Neural Task Programming: Learning to Generalize Across Hierarchical Tasks

Danfei Xu*[1], Suraj Nair*[2], Yuke Zhu[1], Julian Gao[1], Animesh Garg[1], Li Fei-Fei[1], Silvio Savarese[1]

***Abstract*— In this work, we propose a novel robot learning framework called Neural Task Programming (NTP), which bridges the idea of few-shot learning from demonstration and neural program induction. NTP takes as input a task specification (e.g., video demonstration of a task) and recursively decomposes it into finer sub-task specifications. These specifications are fed to a hierarchical neural program, where bottom-level programs are callable subroutines that interact with the environment. We validate our method in three robot manipulation tasks. NTP achieves strong generalization across sequential tasks that exhibit hierarchal and compositional structures. The experimental results show that NTP learns to generalize well towards unseen tasks with increasing lengths, variable topologies, and changing objectives. `stanfordvl.github.io/ntp/`**

I. Introduction

Autonomy in complex manipulation tasks, such as object sorting, assembly, and de-cluttering, requires sequential decision making with prolonged interactions between the robot and the environment. Planning in a complex task and, vitally, adapting to new task objectives and initial conditions is a long-standing challenge in robotics [6, 13].

Consider an object sorting task in a warehouse setting – it requires sorting, retrieval from storage, and packing for shipment. Each task is a sequence of a hierarchy of primitives – such as `pick_up`, `move_to`, and `drop_into` – that can be composed into manipulation sub-tasks such as grasping and placing. This problem has a expansive space of variations – different objects-bin maps in sorting, permutations of sub-tasks, varying length order lists – resulting in a large space of tasks. As a concrete example, Figure 1(C) shows a simplified setup of the object sorting task. The task is to transport objects of four categories to four shipping containers. There is total of 256 possible mapping between object categories and containers, and the variable number of object instances further increases the complexity. In this paper, we attempt to address two challenges in complex task planning domains, namely (a) *learning policies that generalize to new task objectives*, and (b) *hierarchical composition of primitives for long-term environment interactions*.

We propose Neural Task Programming (NTP), a unified, task-agnostic learning algorithm that can be applied to a variety of tasks with latent hierarchical structure. The key underlying idea is to learn reusable representations shared across tasks and domains. NTP interprets a *task specification* (Figure 1 left) and instantiates a hierarchical policy as a neural program (Figure 1 middle), where the bottom-level programs are primitive actions that are executable in the environment. A task specification is defined as a time-series that describes

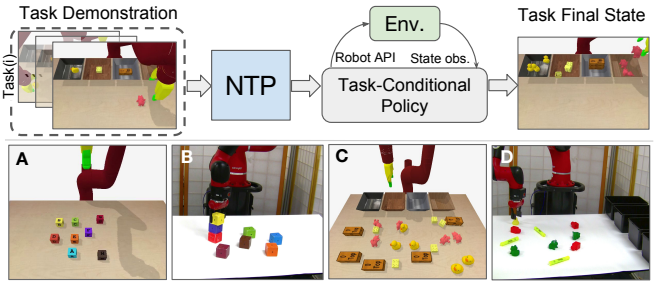

Fig. 1: (top) At test time, NTP instantiates a task-conditional policy (a neural program) that performs the specified task by interpreting a demonstration of a task. The policy interacts with the environment through robot APIs. (bottom) We evaluate NTP on Block Stacking (A,B), Object Sorting (C, D) and Table Clean-up (Figure 8) tasks in both simulated and real environment.

the procedure and the final objective of a task. It can either be a task demonstration recorded as a state trajectory or 1st 3rd person video, or even a list of language instructions. In this work, we use *task demonstration* as the task specification. We experiment with two forms of task demonstration: location trajectories of objects that are involved in a task, and a third-person video demonstration of a task. NTP decodes the objective of a task from the input specification and factorizes it into sub-tasks, interacting with the environment with closed-loop feedback until the goal is achieved (Figure 1 right). Each program call takes as input the environment observation and a task specification, producing the next sub-program and a corresponding sub-task specification. The lowest level of the hierarchy is symbolic actions captured through a Robot API. This hierarchical decomposition encourages information hiding and modularization, as lower-level modules only access their corresponding sub-task specifications that pertain to their functionality. It prevents the model from learning spurious dependencies on training data, resulting in better reusability. Essentially, NTP addresses the key challenges in task generalization: meta-learning for cross-task transfer and hierarchical model to scale to more complex tasks. Hence, NTP builds on the strengths of neural programming and hierarchical RL while compensating for their shortcomings.

We demonstrate that NTP generalizes to three kinds of variations in task structure: 1) *Task Length*: varying number of steps due to the increasing problem size (e.g., having more objects to transport); 2) *Task Topology*: the flexible permutations and combinations of sub-tasks to reach the same end goal (e.g., manipulating objects in different orders); and 3) *Task Semantics*: the varying task definitions and success conditions (e.g., placing objects into a different container).

Summary of Contributions:

1) Our primary contribution is a novel modeling framework: NTP that enables meta-learning on hierarchical tasks.

* These authors contributed equally to the paper
[1] Stanford Vision & Learning Lab, [2] CS, Caltech.

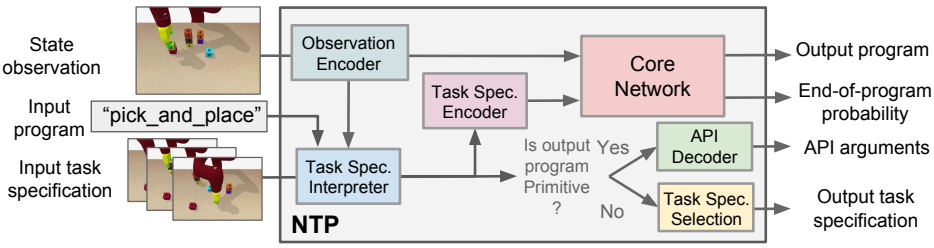

Fig. 2: **Neural Task Programming (NTP)**: Given an input program, a task specification, and the current environment observation, a NTP model predicts the sub-level program to run, the subsequence of the task specification that the sub-level program should take as input, and if the current program should stop.

2) We evaluate NTP in modeling single-arm manipulation tasks: Block Stacking, Object Sorting, and Table Clean-up each in both simulated and real-robot experiments.
3) We show that NTP enables knowledge transfer and one-shot demonstration based generalization to novel tasks with increasing lengths, varying topology, and changing semantics without restriction on initial configurations.
4) We also demonstrate that NTP can be trained with visual input (images and video) end-to-end.

II. Background & Related work

Skill Learning: The first challenge is learning policies that adapt to new task objectives. For learning a single task policy, traditional methods often segment a complex task into hand-engineered state machine composed of motion primitives [6, 13, 24, 31]. Although, the model-based approaches are well-founded in principle, they require meticulous model specification and task-specific treatment leading to challenge in scaling. Contrarily, learning based methods such as reinforcement learning (RL) have yielded promising results using end-to-end policy learning that obviates the need for manually designed state representations through data-driven task-salient features [23, 37]. Yet these methods fall short because they need task-specific reward functions [19, 25].

Learning from Demonstrations: LfD fills these gaps by avoiding the need to define state machines or reward functions. The objective in LfD is to learn policies that generalize beyond the provided examples and are robust to perturbations [3, 20] A common treatment to LfD is to model data as samples from an expert policy for a fixed task, and use behaviour cloning [18, 29] or reward function approximation [26] to output an expert-like policy for that task. However, learning policies that generalize to new objectives with LfD remains largely an unexplored problem.

Few-Shot Generalization in LfD: Our work is an instantiation of the decades-old idea of meta-learning with few examples [12, 34]. It has seen a recent revival in deep learning in part because it can address the problems above [35].

Our setting resembles programming by demonstration (PbD) in robotics [5], particularly one-shot imitation [11, 36]. Our method *learns to learn* from an input task specification during training. At test time, it generates a policy conditioned on a *single* demonstration provided as a time-series showing the task execution. While similar in these aspects, existing works in both skill learning and LfD are inept at tasks with sparse reward functions and complex hierarchical structures such as Montezuma's Revenge [22].

Hierarchical Skill Composition: The second challenge we consider is the hierarchical composition of primitives to enable long-term robot-environment interaction. The idea of using hierarchical models for complex tasks has been widely explored in both reinforcement learning and robotics [20].

A common treatment to manage task structure complexity is to impose hierarchy to the learned policy. The *options* framework composes primitive actions into multi-step actions, which facilitates policy learning at higher-level semantic and/or temporal abstraction [14, 32]. Notable examples include structured reinforcement learning methods, especially hierarchical variants of RL that handle decomposition through multi-stage policies operating over options [4, 22, 27, 33]. However, the naive use of a hierarchical RL model with "sub-policies" or options optimized for a specific task, doesn't guarantee modularity or reusability across task objectives.

The core idea of NTP resonates with recent works on dynamic neural networks, which aim to learn and reuse primitive network modules. These methods have been successfully applied to several domains such as robot control [1] and visual question answering [2]. However, they have exhibited limited generalization ability across tasks. In contrast, we approach the problem of hierarchical task learning via neural programming to attain modularization and reusability [28]. As a result, our model achieves significantly better generalization results than non-hierarchical models such as [11].

FSMs and Neural Program Induction: An exciting and non-intuitive insight of this paper is that the well-studied Finite State Machine (FSM) model lends itself to learning reusable hierarchical policies thereby addressing the problem of composability without the need for hand-crafting state transitions. There have been a few studies learning FSMs from data [15, 21]. In line with the idea, recent works in neural programming using deep models have enabled symbolic reasoning systems to be trained end-to-end, which have shown potential to handle multi-modal and raw input/out data [9, 28] and achieve symbolic generalization [8].

NTP belongs to a family of neural program induction methods, where the goal is to learn a latent program representation that generates program outputs [9, 16, 28]. While these models have been shown to generalize on task length, they are tested on basic computational tasks only with limited generalization to task semantics and topology. Similar to NTP, Neural Programmer-Interpreter (NPI) [28] has proposed to use a task-agnostic recurrent neural network to represent and execute programs. In contrast to previous work on neural program induction, NPI-based models are trained with richer supervision from the full program execution traces and can learn semantically meaningful programs with high data efficiency. However, program induction, including NPI, is not capable of generalizing to novel programs without training.

NTP is a meta-learning algorithm that learns to instantiate

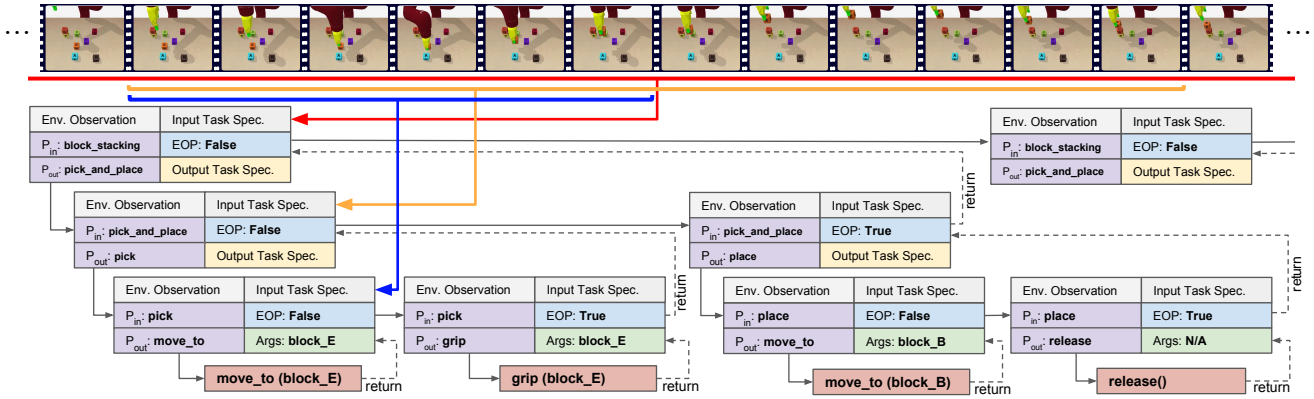

Fig. 3: Sample execution trace of NTP on a block stacking task. The task is to stack lettered blocks into a specified configuration (`block_D` on top of `block_E`, `block_B` on top of `block_D`, etc). Top-level program `block_stacking` takes in the entire demonstration as input (red window), and predicts 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.

neural programs given demonstrations of tasks, thereby generalizing to unseen tasks/programs. Intuitively, NTP decomposes the overall objective (e.g., object sorting) into simpler objectives (e.g., pick and place) recursively. For each of such sub-tasks, NTP delegates a neural program to perform the task. The neural programs, together with the task decomposition mechanism, are trained end-to-end.

While previous work has largely focused on executing a pre-defined task one at a time NTP not only exhibits one-shot generalization to tasks with longer lengths as NPI, but also generalizes to sub-task permutations (topology) and success conditions (semantics).

III. Problem Formulation

We consider the problem of an agent performing actions to interact with an environment to accomplish tasks. Let $\mathbb{T}$ be the set of all tasks, $\mathbb{S}$ be the environment state space, and $\mathbb{A}$ be the action space. For each task $t \in \mathbb{T}$, the Boolean function $g : \mathbb{S} \times \mathbb{T} \to \{0,1\}$ defines the success condition of the task. Given any state $s \in \mathbb{S}$, $g(s,t) = 1$ if the task t is completed in the state s, and $g(s,t) = 0$ otherwise. The task space $\mathbb{T}$ can be infinite. We thus need a versatile way to describe the task semantics. We describe each task using a task specification $\psi(t) \in \Psi$, where Ψ is the set of all possible task specifications. Formally, we consider a task specification as a sequence of random variables $\psi(t) = \{x_1, x_2, \ldots, x_N\}$.

NTP takes a *task specification* $\psi(t)$ as input in order to instantiate a policy. $\psi(t)$ is defined as a time series that describes the procedure and the final objective of the task. In experiments, we consider two forms of task specifications: trajectories of object locations and raw video sequences. In many real-world tasks, the agent has no access to the underlying environment states. It only receives a sample of environment observation $o \in \mathbb{O}$ that corresponds to the state s, where $\mathbb{O}$ is the observation space. Our goal is to learn a "meta-policy" that instantiates an feedback policy from a specification of a task, $\tilde{\pi} : \Psi \to (\mathbb{O} \to \mathbb{A})$. At test time, a specification of a new task $\psi(t)$ is input to NTP. The meta-policy then generates a policy $\pi(a|o; \psi(t)) : \mathbb{O} \to \mathbb{A}$, to reach task-completion state s_T where $g(s_T, t) = 1$.

Why use Neural Programming for LfD? Previous work has mostly used a monolithic network architecture to represent a goal-driven policy [10, 11, 30, 37]. These methods cannot exploit the compositional task structures to facilitate modularization and reusability. Instead, we represent our policy $\tilde{\pi}$ as a neural program that takes a task specification as its input argument. As illustrated in Figure 2, NTP uses a task-agnostic core network to decide which sub-program to run next and adaptively feeds a subset of the task specification to the next program. Intuitively, NTP recursively decomposes a task specification and solves a hierarchical task by divide-and-conquer. Figure 3 highlights this feature with a sample execution of a task. Our method extends upon a special type of neural programming architecture named Neural Programmer-Interpreter (NPI) [8, 28]. NPI generalizes well to input size but cannot generalize to unseen task objectives. NTP combines the idea of meta-learning and NPI. The ability to interpret task specifications and instantiate policies accordingly makes NTP generalize across tasks.

A. Neural Programmer-Interpreter (NPI)

Before introducing our NTP model, it is useful to briefly overview the NPI paradigm [28]. NPI is a type of neural program induction algorithm, in which a network is trained to imitate the behavior of a computer program, i.e., the network learns to invoke programs recursively given certain context or stop the current program and return to upper-level programs. The core of NPI is a long-short memory (LSTM) [17] network. At the i-th time step, it selects the next program to run conditioned on the current observation o_i and the previous LSTM hidden units h_{i-1}. A domain-specific encoder is used to encode the observation o_i into a state representation s_i. The NPI controller takes as input the state s_i, the program embedding p_i retrieved from a learnable key-value memory structure $[M^{key}; M^{prog}]$, and the current arguments a_i. It generates a program key, which is used to invoke a sub-program p_{i+1} using content-based addressing, the arguments to the next program a_{i+1}, and the end-of-program probability r_i. The NPI model maintains a program call stack. Each time a sub-program is called, the caller's

LSTM hidden units embedding and its program embedding is pushed to the stack. Formally, the NPI core has three learnable components, a domain-specific encoder f_{enc}, an LSTM f_{lstm}, and an output decoder f_{dec}. The full update being: $s_i = f_{enc}(o_i, a_i) \quad h_i = f_{lstm}(s_i, p_i, h_{i-1}) \quad r_i, p_{i+1}, a_{i+1} = f_{dec}(h_i)$. When executing a program with the NPI controller, it performs one of the following three things: 1) when the end-of-program probability exceeds a threshold α (set to 0.5), this program is popped up from the stack and control is returned to the called; 2) when the program is not primitive, a sub-program with its arguments is called; and 3) when the program is primitive, a low-level basic action is performed in the environments. The LSTM core is shared across all tasks.

IV. Neural Task Programming

Overview. NTP has three key components: Task Specification Interpreter f_{TSI}, Task Specification Encoder f_{TSE}, and a core network f_{CN} (Figure 2. The Task Specification Encoder transforms a task specification ψ_i into a vector space. The core network takes as input the state s_i, the program p_i, and the task specification ψ_i, producing the next sub-program to invoke p_{t+1} and an end-of-program probability r_t. The program returns to the caller when r_t exceeds a threshold α (set to 0.5). We detail the inference procedure in Algorithm 1.

NTP vs NPI: We highlight three main differences of NTP than the original NPI: (1) NTP can interpret task specifications and perform hierarchical decomposition and thus can be considered as a meta-policy; (2) it uses APIs as the primitive actions to scale up neural programs for complex tasks; and (3) it uses a reactive core network instead of a recurrent network, making the model less history-dependent, enabling feedback control for recovery from failures. In addition to the three key components, NTP implements two modules similar to the NPI architectures [8, 28]: (1) a domain-specific task encoders that map an observation to a state representation $s_i = f_{ENC}(o_i)$, and (2) a key-value memory that stores and retrieves embeddings: $j^* = \arg\max_{j=1\ldots N}(M^{key}_{j,:} k_i)$ and $p_i = M^{prog}_{j^*,:}$, where k_i is the program key predicted by the core network.

Scaling up NTP with APIs. The bottom-level programs in NPI correspond to primitive actions that are executable in the environment. To scale up neural programs in coping with the complexity of real-world tasks, it is desirable to use existing tools and subroutines (i.e., motion planner) such that learning can be done at an abstracted level. In computer programming, application programming interfaces (APIs) have been a standard protocol of developing software by using basic modules. Here we introduce the concept of API to neural programming, where the bottom-level programs correspond to a set of robot APIs, e.g., moving the robot arm using inverse kinematics. Each API takes specific arguments, e.g., an object category or the end effector's target position. NTP jointly learns to select APIs functions and to generate their input arguments. The APIs that are used in the experiments are `move_to`, `grip`, and `release`. `move_to` takes an object index as the API argument and calls external functions to move the gripper to above the object whose position is either given by the simulator or predicted by an object detector. `grip` closes the gripper and `release` opens the gripper.

Algorithm 1 NTP Inference Procedure

Inputs: task specification ψ, program id i, and environment observation o
function RUN(i, ψ)
  $r \leftarrow 0$, $p \leftarrow M^{prog}_{i,:}$, $s \leftarrow f_{ENC}(o)$, $c \leftarrow f_{TSE}(\psi)$
  while $r < \alpha$ **do**
    $k, r \leftarrow f_{CN}(c, p, s)$, $\psi_2 \leftarrow f_{TSI}(\psi, p, s)$
    $i_2 \leftarrow \arg\max_{j=1\ldots N}(M^{key}_{j,:} k)$
    if program i_2 is primitive **then** ▷ if i_2 is an API
      $\mathbf{a} \leftarrow f_{TSI}(\psi_2, i_2, s)$ ▷ decode API args
      RUN_API($i_2, \mathbf{a}$) ▷ run API i_2 with args $\mathbf{a}$
    else
      RUN(i_2, ψ_2) ▷ run program i_2 w/ task spec ψ_2
    end if
  end while
end function

Task Specification Interpreter. The Task Specification Interpreter, takes a task specification as input, chooses to perform one of the two operations: (1) when the current program p is not primitive, it predicts the sub-task specification for the next sub-program; and (2) when p is primitive (i.e., an API), it predicts the arguments of the API.

Let ψ_i be the task specification of the i-th program call, where ψ_i is a sequence of random variables $\psi_i = \{x_1, x_2, \ldots, x_{N_i}\}$. The next task specification ψ_{i+1} is determined by three inputs: the environment state s_i, the current program p_i, and the current specification ψ_i. When p_i is a primitive, TSI uses an API-specific decoder (i.e., an MLP) to predict the API arguments from the tuple (s_i, p_i, ψ_i).

We focus on the cases when p_i is not primitive. In this case, TSI needs to predict a sub-task specification ψ_{i+1} for the next program p_{i+1}. This sub-task specification should only access relevant information to the sub-task. To encourage information hiding from high-level to low-level programs, we enforce the scoping constraint, such that ψ_{i+1} is a contiguous subsequence of ψ_i. Formally, given $\psi_i = \{x_1, x_2, \ldots, x_{N_i}\}$, the goal is to find the optimal contiguous subsequence $\psi_{i+1} = \{x_p, x_{p+1}, \ldots, x_{q-1}, x_q\}$, where $1 \le p \le q \le N_i$.

Subsequence Selection (Scoping). We use a convolutional architecture to tackle the subsequence selection problem. First, we embed each input element $\psi_i = \{x_1, x_2, \ldots, x_{N_i}\}$ into a vector space $\phi_i = \{w_1, w_2, \ldots, w_{N_i}\}$, where each $w_i \in \mathbb{R}^d$. We perform temporal convolution at every temporal location j of the sequence ϕ_i, where each convolutional kernel is parameterized by $W \in \mathbb{R}^{m \times dk}$ and $b \in \mathbb{R}^m$, which takes a concatenation of k consecutive input elements and produces a single output $y^j_i \in \mathbb{R}^m$. We use *relu* as the nonlinearities. The outputs from all convolutional kernels $\mathbf{y}^j_i$ are concatenated with the program embedding p_i and the encoded states s_i into a single vector $h_j = [p_i; \mathbf{y}^j_i; s_i]$. Finally, we compute the softmax probability of four scoping labels $\Pr_j(l \in \{\texttt{Start}, \texttt{End}, \texttt{Inside}, \texttt{Outside}\})$. These scoping labels indicate whether this temporal location is the start/end

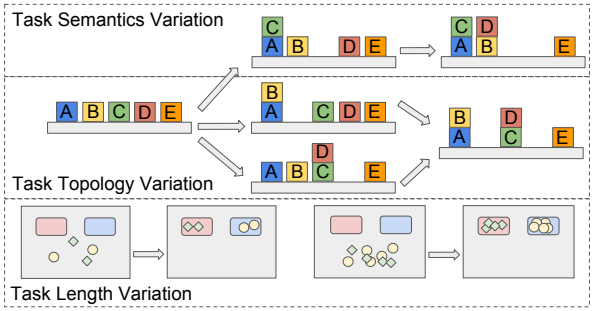

Fig. 4: The variability of a task structure consists of changing success conditions (task semantics), variable subtask permutations (task topology), and larger task sizes (task length). We evaluate the ability of our proposed model in generalizing towards these three types of variations.

point of the correct subsequence, or if it resides inside/outside the subsequence. We use these probabilities to decode the optimal subsequence as the output sub-task specification ψ_{i+1}.

The decoding process can be formulated as the maximum contiguous subsequence sum problem, which can be solved optimally in linear time. However in practice, taking the start and end points with the highest probabilities results in a good performance. In our experiments, we set $\psi_{i+1} = \{x_{\text{st}}, x_{\text{st}+1}, \ldots, x_{\text{ed}}\}$, where $st = \arg\max_{j=1\ldots N_i} \Pr_j(\texttt{Start})$ and $ed = \arg\max_{j=1\ldots N_i} \Pr_j(\texttt{End})$. This process is illustrated in Figure 3, wherein the model factorizes a video sequence which illustrates the procedure of *pick_and_place* into a fraction that only illustrates *pick*. This convolutional TSI architecture is invoked recursively along the program execution trace. It decomposes a long task specification into increasingly fine-grained pieces from high-level to low-level tasks. This method naturally enforces the scoping constraint. Our experimental results show that such information hiding mechanism is crucial to good generalization.

Model Training. We train the model using rich supervision from program execution traces. Each execution trace is a list of tuples $\{\xi_t \,|\, \xi_t = (\psi_t, p_t, s_t), t = 1 \ldots T\}$, where T is the length of the execution trace. Our training objective is to maximize the probability of the correct executions over all the tasks in the dataset $\mathcal{D} = \{(\xi_t, \xi_{t+1})\}$, such that $\theta^* = \Sigma_{\mathcal{D}} \log \Pr[\xi_{t+1} | \xi_t; \theta]$.

We collect a dataset that consists of execution traces from multiple types of tasks and their task specifications. For each specification, we provide the ground-truth hierarchical decomposition of the specification for training by rolling a hard-coded expert policy. We use cross-entropy loss at every temporal location of the task specification to supervise the scoping labels. We also adopted the idea of adaptive curriculum from NPI [28], where the frequency of each mini-batch being fetched is proportional to the model's prediction error with respect to the corresponding program. Full implementation details at `stanfordvl.github.io/ntp/`

V. Experimental Setup

The goal of our experimental evaluation is to answer the following questions: (1) Does NTP generalize to changes in all three dimensions of variation: length, topology, and semantics, as illustrated in Figure 4. (2) Can NTP use image-based input without access to ground truth state. (3) Would NTP also work in real-world tasks which have combinations of these variations. We evaluate NTP in three robot manipulation tasks: Object Sorting, Block Stacking, and Table Clean-up. Each of these tasks requires multiple steps to complete and can be recursively decomposed into repetitive sub-tasks.

Input State Representation. We use an expert policy to generate program execution traces as training data. An expert policy is an agent with hard-coded rules that calls programs (`move_to`, `pick_and_drop`, etc.) to perform a task. In our experiment, we use the demonstration of a robot carrying out a task as the task specification. For all experiments, unless specified, the state representation in the task demonstrations is in the form of object position trajectories relative to the gripper frame. In the Block Stacking experiment, we also report the results of using a learned object detector to predict object locations and the results of directly using RGB video sequence as state observations and task demonstrations.

Simulator Setup. We conduct our experiments in a 3D environment simulated using the Bullet Physics engine [7]. We use a disembodied PR2 gripper for both gathering training data and evaluation. We also evaluate NTP on a simulated 7-DoF Sawyer arm with a parallel-jaw gripper as shown in Figure 1 and Figure 8. Since NTP only considers end-effector pose, the choice of robot does not affect its performance in the simulated environment.

Real-Robot Setup. We also demonstrate NTP's performance on the Block Stacking and the Object Sorting tasks on a 7-DoF Sawyer arm using position control. We use NTP models that are trained with simulated data. Task demonstration are obtained in the simulator, and the instantiated NTP models are executed on the robot. All real-robot experiments use object locations relative to the gripper as state observations. A Kinect2 camera is used to localize objects in the 3D scene.

Evaluation Metrics. We evaluate NTP on three variations of task structure as illustrated in Figure 4: 1) *task length*: varying number of steps due to the increasing problem size (e.g., having more objects to transport); 2) *task topology*: variations in permutations of steps of sub-tasks to reach the same end goal (e.g., manipulating objects in different orders); and 3) *task semantics*: the unseen task objectives and success conditions (e.g., placing objects into a different container).

We evaluate *Task length* on the Object Sorting task varying the number of objects instances from 1 to 10 per category. Further, we evaluate *Task Topology* on the Block Stacking task with different permutations of pick-and-place sub-tasks that lead to the same block configurations. Finally, we evaluate *Task Semantics* on Block Stacking on a held-out set of task demonstrations that lead to unseen block configurations as task objectives.We report success rates for simulation tasks, and we analyze success rates, causes of failure, and proportion of task completed for real-robot evaluation. All objects are randomly placed initially in all of the evaluation tasks in both the simulated and the real-robot setting.

Baselines. We compare NTP to four baselines architecture variations. (1) **Flat** is a non-hierarchical model, similar to [11], that takes as input task demonstration and current observation,

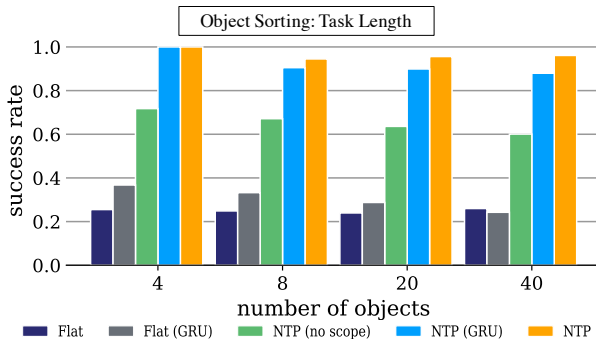

Fig. 5: **Task Length**: Evaluation of the Object Sorting in simulation. The axes represent mean success rate (y) with 100 evaluations each and the number of objects in unseen task instances (x). NTP generalizes to increasingly longer tasks while baselines do not.

and directly predicts the primitive APIs instead of calling hierarchical programs. (2) **Flat (GRU)** is the Flat model with a GRU cell. (3) **NTP (no scope)** is a variant of the NTP model that feeds the entire demonstration to the subprograms, thereby discarding the scoping constraint. (4) **NTP (GRU)** is a complete NTP model with a GRU cell. This is to demonstrate that the reactive core network in NTP can better generalize to longer tasks and recover from unexpected failures due to noise, which is crucial in robot manipulation tasks.

VI. Experiment 1: Object Sorting

Setup. The goal of Object Sorting is to transport objects randomly scattered on a tabletop into their respective shipping containers stated in the task demonstration. We use 4 object categories and 4 containers in evaluating the Object Sorting task. In the real robot setup, a toy duck, toy frog, lego block, and marker are used as the objects for sorting, and are sorted into 4 black plastic bins. This results in a total of $4^4 = 256$ category-container combinations (multiple categories may be mapped to the same container). However, as each category can be mapped to 4 possible containers, a minimum of 4 tasks can cover all possible category-container pairs. We select these 4 tasks for training and the other 252 unseen tasks for evaluation. We train all models with 500 trajectories. Each test run is on 100 randomly-selected unseen tasks.

Simulator. As shown in Figure 5, NTP significantly outperforms the flat baselines. We examine how the task size affects its performance. We vary the numbers of objects to be transported from 4 to 40 in the experiments. The result shows that NTP retains a stable and good performance (over 90%) in longer tasks. On the contrary, the flat models' performances decline from around 40% to around 25%, which is close to random. The performance of the NTP (GRU) model also declines faster comparing to the NTP model as the number of objects increases. This comparison illustrates NTP's ability to generalize towards task length variations.

Real robot. Table I shows the results of the Object Sorting task on the robot. We use 4 object categories with 3 instance of each category. We carried out a total of 10 evaluation trials on randomly selected unseen Object Sorting tasks. 8 trials completed successfully, and 2 failed due to of manipulation failures: a grasp failure and a collision checking failure.

VII. Experiment 2: Block Stacking

Setup. The goal of Block Stacking is to stack a set of blocks into a target configuration, similar to the setup in [11]. We use 8, 5×5 cm wooden cubes of different colors both in simulation and with real-robot. We randomly generate 2000 distinct Block Stacking task instances. Two tasks are considered equivalent if they have the same end configuration. We use a maximum of 1000 training tasks and 100 trials for each task, leaving the remaining 1000 task instances as unseen test cases. A task is considered successful if the end configuration of the blocks matches the task demonstration. We evaluate both seen and unseen tasks, i.e., whether the end configuration appears in training set. We use $N = 8$ blocks in our evaluation.

Simulator. Figure 6 shows that all models except the Flat baseline are able to complete the seen tasks at around 85% success rate. The performance of the Flat baseline decreases dramatically when training with more than 400 tasks. It is because the Flat model has very limited expressiveness power to represent complex tasks. The Flat (GRU) model performs surprisingly well on the seen tasks. However, as shown in Figure 6, both Flat and Flat (GRU) fail to generalize to unseen tasks. We hypothesize that the Flat (GRU) baseline simply memorizes the training sequences. On the other hand, NTP achieves increasingly better performances when the diversity of the training data increases.

We evaluate task topology generalization on random permutations of the pick-and-place sub-tasks that lead to the same end configuration. Specifically, the task variations are generated by randomly shuffling the order that the "block towers" are built in the training tasks. Figure 6 illustrates that NTP generalizes better towards variable topologies when trained on a larger variety of tasks. We find that increasing the diversity of training data facilitates NTP to learn better generalizable modules.

Next we evaluate task semantics generalization. The variability of real-world environments prevents any task-specific policy learning method from training for every possible task. Figure 6(A) illustrates that NTP generalizes well to novel task demonstrations and new goals. As the number of training tasks increases, both NTP and its recurrent variation steadily improve their performance on the unseen tasks. When trained with 1000 tasks, their performances on unseen tasks are almost on par with that of seen tasks.

The performance gaps between NTP (no scope) and NTP highlight the benefit of the scoping constraint. NTP (no scope) performance drops gradually as the task size grows implying that the programs in NTP learn modularized and reusable semantics due to information hiding, which is crucial to achieving generalization towards new tasks.

Real robot. Table I shows the results of the Object Sorting task in the real world setting. We carried out 20 trails of randomly selected unseen Block Stacking tasks. Out of the 2 failure cases, one is caused by a incorrect placing; the other is caused by the gripper knocking down a stacked tower and not able to recover from the error.

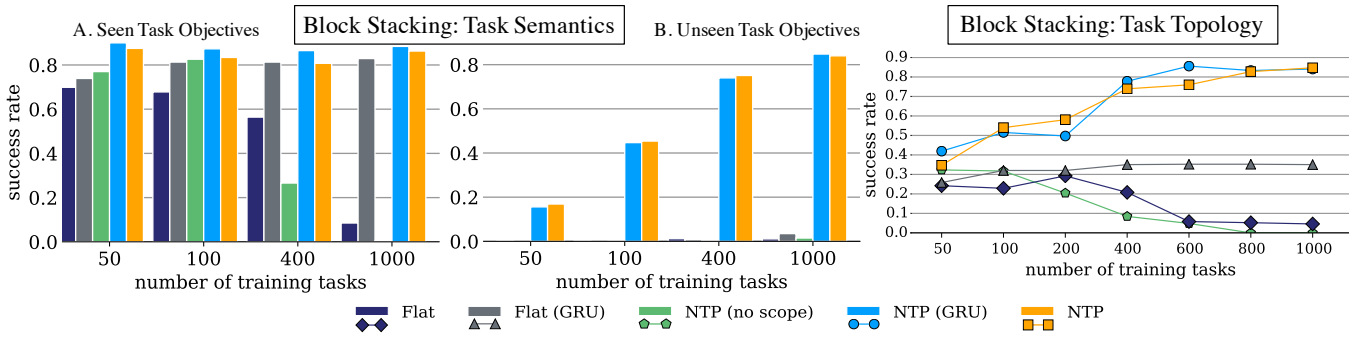

Fig. 6: **Task Semantics**: Simulated evaluation of the Block Stacking. The *x*-axis is the number of tasks used for training. The *y*-axis is the overall success rate. (**A**) and (**B**) show that NTP and its variants generalize better to novel task demonstrations and objectives as the number of training tasks increases.
Task Topology: Simulated evaluation of the Block Stacking. NTP shows better performance in task topology generalization as the number of training tasks grows. In contrast, the flat baselines cannot handle topology variability.

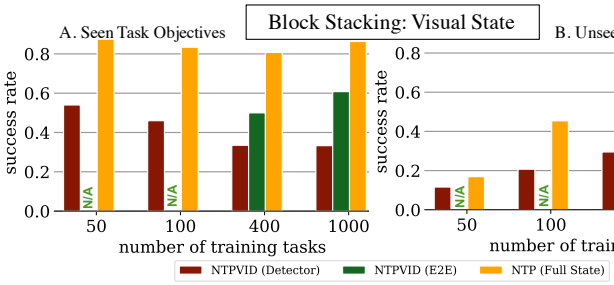

Fig. 7: **NTP with Visual State:** NTPVID (Detector) uses an object detector on images which is subsequently used as state in NTP. NTP (E2E) is an end to end model trained completely on images with no low-level state information. We note that in the partial observation case (only video), similar learning trends were observed as compared to fully observed case (NTP (Full State)), albeit with a decrease in performance.

A. Adversarial Dynamics

We show that the reactive core network in NTP enables it to better recover from failures compared to its recurrent variation. We demonstrate this by performing Block Stacking under an adversary. Upon stacking each block, an adversary applies a force to the towers with a probability of 25%. The force can knock down the towers. We evaluate NTP and its recurrent variant on the 1000 unseen tasks. Table II shows that under the same adversary, the success rate of NTP with the GRU core decreases by 46%, whereas the success rate of NTP only decreases by 20%. This indicates that a reactive model is more robust against unexpected failures as its behavior is less history dependent than the recurrent counterpart. We also demonstrate this feature in the supplementary video in the real world setting.

B. NTP with Visual State

This experiment examines the ability of NTP to learn when demonstrations come in the form of videos and the state is a single image. Unlike the full state information used in experiments thus far, we train an NTP model NTPVID (E2E) to jointly learn a policy and task-relevant features without explicit auxiliary supervision. An alternative is to use a 2-phase pipeline with an object detector as state preprocessor for NTP, termed as NTPVID (Detector). The detector is a separately trained CNN to predict object position in $\mathbb{R}^3$.

We explore these results in Figure 7, where we see compare the visual models (NTPVID (E2E) and NTPVID (Detector)) against the best full state model (NTP), all trained on 100 demonstrations per task, for a varying number of tasks. For NTPVID (E2E) we use a 7-layer convolutional network, which takes as input a (3,64,64) image and outputs a length 128 feature vector. For NTPVID (Detector) , we use a VGG16 based architecture, predicting the position of the *N*-task objects from an input image of size (3,224,224).

We note that NTPVID (E2E) outperforms NTPVID (Detector) and achieves a higher success rate despite only having partial state information. Both of these methods are inferior to the full-state NTP version. NTPVID (Detector) does not generalize due to task specific state representation, and cascading errors in detection propagate to NTP reducing performance even when using a very deep network for the detection. The detector errors are Gaussian with standard deviation of 2 cm. However this performance comes at a computational cost. NTPVID (E2E) was trained on 1000 training tasks for 10 days on 8 Nvidia Titan X GPUs. NTPVID (Detector) was trained for 24 hours on a single GPU. Due to computational cost, we only evaluated NTPVID (E2E) on 400 and 1000 training tasks.

VIII. Experiment 3: Table Clean-up

Setup. We also evaluate NTP on the Table Clean-up task, which combines the features of stacking and sorting and exemplifies a practical real-world task. Specifically, the goal of the task is to clear up to 4 white plastic bowls and 20 red plastic forks into a bin such that the resulting stack of bowls and forks can be steadily carried away in a tray. Task variation comes in task length, where the number of utensils varies, and task topology, where the ordering in which bowls are stacked can vary. Using trajectories as demonstrations and object positions as state space, a model is trained using 1000 task instances.

Simulator. We observe that performance varies between 55%-100% where increasing errors with more objects are attributed to failures in collision checking, not incorrect decisions from NTP. The result shows that NTP retains

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).

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

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.

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

it's generalization ability in a task that requires multiple dimensions of generalization.

Real robot. We have also transferred the trained model on the real-Sawyer arm to evaluate the feasibility as shown in Figure 8. We demonstrate this task in the supplementary video in the real world setting.

IX. Discussion & Future Work

We introduced Neural Task Programming (NTP), a meta-learning framework that learns modular and reusable neural programs for hierarchical tasks. We demonstrate NTP's strengths in three robot manipulation tasks that require prolonged and complex interactions with the environment. NTP achieves generalization towards task length, topology, and semantics. This work opens up the opportunity to use generalizable neural programs for modeling hierarchical tasks. For future work, we intend to 1) improve the state encoder to extract more task-salient information such as object relationships, 2) devise a richer set of APIs such as velocity and torque-based controllers, and 3) extend this framework to tackle more complex tasks on real robots.

Acknowledgment

This research was performed at the SVL at Stanford in affiliation with the Stanford AI Lab, Stanford-Toyota AI Center.

References

[1] J. Andreas, D. Klein, and S. Levine, "Modular multitask reinforcement learning with policy sketches", in *ICML*, 2017.

[2] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Neural module networks", in *CVPR*, 2016.

[3] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A Survey of Robot Learning from Demonstration", *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009.

[4] P. Bacon, J. Harb, and D. Precup, "The option-critic architecture", *ArXiv preprint arXiv:1609.05140*, 2016.

[5] A. Billard, S. Calinon, R. Dillmann, and S. Schaal, "Robot programming by demonstration", in *Handbook of robotics*, 2008.

[6] R. Brooks, "A robust layered control system for a mobile robot", *IEEE journal on robotics and automation*, 1986.

[7] *Bullet Physics Library*, `http://bulletphysics.org/`.

[8] J. Cai, R. Shin, and D. Song, "Making neural programming architectures generalize via recursion", *ICLR*, 2017.

[9] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, "RobustFill: Neural Program Learning under Noisy I/O", *ArXiv preprint arXiv:1703.07469*, 2017.

[10] A. Dosovitskiy and V. Koltun, "Learning to act by predicting the future", *ICLR*, 2017.

[11] Y. Duan, M. Andrychowicz, B. C. Stadie, J. Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba, "One-Shot Imitation Learning", *ArXiv preprint arXiv:1703.07326*, 2017.

# Bowls, # Forks	Success
2 B, 1 F	1.00
2 B, 2 F	0.95
3 B, 2 F	0.75
3 B, 3 F	0.55

Fig. 8: **Table Clean-up**: in simulated and real environment.The table shows success rates for varying numbers of forks and bowls in simulated evaluation.

[12] L. Fei-Fei, R. Fergus, and P. Perona, "One-shot learning of object categories", *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 4, pp. 594–611, 2006.

[13] R. E. Fikes, P. E. Hart, and N. J. Nilsson, "Learning and executing generalized robot plans", *ARTIFICIAL INTELLIGENCE*, 1972.

[14] R. Fox, S. Krishnan, I. Stoica, and K. Goldberg, "Multi-Level Discovery of Deep Options", in *Preprint arXiv:1703.08294*, 2017.

[15] C. L. Giles, C. B. Miller, D. Chen, H.-H. Chen, G.-Z. Sun, and Y.-C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks", *Learning*, vol. 4, no. 3, 2008.

[16] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines", *ArXiv preprint arXiv:1410.5401*, 2014.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[18] T. M. Jochem, D. A. Pomerleau, and C. E. Thorpe, "Maniac: A next generation neurally based autonomous road follower", in *Int'l Conf. on Intelligent Autonomous Systems*, IOS Publishers, Amsterdam., Pittsburgh, PA, 1993, pp. 15–18.

[19] K. Judah, A. P. Fern, P. Tadepalli, and R. Goetschalckx, "Imitation learning with demonstrations and shaping rewards", in *AAAI*, 2014.

[20] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey", *The Int'l Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[21] S. Krishnan*, A. Garg*, S. Patil, C. Lea, G. Hager, P. Abbeel, and K. Goldberg, "Transition state clustering: Unsupervised surgical trajectory segmentation for robot learning", *IJRR*, 2018.

[22] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation", in *NIPS*, 2016.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[24] A. Murali, S. Sen, B. Kehoe, A. Garg, S. McFarland, *et al.*, "Learning by observation for surgical subtasks: Multilateral cutting of 3d viscoelastic and 2d orthotropic tissue phantoms", in *IEEE Int'l Conf. on Robotics and Automation, ICRA*, 2015.

[25] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping", in *ICML*, 1999, pp. 278–287.

[26] A. Y. Ng and S. J. Russell, "Algorithms for inverse reinforcement learning.", in *ICML*, 2000.

[27] R. Parr and S. J. Russell, "Reinforcement learning with hierarchies of machines", in *NIPS*, 1998.

[28] S. Reed and N. de Freitas, "Neural programmer-interpreters", in *ICLR*, 2016.

[29] S. Ross, G. J. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning", in *Int'l Conf. on Artificial Intelligence and Statistics*, 2011.

[30] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators", in *ICML*, 2015.

[31] S. Sen*, A. Garg*, D. Gealy, S. McKinley, Y. Jen, and K. Goldberg, "Autonomous Multiple-Throw Multilateral Surgical Suturing with a Mechanical Needle Guide and Optimization based Needle Planning", in *IEEE Int. Conf. Robotics and Automation (ICRA)*, 2016.

[32] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning", *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[33] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor, "A deep hierarchical approach to lifelong learning in minecraft", *ArXiv preprint arXiv:1604.07255*, 2016.

[34] R. Vilalta and Y. Drissi, "A perspective view and survey of meta-learning", *Artificial Intelligence Review*, 2002.

[35] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, *et al.*, "Matching networks for one shot learning", in *Advances in Neural Information Processing Systems*, 2016, pp. 3630–3638.

[36] Y. Wu and Y. Demiris, "Towards one shot learning by imitation for humanoid robots", in *ICRA*, 2010.
[37] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning", in *ICRA*, 2017.