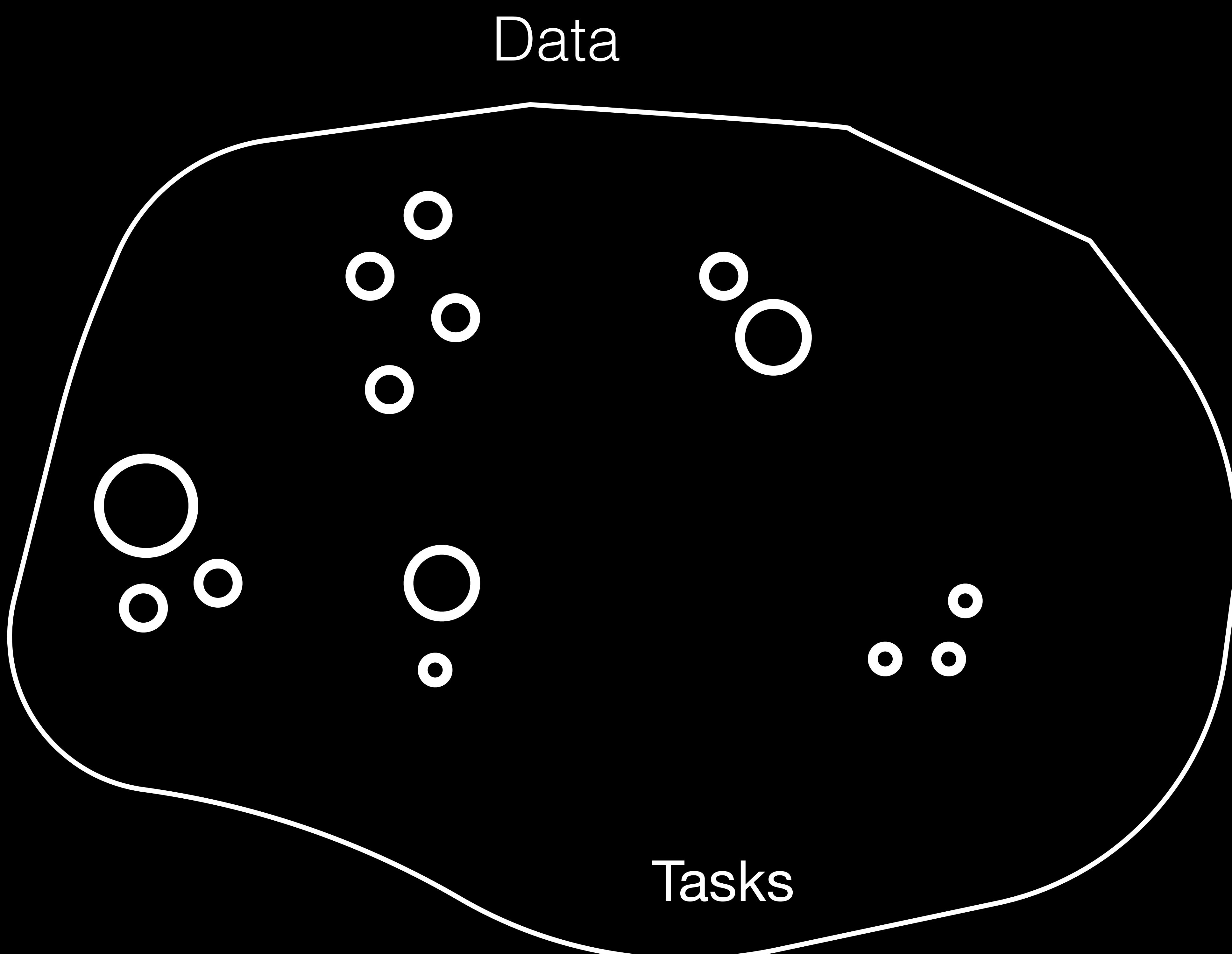
In Search of Strong Generalization:

What can we learn from programming languages?

Danny Tarlow
Researcher
Microsoft Research

Joint work with Alex Gaunt, Marc Brockschmidt, Yujia Li, Richard Zemel, Nate Kushman, Chengtao Li.

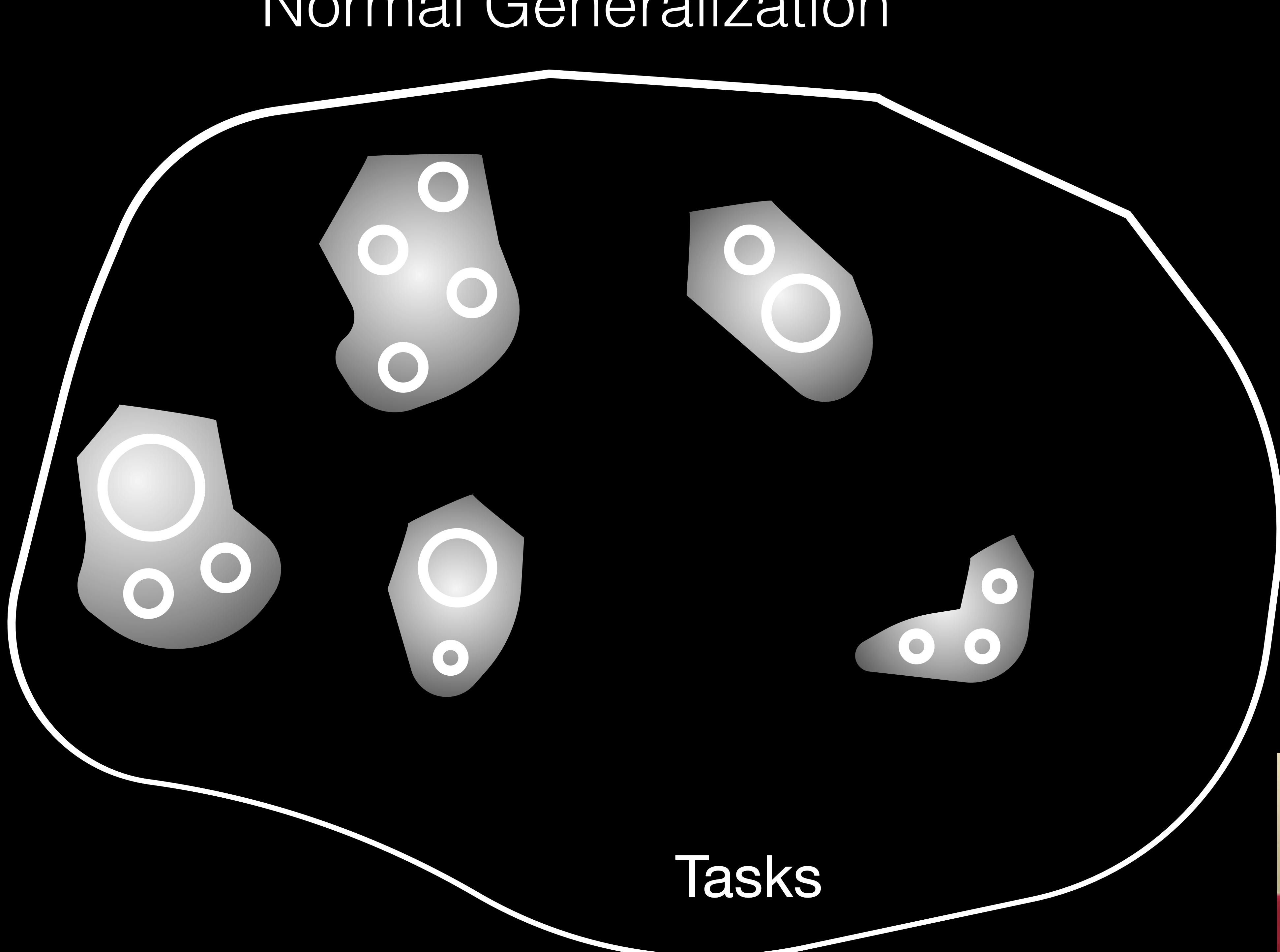
Motivation - Strong Generalization



http://www.scriptol.com/robotics/robots/household.php

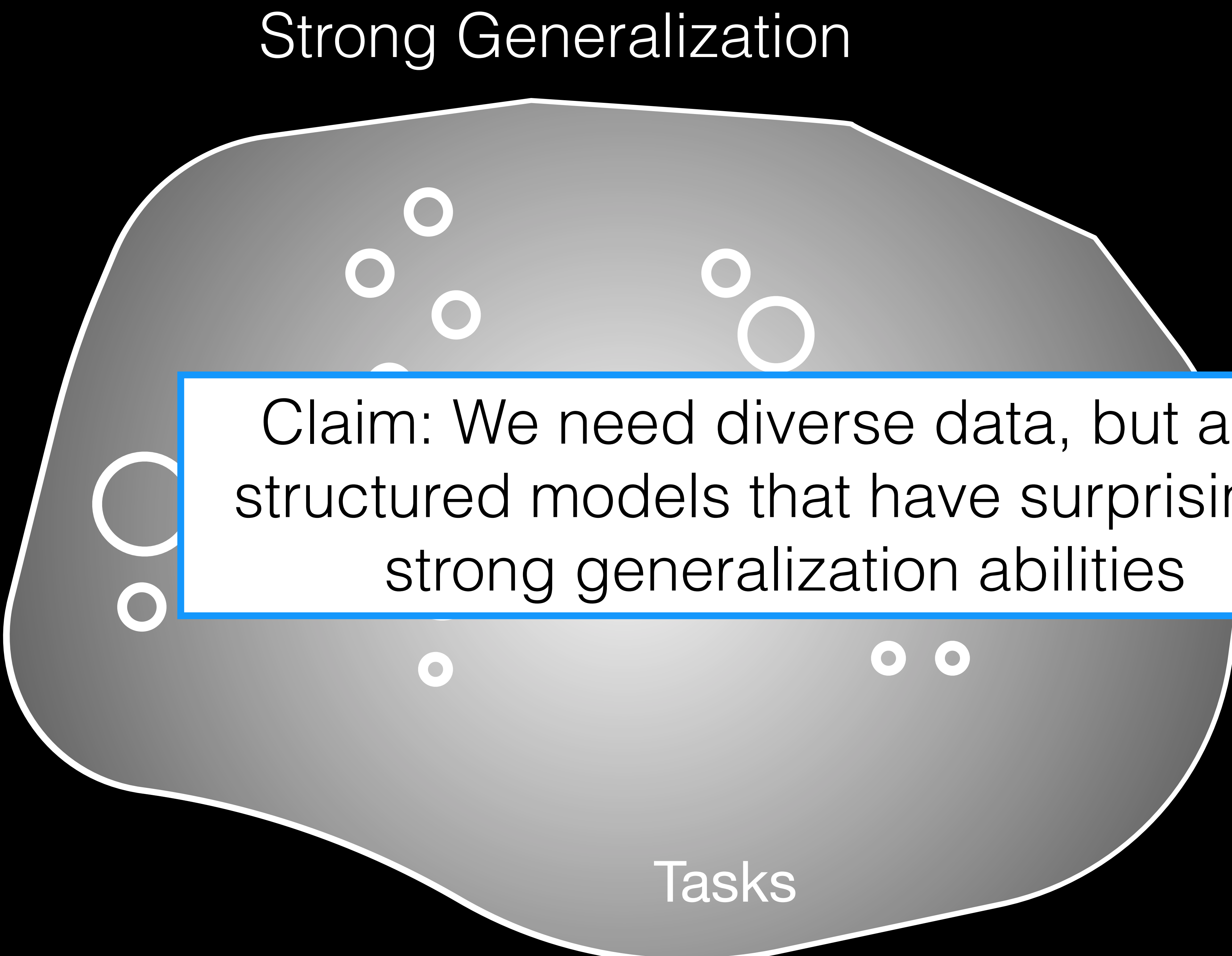
Motivation - Strong Generalization

Normal Generalization



http://www.scriptol.com/robotics/robots/household.php

Motivation - Strong Generalization



Claim: We need diverse data, but also structured models that have surprisingly strong generalization abilities

http://www.scriptol.com/robotics/robots/household.php

This Talk

1. Strong generalization
   - By building neural network models inspired by *natural source code*[1]

2. Weak supervision
   - By differentiating through approximate marginalization algorithms

---

Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., 2012, June. On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 837-847). IEEE.

This Talk

1. **Strong generalization**
   - **By building neural network models inspired by *natural source code*[1]**

2. Weak supervision
   - By differentiating through approximate marginalization algorithms

---

Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., 2012, June. On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 837-847). IEEE.

Inductive Bias of Source Code

Write a short natural program to add a and b:

```
def add(a, b):
  carry = 0
  for i in range(len(a)):
     cur = a[i] + b[i] + carry
     result.append(cur % 10)
     carry = cur / 10
  result.append(carry)
  return result
```

Inductive Bias of Source Code

Write a short natural program to add a and b:
Now modify so that it works on lengths 1-20 but not >20.

```
def add(a, b):
  carry = 0
  for i in range(len(a)):
     cur = a[i] + b[i] + carry
     result.append(cur % 10)
     carry = cur / 10
  result.append(carry)
  return result
```

Natural Source Code → Inductive Bias → Strong Generalization

Claim / Aspiration:

Programming languages are designed to compactly express the computations that people want to perform, and to make it easy for humans to reason about complex computations.

Natural source code induces a prior over natural computations, which we can leverage as inductive bias in machine learning models to achieve strong generalization.

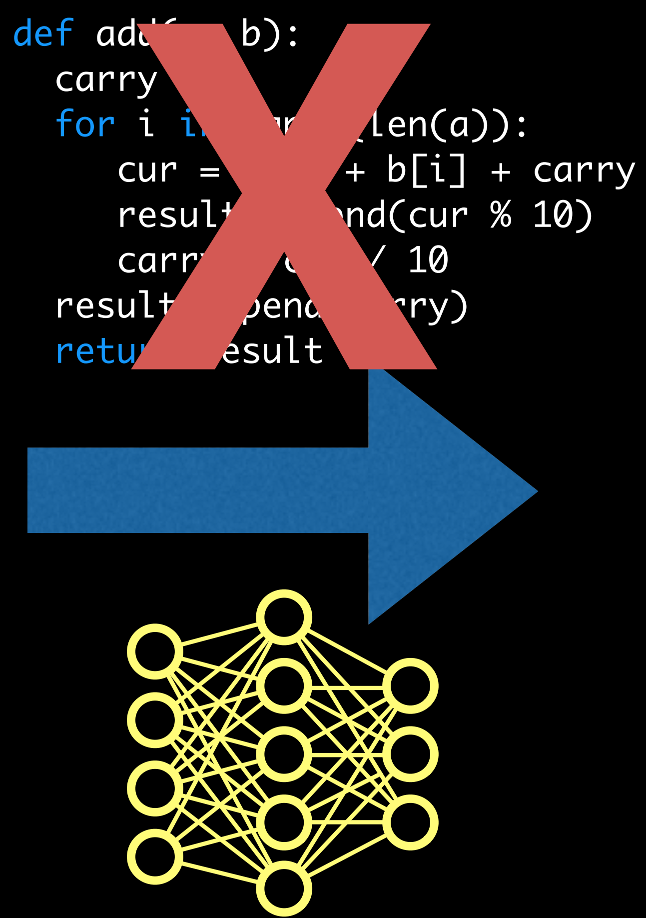
Compared to Kolmogorov complexity, Solomonoff induction, AIXI, etc: here we care about the constants and specific details of, e.g., how modern programming languages represent algorithms. Think python, not binary encoding of programs.

Properties of natural programs:

- algorithm structure often (but not always) invariant to data values
- structured loops (`for` loops vs `goto` spaghetti), recursion
- locality / sparsity in accessing & modifying data
- modularity / compositionality / abstraction
- organized into reusable libraries, object-oriented programming
- …

Source Code Inductive Bias Not Always Favorable

Write a short natural program to classify an image



Tusker

Goal

Source code inductive bias for strong generalization

+

Neural networks to handle rich data types

Other examples of encoding algorithmic structure into model

Hinton, G.E., 1986, August. Learning Distributed Representations of Concepts. In *Conference of the Cognitive Science Society*.

…

Bottou, L., 2014. From Machine Learning to Machine Reasoning. *Machine Learning*, 94(2), pp.133-149.

Graves, A., Wayne, G. and Danihelka, I., 2014. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*.

Duvenaud, D.K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A. and Adams, R.P., 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*.

Reed, S. and de Freitas, N., 2015. Neural Programmer-Interpreters. In *International Conference on Learning Representations*.

Kaiser, Ł. and Sutskever, I., 2015. Neural GPUs learn algorithms. In *International Conference on Learning Representations*.

Andreas, J., Rohrbach, M., Darrell, T. and Klein, D., 2016. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J. and Badia, A.P., 2016. Hybrid computing using a neural network with dynamic external memory. Nature.

Andrychowicz, M. and Kurach, K., 2016. Learning Efficient Algorithms with Hierarchical Attentive Memory. arXiv preprint arXiv:1602.03218.

Tamar, A., Levine, S. and Abbeel, P., 2016. Value Iteration Networks. In *Advances in Neural Information Processing Systems*.

Sukhbaatar, S., Szlam, A. and Fergus, R., 2016. Learning Multiagent Communication with Backpropagation. In *Advances in Neural Information Processing Systems.*

Nowak, A., Bruna, J., 2016. Divide and Conquer with Neural Networks. Submitted to ICLR 2017.

How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures

   - Graph algorithms → Graph Neural Networks

b. Learn models represented as source code

   - Extend differentiable interpreters to learn neural network subroutines

How to combine source code bias and neural nets?

a. **Build models around specific algorithmic structures**

   - **Graph algorithms → Graph Neural Networks**

b. Learn models represented as source code

   - Extend differentiable interpreters to learn neural network subroutines

Example Graph Algorithm: Bellman Ford

Computes shortest paths from `source` to all other vertices — works for *any* graph structure

```
for v in vertices:
    predecessor[v] = null
distance[source] = 0
```

Initialize node representations

```
for i in range(len(vertices)) - 1:
  for
    if
                                                    assume edge costs=1
      predecessor[v] = u
```

Repeatedly update node representations as function of neighbor representations

Decode solution from node representations

Graph Neural Networks

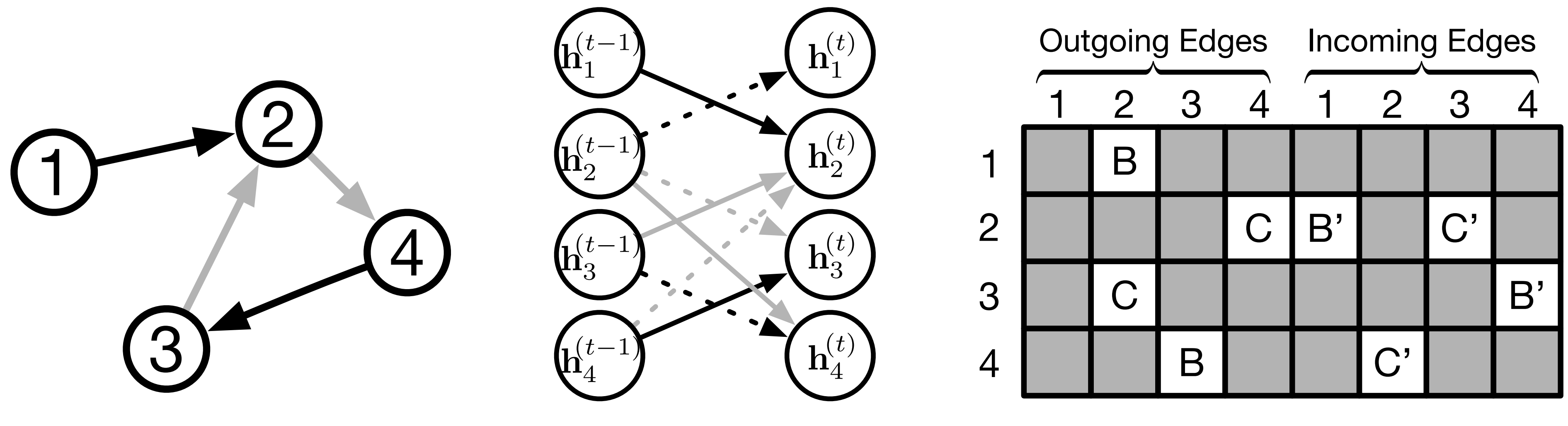
Initialize node representations (hidden vector per node)

Repeatedly update node representations as learned function of neighbor representations and edge types

Decode prediction from node representations

End-to-end differentiable, train by SGD

Gated Graph Sequence Neural Networks



(a) (b) (c) $\mathbf{A} = \left[\mathbf{A}^{(\text{out})}, \mathbf{A}^{(\text{in})}\right]$

Figure 1: (a) Example graph. Color denotes edge types. (b) Unrolled one timestep. (c) Parameter tying and sparsity in recurrent matrix. Letters denote edge types with B' corresponding to the reverse edge of type B. B and B' denote distinct parameters.

$$\mathbf{h}_v^{(1)} = [\boldsymbol{x}_v^\top, \mathbf{0}]^\top \tag{1}$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{v:}^\top \left[\mathbf{h}_1^{(t-1)\top} \dots \mathbf{h}_{|\mathcal{V}|}^{(t-1)\top}\right]^\top + \mathbf{b} \tag{2}$$

$$\mathbf{z}_v^t = \sigma\left(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}\right) \tag{3}$$

$$\mathbf{r}_v^t = \sigma\left(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}\right) \tag{4}$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \tag{5}$$

$$\mathbf{h}_v^{(t)} = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \widetilde{\mathbf{h}_v^{(t)}}. \tag{6}$$

From Li et al (ICLR 2016).

Gori, M., Monfardini, G. and Scarselli, F., 2005. A New Model for Learning in Graph Domains. In *Proc of IEEE International Joint Conference on Neural Networks 2005*.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M. and Monfardini, G., 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks 2009*.

Li, Y., Tarlow, D., Brockschmidt, M. and Zemel, R., 2015. Gated Graph Sequence Neural Networks. In *Proc of International Conference on Learning Representations 2016*.

Simple Reasoning Task

D is A
B is E
A has_fear F
G is F
E has_fear H
H has_fear A
C is H
eval B has_fear H?

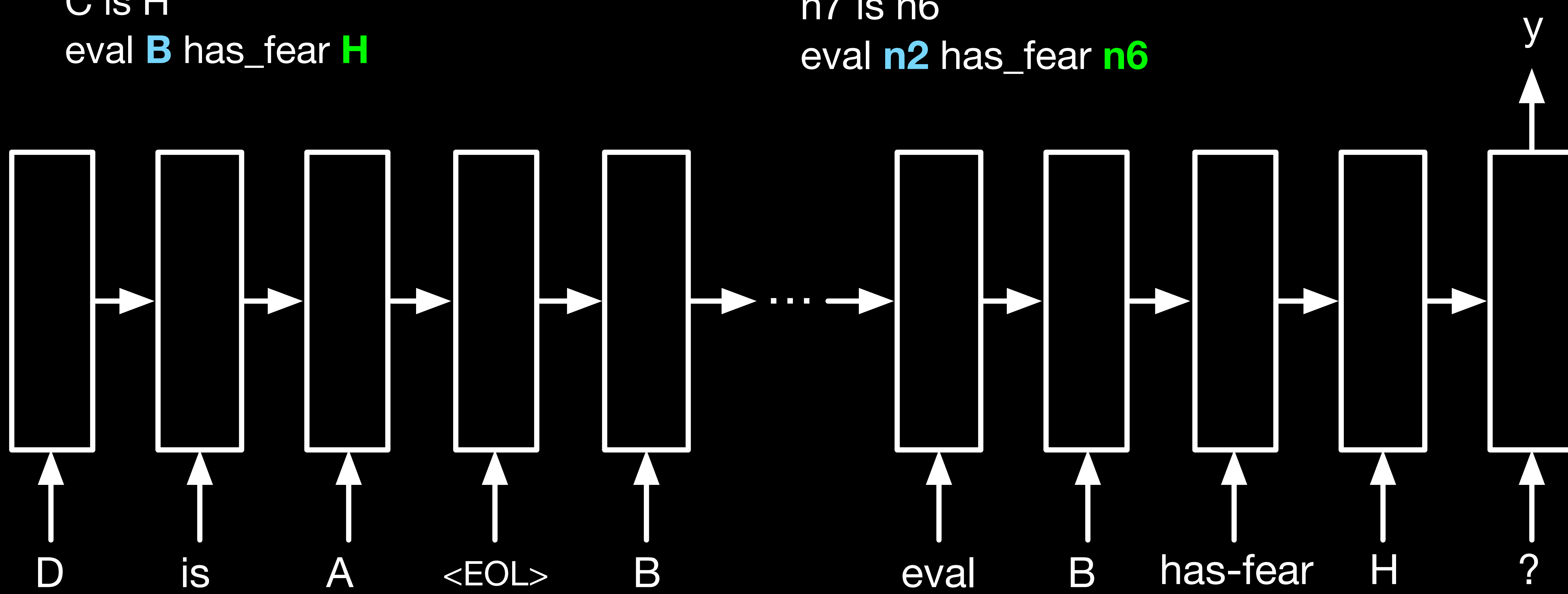
Weston, J., Bordes, A., Chopra, S., Rush, A.M., van Merriënboer, B., Joulin, A. and Mikolov, T., 2015. Towards AI-complete question answering: A set of prerequisite toy tasks. arXiv preprint arXiv:1502.05698.

Simple Reasoning Task - RNN representation

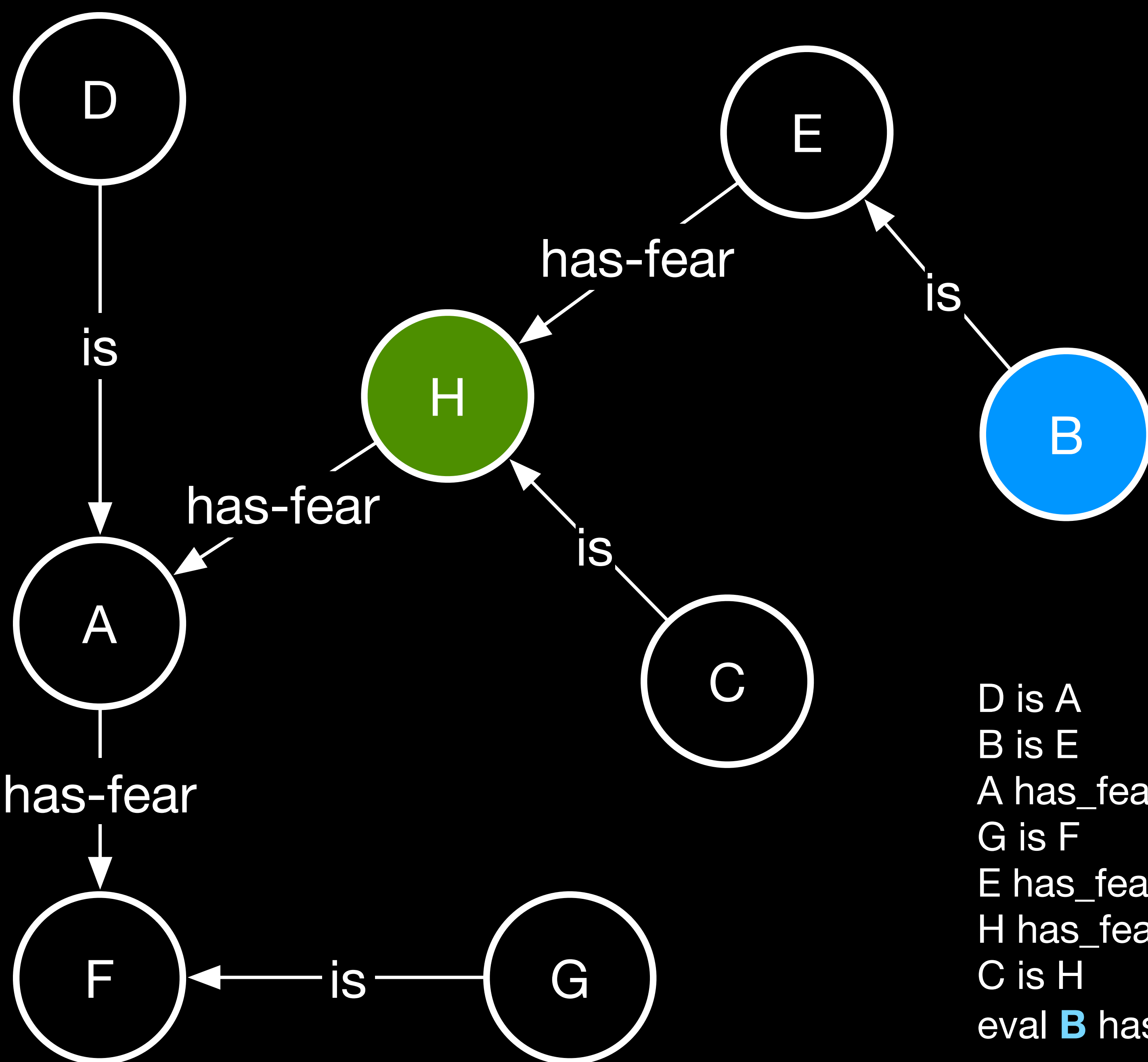
D is A
B is E
A has_fear F
G is F
E has_fear H
H has_fear A
C is H
eval **B** has_fear **H**

Normalize

n0 is n1
n2 is n3
n1 has_fear n4
n5 is n4
n3 has_fear n6
n6 has_fear n1
n7 is n6
eval **n2** has_fear **n6**



Simple Reasoning Task - GNN representation



D is A
B is E
A has_fear F
G is F
E has_fear H
H has_fear A
C is H
eval **B** has_fear **H**

Properties that enable strong generalization:

Value independence: Learning propagation algorithm, not mapping of name sequences to answers → generalizes to node names and graph structures never seen during training.

Modular: Resilient to adding "distraction subgraphs".

Main Limitations: Not always easy to convert real data (e.g., natural language) to graph format (but see Johnson (2016)) + memory use

Johnson, D. 2016. Learning Graphical State Transitions. *Submitted to ICLR 2017.*

Gated Graph Sequence Neural Networks - Experiments

Single Output tasks

Task	RNN	LSTM	GG-NN
bAbI Task 4	97.3±1.9 (250)	97.4±2.0 (250)	100.0±0.0 (50)
bAbI Task 15	48.6±1.9 (950)	50.3±1.3 (950)	100.0±0.0 (50)
bAbI Task 16	33.0±1.9 (950)	37.5±0.9 (950)	100.0±0.0 (50)
bAbI Task 18	88.9±0.9 (950)	88.9±0.8 (950)	100.0±0.0 (50)

Table 1: Accuracy in percentage of different models for different tasks. Number in parentheses is number of training examples required to reach shown accuracy.

Sequential Output tasks

Task	RNN	LSTM	GGS-NNs		
bAbI Task 19	24.7±2.7 (950)	28.2±1.3 (950)	71.1±14.7 (50)	92.5±5.9 (100)	99.0±1.1 (250)
Shortest Path	9.7±1.7 (950)	10.5±1.2 (950)	100.0± 0.0 (50)		
Eulerian Circuit	0.3±0.2 (950)	0.1±0.2 (950)	100.0± 0.0 (50)		

Table 3: Accuracy in percentage of different models for different tasks. The number in parentheses is number of training examples required to reach that level of accuracy.

Takeaway - when problem is well-represented as a simple graph, GNN formulation learns a more accurate model from less data.

How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures
   - Graph algorithms → Graph Neural Networks

b. Learn models represented as source code
   - Extend differentiable interpreters to learn neural network subroutines

How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures

- Graph algorithms → Graph Neural Networks

b. Learn models represented as source code

- **Extend differentiable interpreters to learn neural network subroutines**

Motivation

GNNs are built around a very restricted algorithmic template.

Can we also learn the algorithmic template?

→ Build on differentiable interpreters

Bunel, Desmaison, Kohli, Torr, Kumar. “Adaptive Neural Compilation.” NIPS 2016.
Riedel, Bošnjak, Rocktäschel. “Programming with a Differentiable Forth Interpreter.” 2016.
Gaunt et al. “TerpreT: A Probabilistic Programming Language for Program Induction. 2016.

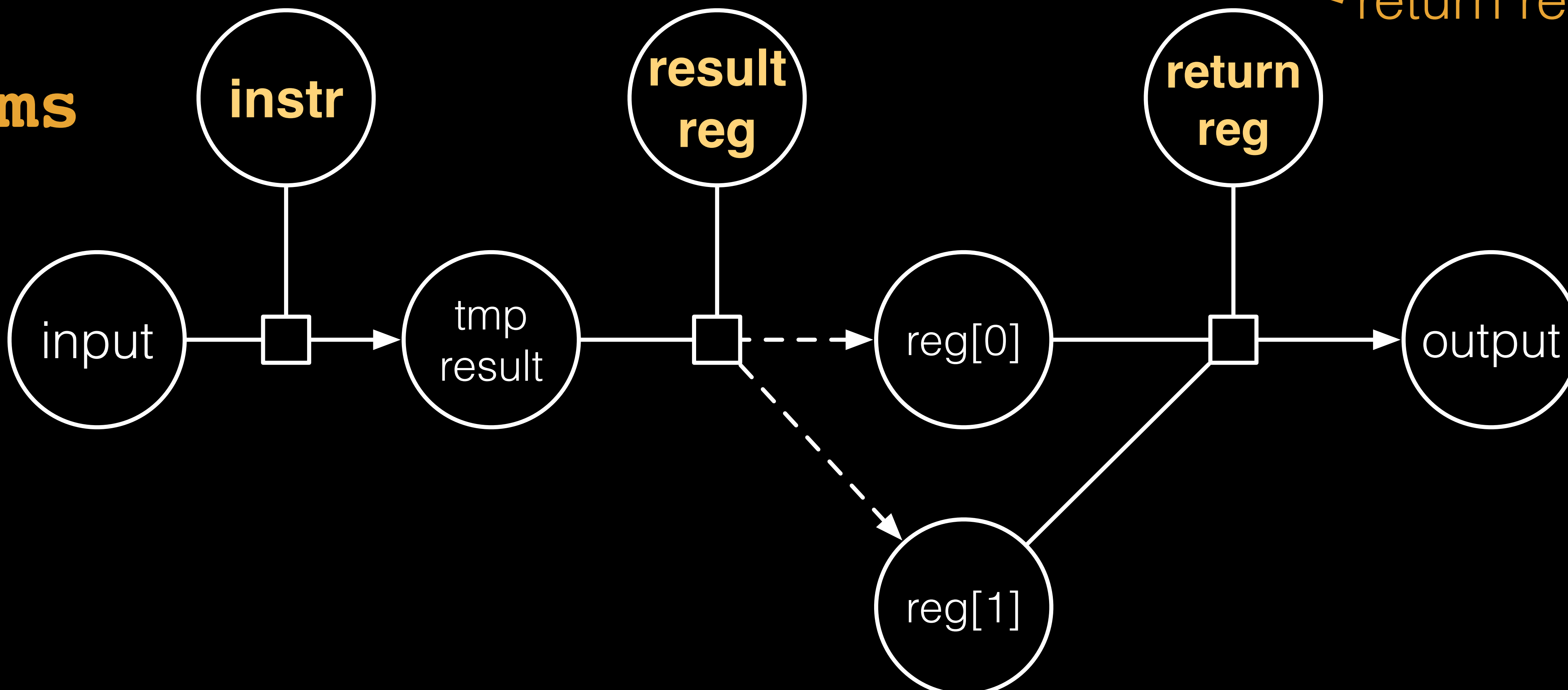
Differentiable Interpreter: Example

result reg　　　　　　　　instr

```
reg[0] = 0; reg[1] = 0
reg[??] = instruction[??](input)
return reg[??]
```

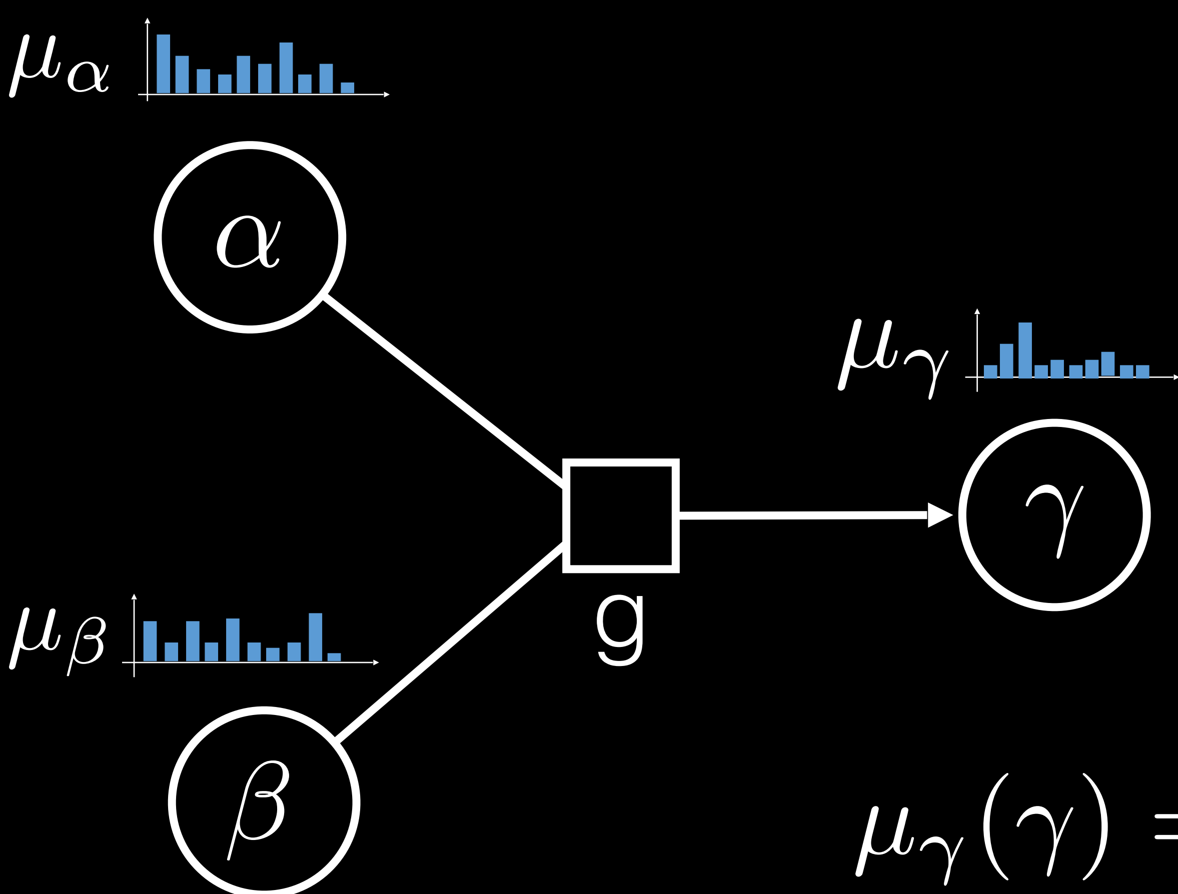
return reg

Params



1. Let **`Params`** be learnable categorical distributions
2. Lift all operations (squares) to be differentiable
3. Observe discrete inputs and outputs
4. Maximize `p(outputs|inputs; Params)`

Lifting Function Applications



$$\mu_\gamma(\gamma) = \sum_{\alpha,\beta} \mu_\alpha(\alpha)\mu_\beta(\beta)[\gamma = g(\alpha,\beta)]$$

Example:

α = register 1

β = register 2

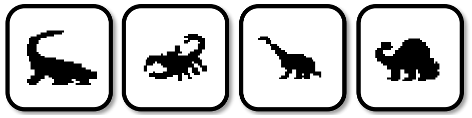
γ = result

g = add

Similar logic applies to `if` statements

See Gaunt et al. poster on “TerpreT” for lots more.

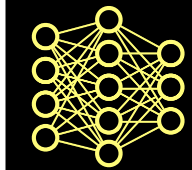
Adding Neural Function Calls to Differentiable Interpreter




```
T = 5; tape_length = 4; max_int = tape_length

@Runtime([max_int, 2], max_int)
def add(a, b): return (a + b) % max_int

@Runtime([tape_length], tape_length)
def inc(p): return (p + 1) % tape_length

@Learn([Tensor(28,28)],2,hid_sizes=[256,256])
def is_dinosaur(image): pass

tape = InputTensor(28,28)[tape_length]
instr = Param(2)[T]
count = Var(max_int)[T + 1]
pos = Var(tape_length)[T + 1]
tmp = Var(2)[T + 1]

pos[0].set_to(0)
count[0].set_to(0)

for t in range(T):
    if instr[t] == 0:   # MOVE
        pos[t + 1] = inc(pos[t])
        count[t + 1].set_to(count[t])
    elif instr[t] == 1: # READ
        pos[t + 1].set_to(pos[t])
        with pos[t] as p:
            tmp[t].set_to(is_dinosaur(tape[p]))
            count[t + 1].set_to(
             add(count[t], tmp[p]))

final_count = Output(max_int)
final_count.set_to(count[T - 1])
```

Task: learn to classify dinosaurs from counts supervision

Define some discrete functions



Define a neural function = instantiate a neural net

Program to infer: decide whether to MOVE or READ at each timestep t = 1…T

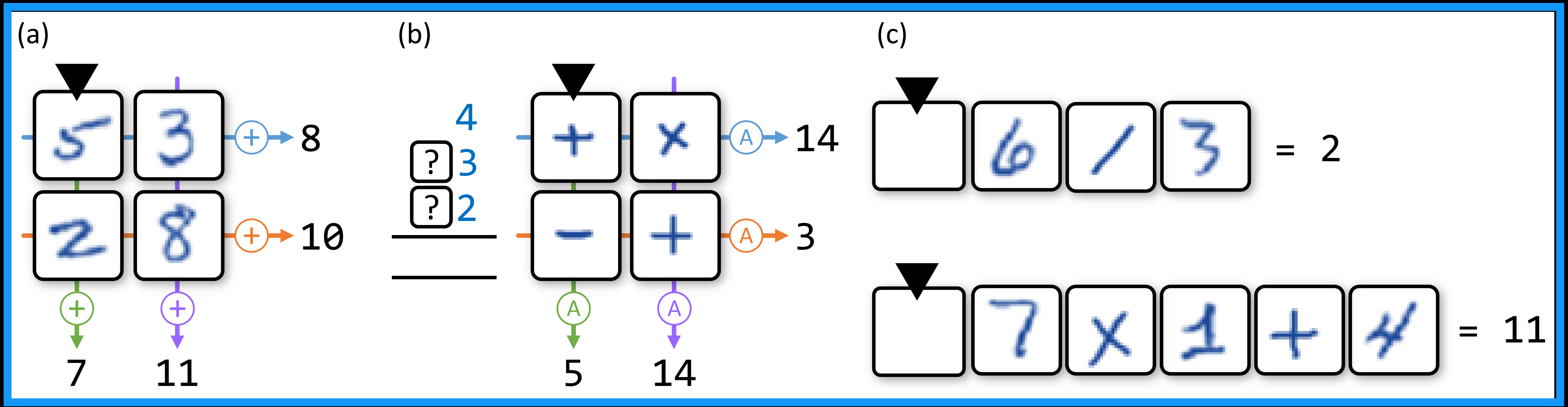
Can then call neural functions like discrete functions, taking tensor data as input

Observe total counts

Lifelong Perceptual Programming by Example
Alex Gaunt, Marc Brockschmidt, Nate Kushman, Daniel Tarlow.
arXiv:1611.02109

End-to-end differentiable, train by SGD

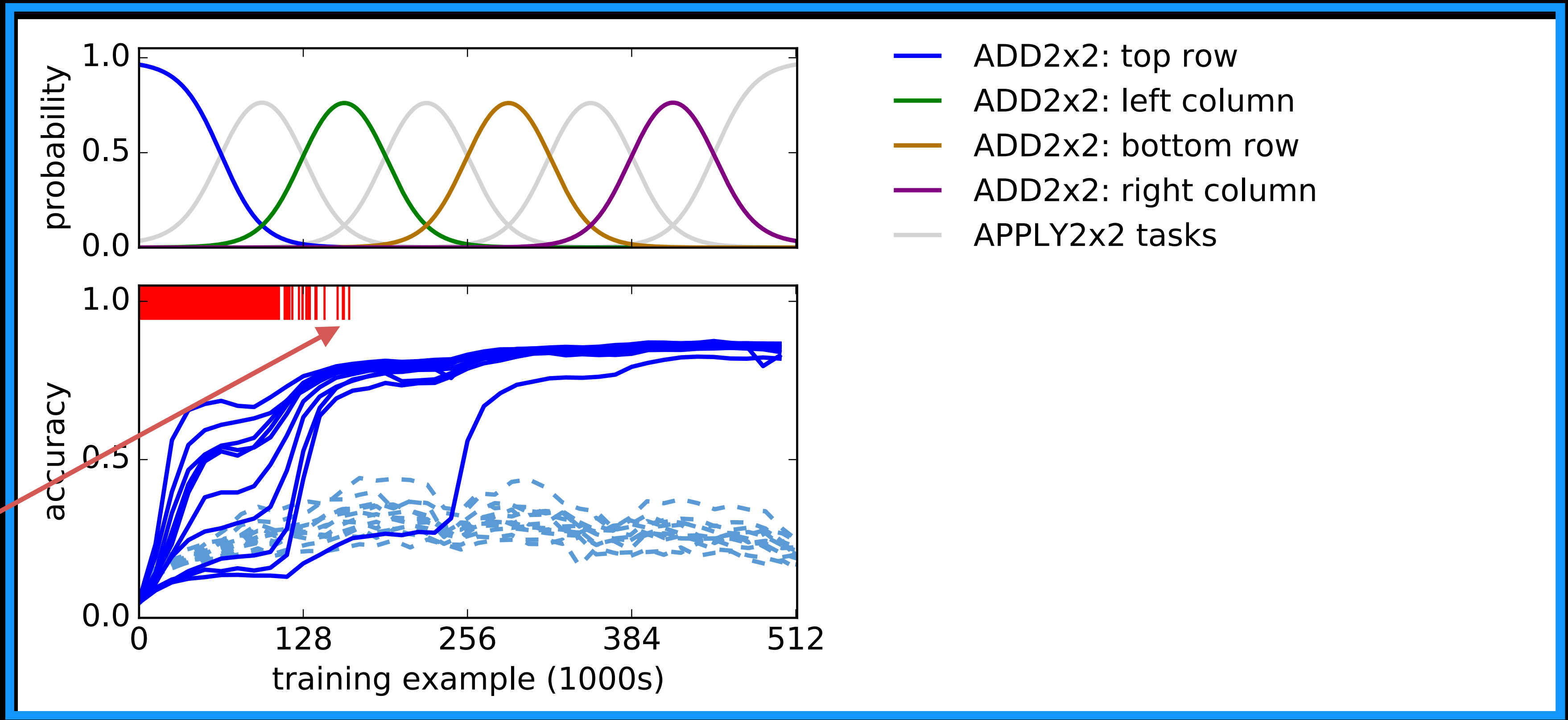
Learn programs for sequence of tasks; share neural functions



Resilient to catastrophic forgetting; demonstrates reverse transfer

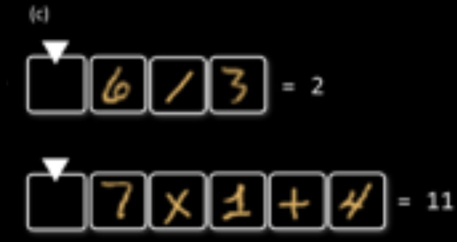
Distribution of tasks vs time

Last seen example of first task



First task performances (solids) and baseline multitask net (dashed)

Strong Generalization



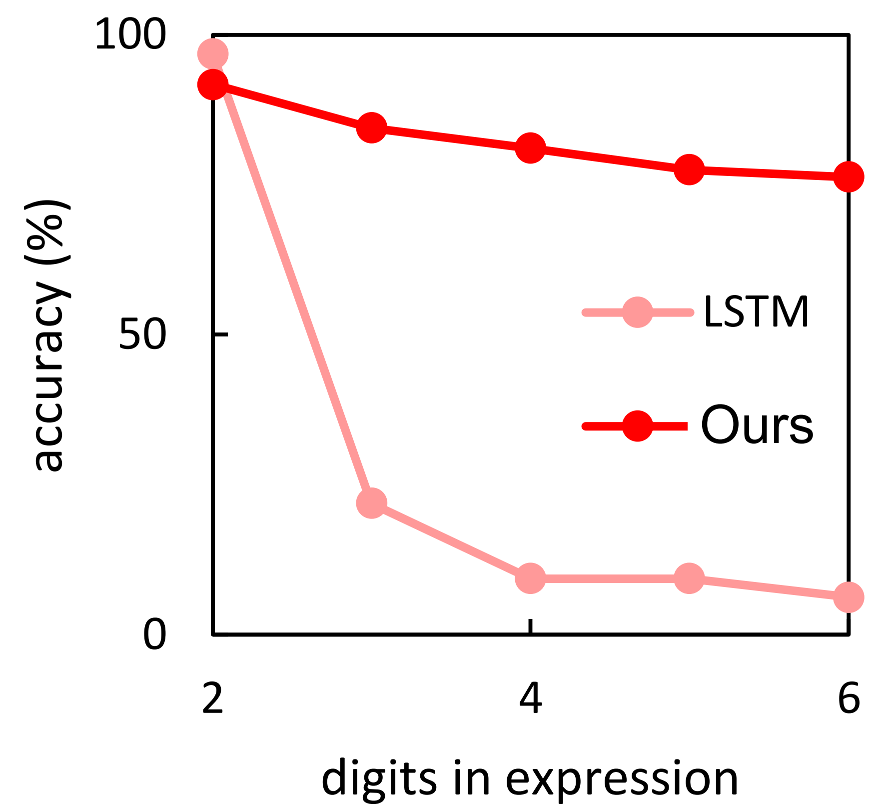


Figure 7: Generalisation behaviour on MATH expressions of varying length after training on 2 digit expressions.

Strengths:

Exhibits strong generalization- error is just based on error of MNIST classification

Learning is cumulative; can continue to update all parameters as task distribution shifts

Main Limitation:

Differentiable interpreters are susceptible to local optima. Training requires many random restarts (here ~50)

Future:

Incorporate learned priors over source code into model

Why does this mitigate catastrophic forgetting?

Hinton on Mixture of Experts:

"This may allow particular models to specialize in a subset of the training cases. **They do not learn on cases for which they are not picked. So they can ignore stuff they are not good at modeling.**"

Our speculation:

When a network starts to specialize, it provides enough signal to the source code component to know which network to use. Once the source code component picks a network, the others can ignore stuff they are not good at modeling. The source code component *focuses the supervision* for the neural nets.

Jacobs, R.A., Jordan, M.I., Nowlan, S.J. and Hinton, G.E., 1991. Adaptive mixtures of local experts. Neural computation, 3(1), pp.79-87.
Hinton, G.E. CSC2515 Lecture notes, http://www.cs.toronto.edu/~hinton/csc2515/notes/lec7.htm

This Talk

1. Strong generalization
    - By building neural network models inspired by *natural source code*[1]

2. Weak supervision
    - By differentiating through approximate marginalization algorithms

Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., 2012, June. On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 837-847). IEEE.

This Talk

1. Strong generalization
    - By building neural network models inspired by *natural source code*[1]

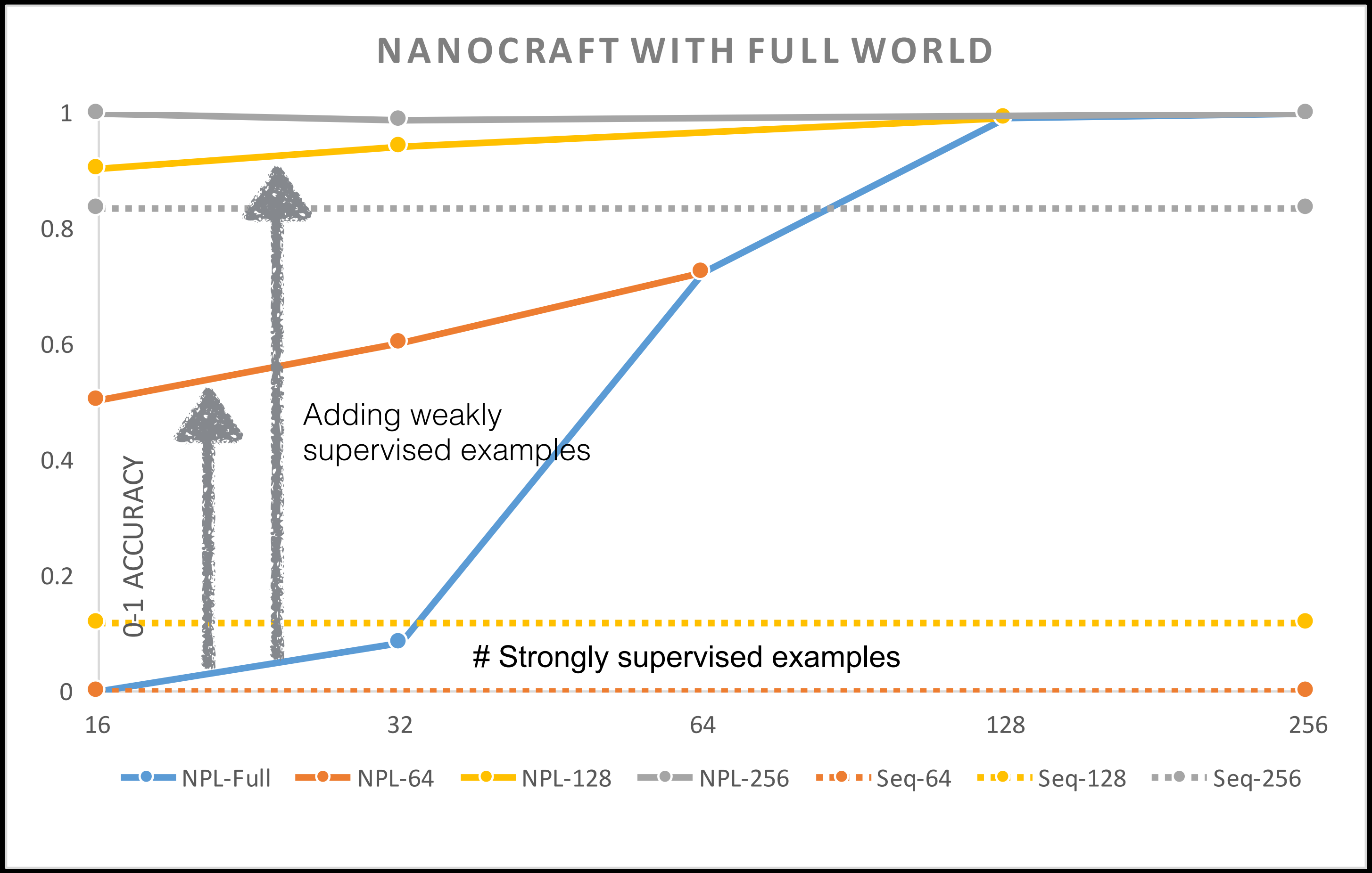
2. **Weak supervision**
    - **By differentiating through approximate marginalization algorithms**

Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., 2012, June. On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 837-847). IEEE.

Weak Supervision

Idea: train Neural Programmer-Interpreter with weaker supervision by building model that shares algorithm structure with dynamic program to compute marginal log likelihood of observations.

Builds on:
- Connectionist Temporal Classification (Graves et al, 2006)
- Stack RNN (Joulin & Mikolov, 2015)



Neural Program Lattices
Chengtao Li, Daniel Tarlow, Alex Gaunt, Marc Brockschmidt, Nate Kushman.
ICLR 2017 Submission

Conclusions

Graph Neural Networks

- It might be natural to encode your data as a graph.
- Node representations can be output of / input to other net to handle perceptual data.
- We can go a long ways by learning models like GNNs that generalize across graph structures.

Source Code Inductive Bias

- Inspires models that can strongly generalize and build up a library of components over time. Not restricted to symbolic data!
- Some surprising benefits like resilience to catastrophic forgetting.
- Need better program induction methods (see Balog et al. “DeepCoder” for using bottom-up cues to aid synthesis)
- Lots more to do in this space!

End