Neural Task Programming

Learning to Generalize Across Hierarchical Tasks

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, Silvio Savarese
Presenter: Atharva Sehgal

11/11/2021

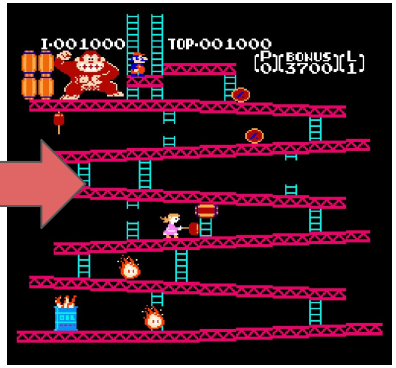
Overview

- Why?
  - Motivations
  - Main Goals
  - Problem Setting
  - Related Work
- How?
  - Approach
  - Evaluation
  - Results & Critique
- What next?
  - Limitations
  - Extended Readings

Motivations

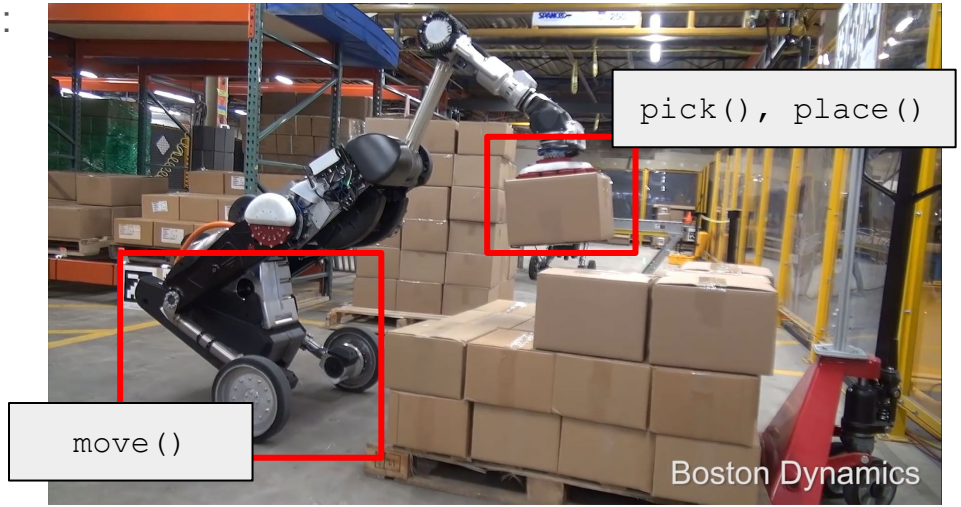
We've focused a lot on learning primitive functions:

- Policy for Grasping/Manipulation/Tool handling.
- Policy for Montezuma's revenge.



However, real world tasks requires learning progams:

- program := composition of primitive functions
- Warehouse robots:
  - retrieving, sorting, packing objects



Why Programs?

- Reusability
- Data hiding
- Interpretability

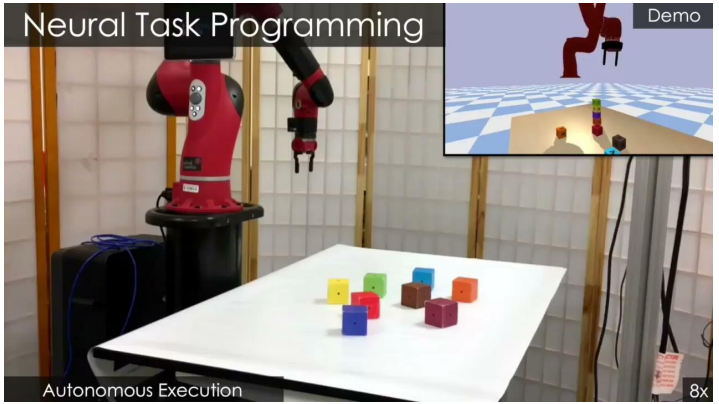
Main Goals

Goals of the paper:

- Learn a policy that:
  - allows us to perform multi-step hierarchical tasks
  - generalizes to new, unseen tasks in a data-efficient manner

Challenges

- Scaling up to complex multi-stage, long horizon tasks:
  - Traditional RL / Meta-learning based algorithms suffer.
- Generalizing across tasks:
  - Hierarchical RL based models suffer



Demo of final model on object stacking task

Problem Setting

Given distributions of observations ($\mathbb{O}$), actions ($\mathbb{A}$) tasks ($\mathbb{T}$), states ($\mathbb{S}$), and task specifications (Ψ), we want to find a meta-policy ($\tilde{\pi}$) that induces a policy (π) which can reach a task-completion state $s_T \in \mathbb{S}$. More specifically,

$$\pi(a \mid o; \psi(t)) : \mathbb{O} \to \mathbb{A} \qquad \mid o \in \mathbb{O}, a \in \mathbb{A}, t \in \mathbb{T}, \psi \in \Psi$$
$$\tilde{\pi} : \Psi \to (\mathbb{O} \to \mathbb{A})$$

Related Work

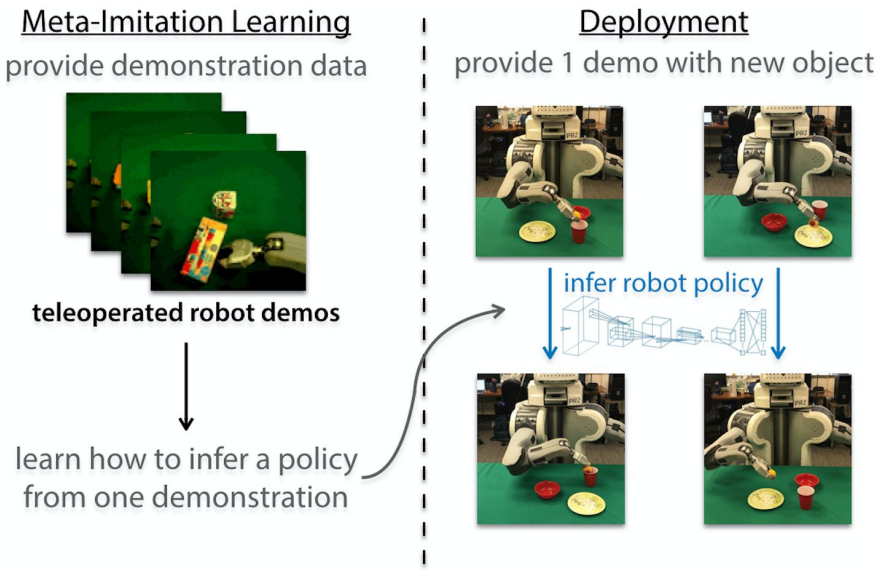
Learning from Demonstrations

- Use a policy to generalize from expert demonstrations.
- (+) No reward specification needed.
- (?) Policy generalizing to new objectives

Few Shot Generalization in LfD

Hierarchical Skill Composition

Neural Program Induction



Related Work

Learning from Demonstrations

Few Shot Generalization in LfD

- Use meta-learning to learn to induce models for learning from input specification
- (+) Generalization from single demostration
- (-) Require rich reward functions

Hierarchical Skill Composition

Neural Program Induction

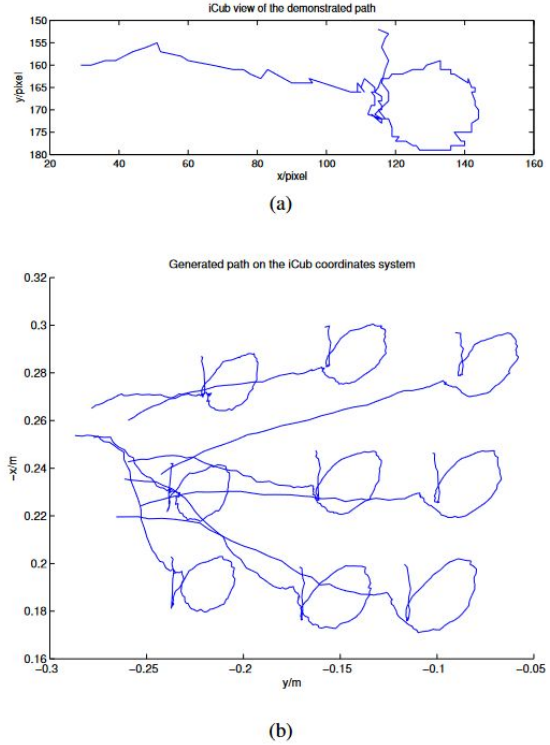


Fig. 4: **Paths imitated from one single demonstration.** (a) shows the demonstrated path seen from the left camera of the iCub. (b) shows the generated paths for marking different cells in the iCub's coordinates system.

Related Work

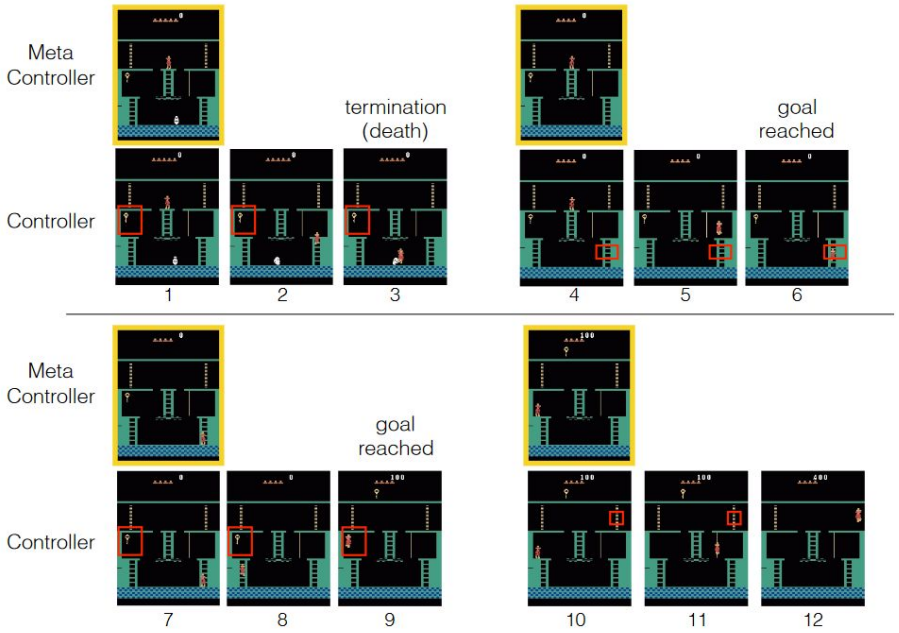
Learning from Demonstrations

Few Shot Generalization in LfD

Hierarchical Skill Composition

- Divide and conquer learning skills as learning sub-skills and learning to combine sub-skills
- (+) Great performance in several tasks
- (-) Cannot generalize across tasks

Neural Program Induction



Related Work

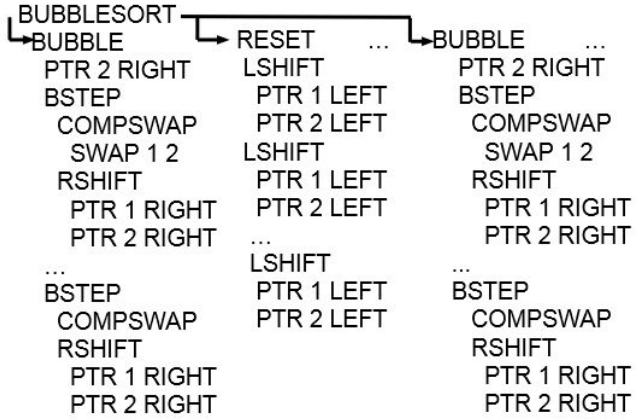
Learning from Demonstrations

Few Shot Generalization in LfD

Hierarchical Skill Composition

Neural Program Induction

- Use a neural network to induce a program from a given specification
- (+) Can learn complex, intricate programs
- (-) Retrain to adapt to a different domain.



(b) Excerpt from the trace of the learned bubblesort program.

Approach

Insight 1: Neural Program Induction

- Splice the full demonstration into smaller parts and recursively evaluate the smaller parts using subroutine calls.

Insight 2: Clever Inputs/Outputs to make NPI recursive

- **Inputs**: Current Observation, Current Program, and Current Task specification
  - The task specification can be symbolic (expert program trace) or neural (expert video embedding)
- **Outputs**: Output program, EoP probability, next program arguments
  - The next program arguments are API parameters if “primtive” program or the task specification pertaining to the next program otherwise.

Insight 3: Primitive API (`grip(obj), release(), moveto(obj)`)

- Need to abstract away the low-level controls otherwise will be hard to generalize to tasks.
- Restricts depth of program trace as well.
- Implemented using symbolic motion planners.

Approach

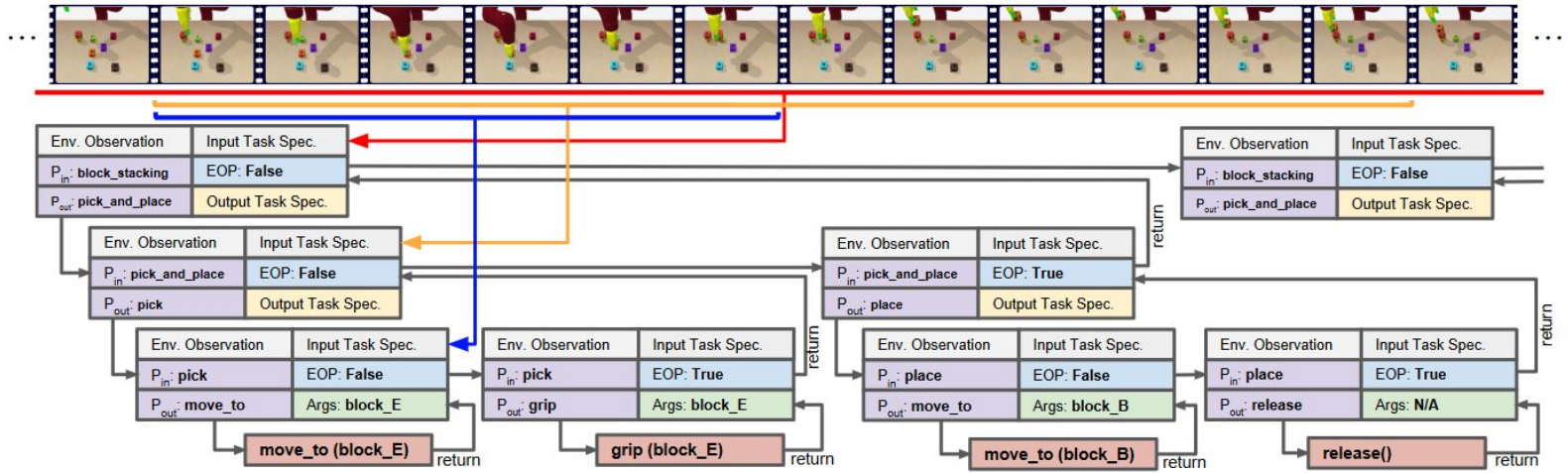


Fig. 3: Sample execution trace of NTP on a block stacking task. The task is to stack lettered blocks into a specified configuration (`block_D` on top of `block_E`, `block_B` on top of `block_D`, etc). Top-level program `block_stacking` takes in the entire demonstration as input (red window), and predicts the next sub-program to run is `pick_and_place`, and it should take the part of task specification marked by the orange window as the input specification. The bottom-level API call moves the robot and close / open the gripper. When End of Program (EOP) is True, the current program stops and return its caller program.

Evaluation

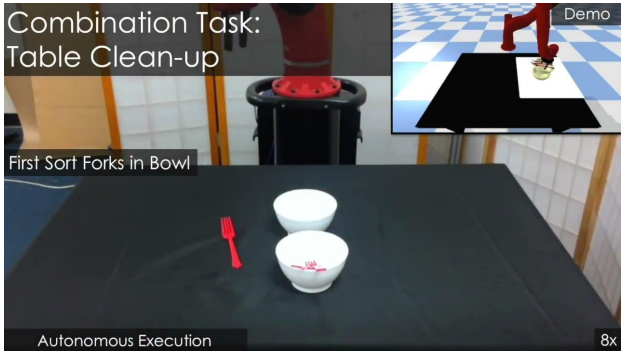
Main Questions (quoted verbatim) and Evaluation Paradigms:

1. Does NTP generalize to changes in length, topology, and semantics?
    - Vary task length by varying number of steps needed to achieve goal state.
    - Vary task topology by varying the permutations of steps needed to achieve goal state.
    - Vary task semantics by evaluating on unseen goal state configurations.
2. Can NTP use image-based input without access to ground truth input?
    - Replace state information with video as task-specification.
3. Would NTP also work in real-world complex tasks?
    - Deploy NTP on real robots in real environments.

Evaluation

Tasks:

1. Object Stacking
2. Object Sorting
3. Table Cleanup



Evaluation

Baselines:

1. **Flat**: Non-hierarchical model that directly predicts primitive API from input demonstration.
2. **Flat (GRU):** Same as above but a GRU cell allows storing past state information.
3. **NTP (no scope):** NTP without data hiding. The entire demonstration is fed to subroutines.
4. **NTP (GRU):** Complete NTP model with GRU cell replacing the reactive core.

Metrics

The mean success rate per 100 evaluations is calculated for all experiments.

Adversarial Dynamics:

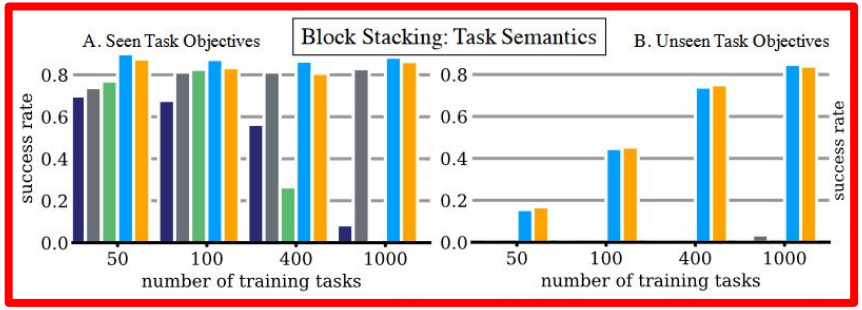
Showcase that NTP is a closed loop policy and can recover from failures.

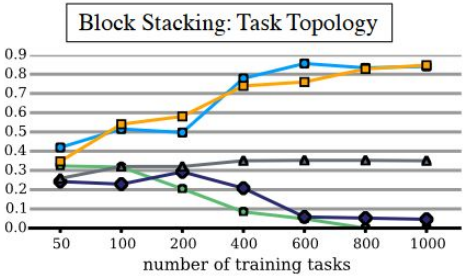
Evaluation

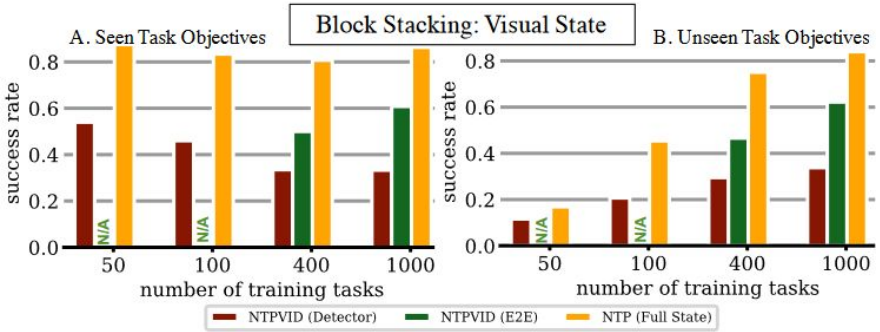
=> NTP and variants steadily improves in performance with more training tasks.

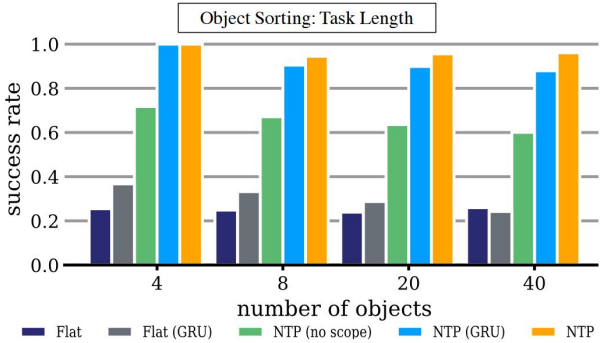
=> NTP is robust to changes in task semantics.

=> Data scoping is important for generalization





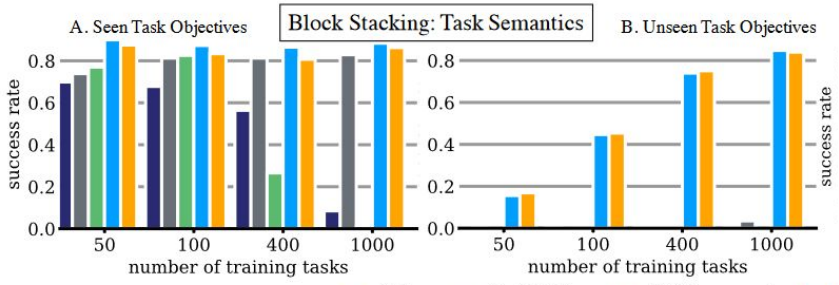


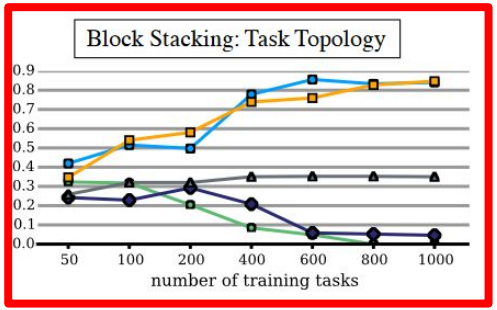


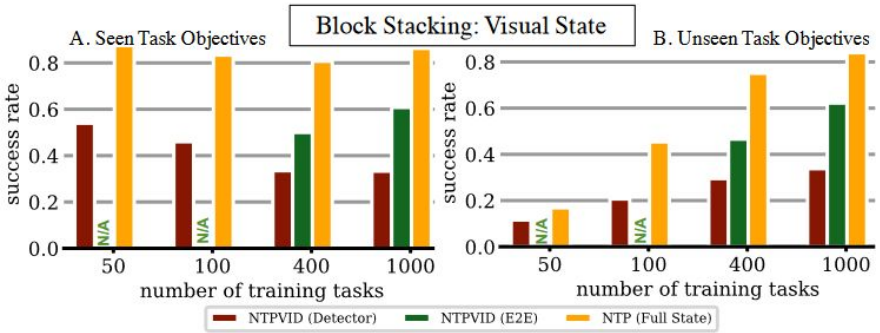
Evaluation

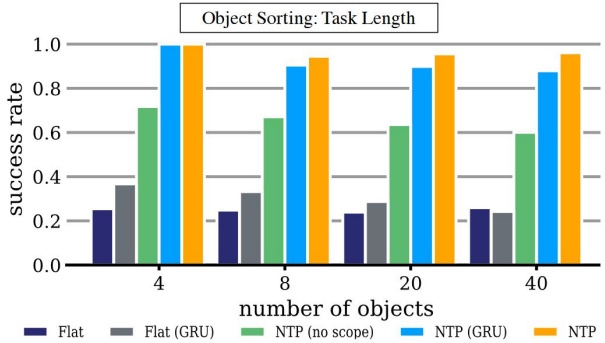
=> Flat baselines are not good at handling permutations

=> Hierarchical scoping of NTP helps it generalize better to task topology





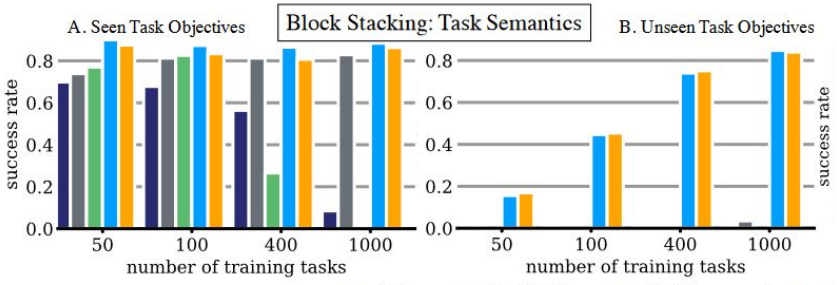


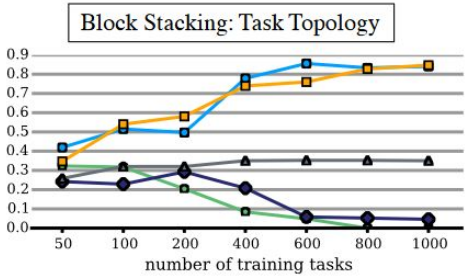


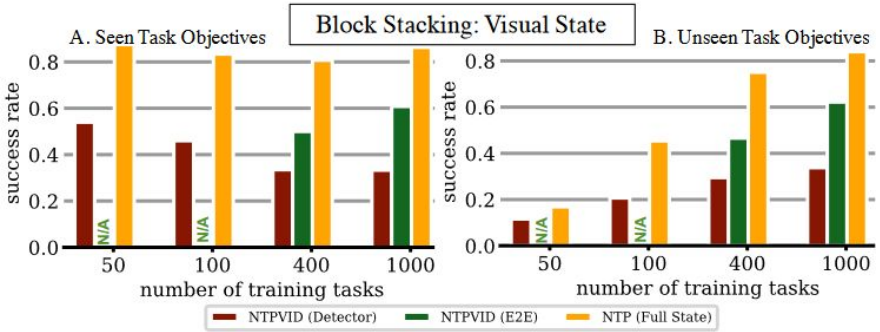
Evaluation

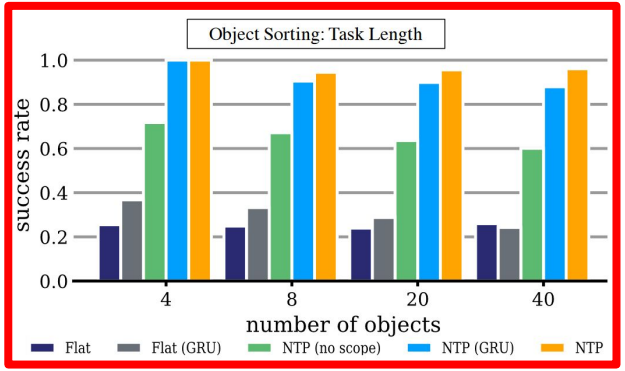
=> NTP is robust to changes in task length and can even handle upto 40 objects.

Most significant result of the paper.



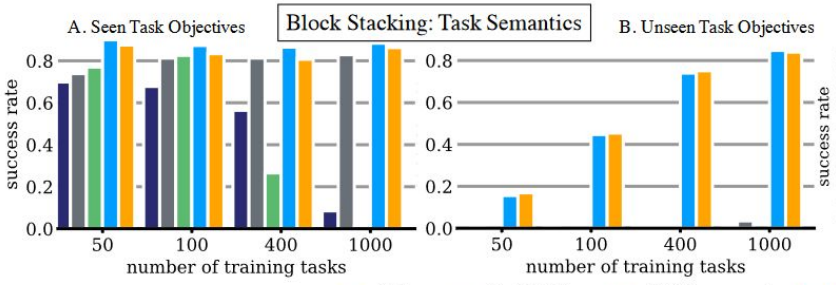


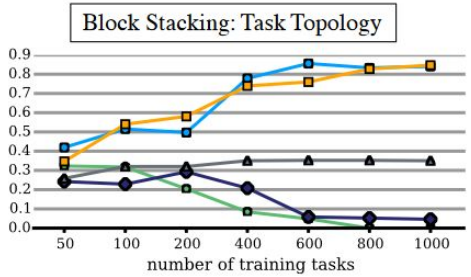


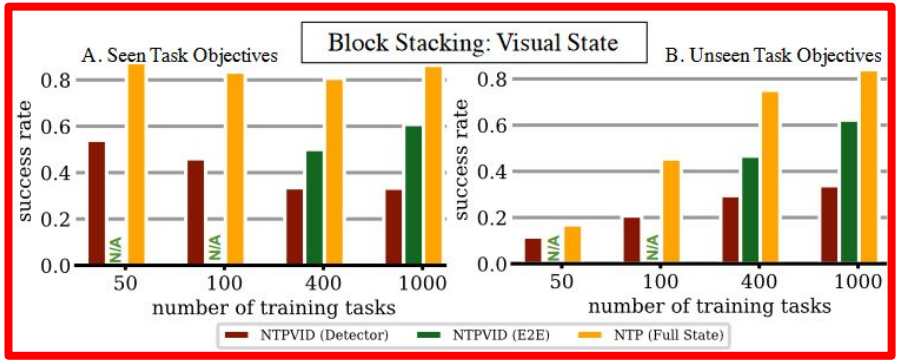


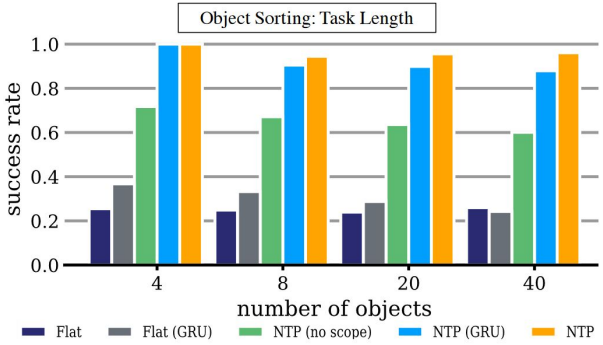
Evaluation

NTPVID (detector): input = object positions
NTPVID (E2E): input = image embedding
NTP(full state): input = program state
=> We can reliably learn from videos.
=> Task specific encoders (detector) can imbue vision model errors which can cascade.









Evaluation

=> NTP can generalize to real world situations.

Tasks	# Trials	Success	NTP Fail	Manip. Fail
Blk. Stk.	20	0.9	0.05	0.05
Sorting	10	0.8	0	0.20

TABLE I: **Real Robot Evaluation**: Results of 20 unseen Block Stacking evaluations and 10 unseen sorting evaluations on Sawyer robot for the NTP model trained on simulator. NTP Fail denotes an algorithmic mistake, while Manip. Fail denotes a mistake in physical interaction (e.g. grasping failures and collisions).

Model	No failure	With failures
NTP	0.863	0.663
NTP (GRU)	0.884	0.422

TABLE II: **Adversarial Dynamics**: Evaluation results of the Block Stacking Task in a simulated adversarial environment. We find that NTP with GRU performs markedly worse with intermittent failures.

Evaluation

=> NTP can recover from failures.

=> NTP's reactive core is instrumental in enabling it to be a closed loop policy.

Tasks	# Trials	Success	NTP Fail	Manip. Fail
Blk. Stk.	20	0.9	0.05	0.05
Sorting	10	0.8	0	0.20

TABLE I: **Real Robot Evaluation**: Results of 20 unseen Block Stacking evaluations and 10 unseen sorting evaluations on Sawyer robot for the NTP model trained on simulator. NTP Fail denotes an algorithmic mistake, while Manip. Fail denotes a mistake in physical interaction (e.g. grasping failures and collisions).

Model	No failure	With failures
NTP	0.863	0.663
NTP (GRU)	0.884	0.422

TABLE II: **Adversarial Dynamics**: Evaluation results of the Block Stacking Task in a simulated adversarial environment. We find that NTP with GRU performs markedly worse with intermittent failures.

Results & Critique

From the experiments, the authors conclude that:

1. NTP learns modular and reusable neural programs for hierarchical tasks.
2. NTP can successfully operate in prolonged and complex interactions with the environment.

Strengths:

- NTP can learn subroutines for long-horizon multi-task learning. This is pretty cool!

Weaknesses:

- Object sorting, object stacking, and table cleanup all use the same primitives and some common subroutines. However, It is unclear if the model can generalize across tasks. Would the model generalize to object stacking if its trained on object sorting (and given an object stacking video at test time)?

Limitation

1. Restrictive API
   - NTP needs to use a high level API to restrict the program depth and to make training easier. However, because of this, NTP cannot learn a rich library of methods to perform a certain action.
   - For instance, NTP cannot perform a side-grasp as no rotation primitive is available.
2. Introducing new “subroutines” requires retraining
   - Limitation of program induction.
   - Adding new constructs to NTP requires changing its entire architecture.
3. NTP requires rich structural supervision:
   - It needs 1000 full program traces to generalize to unseen examples for a task.
   - Follow up work removed the need for the supervision to be structured.

Future Work

1. Using a lower-level API to allow torque and velocity based controllers.
2. Improving the perception to capture richer relationships between objects
3. Extending framework to more complex tasks
4. Using symbolic methods to generate programs instead of program induction.

Extended Readings

- Neural Module Networks: https://arxiv.org/abs/1511.02799v4
- One-Shot Imitation from Watching Videos: https://bair.berkeley.edu/blog/2018/06/28/daml/
- Incremental Task and Motion Planning: http://www.roboticsproceedings.org/rss12/p02.pdf
- Hierarchical Deep Reinforcement Learning: https://arxiv.org/abs/1604.06057
- DreamCoder: https://arxiv.org/abs/2006.08381
- RobustFill: https://proceedings.mlr.press/v70/devlin17a/devlin17a.pdf
- Neural Guided Synthesis Tutorial:
  https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture22.htm

Summary

- We want to learn policies for long-horizon multi-stage tasks in a few-shot learning paradigm.
- This is a common setting in real-life robot use cases (warehouses)
- Prior work either couldn’t handle task complexity or required engineered reward signals.
- Current work alleviated this by doing recursive neural program induction. This allowed them to reuse modules and delegate tasks to specialized networks.
- Their method allowed them to learn a closed loop policy that could operate for long time-steps and solve complex hierarchical tasks with only a single video as test-time supervision.