesis?
2. Learning logic programs
3. How to learn logic programs from examples
4. Differentiable Inductive Logic Programming
5. Experiments
6. **Related Work**

6. Related Work

Deduction, Abduction, Induction

Let *A* and *A'* be sets of ground atoms and *R* be a set of first-order rules.

Then we can define:

- **Deduction**: given *R* and *A*, find *A'* such that *R*, *A* $\models$ *A'*
- **Abduction**: given *R* and *A'*, find *A* such that *R*, *A* $\models$ *A'*
- **Induction**: given *A* and *A'*, find *R* such that *R*, *A* $\models$ *A'*

First-Order Deduction using Neural Nets

Holldobler, Kalinke, and Storr (1999)

Given a set *G* of ground atoms, they define a function $R : 2^G \rightarrow \mathbb{R}$

$$R(M) = \sum_{g \in M} 4^{-||g||}$$

Given a program *P*, they define a function $f_P : \mathbb{R} \rightarrow \mathbb{R}$ on reals that mirrors the consequence operator:

$$f_P(x) = R(T_P(R^{-1}(x)))$$

First-Order Abduction Using Neural Nets

Serafini and Garcez (2016)

- A constant c is represented by a vector $G(c) \in \mathbb{R}^n$
- A function *f* is represented by $G(f) \in \mathbb{R}^{\text{arity}(f) \times n} \rightarrow \mathbb{R}^n$
- A predicate *p* is represented by $G(p) \in \mathbb{R}^{\text{arity}(f) \times n} \rightarrow [0, 1]$

$$
\begin{aligned}
G(f(t_1, ..., t_m) &= G(f)(G(t_1), ..., G(t_m)) \\
G(p(t_1, ..., t_m) &= G(p)(G(t_1), ..., G(t_m)) \\
G(\neg\phi) &= 1 - G(\phi) \\
G(\phi_1 \vee ... \vee phi_m) &= \mu(G(\phi_1), ..., G(\phi_m))
\end{aligned}
$$

First-Order Induction Using Neural Nets

- “Neural Theorem Prover”: Rocktaschel and Riedel (2016)
- “NeuralLP”: Yang, Yang, and Cohen (2017)
- “∂ILP”: Evans and Grefenstette (2017)
- “DiffLog”: Raghothaman et al (2018)

Conclusion

Neural Program Synthesis aims to combine the advantages of Symbolic Program Synthesis with the advantages of Neural Program Induction:

- It has low *sample complexity*
- It can learn *interpretable* and *general* rules
- It is robust to *mislabelled* data
- It can handle *ambiguous* input
- It can be integrated and trained jointly within larger neural systems/agents

Conclusion

I have argued that Datalog is an excellent target language for program synthesis because it is:

- referentially transparent
- highly expressive

Conclusion

I have described one way for a neural network to induce Datalog programs from examples.

- Convert the induction problem into a SAT problem
- Neuralise it, replacing discrete operations with differentiable operations
- Train using gradient descent