Hierarchical Task Generalization with Neural Programs

Danfei Xu, Yuke Zhu, Animesh Garg, Julian Gao, Li Fei-Fei, and Silvio Savarese
Department of Computer Science, Stanford University

Abstract—In this work, we propose a novel modeling framework for hierarchical tasks called Neural Task Programming (NTP). We bridge the idea of meta-learning and neural programming. NTP takes as input a task specification and recursively decomposes it into finer subtask specifications. These specifications are fed to a hierarchical neural program, where bottom-level programs are callable subroutines to interact with the environment. We validate our method in two robot manipulation tasks. NTP achieves strong generalization results across sequential tasks that exhibit a hierarchical and compositional structure. The experimental results show that NTP learns to generalize well towards unseen tasks with increasing lengths, variable topologies, and changing objectives.

I. INTRODUCTION

Generalizable algorithms, that enable synthesis and execution of an abstract and hierarchical task plan conditioned on the environment context, are highly desired in a multitude of AI problems – such as symbolic reasoning, sequential decision making & reinforcement learning, and motor control. Learning to perform such tasks and, more importantly, efficient adaption to new tasks is an age-old problem needed to impart intelligence in artificial agents.

In this paper, we learn to generalize across sequential tasks that exhibit a hierarchical and compositional structure. A concrete instantiation to study this learning problem is robot manipulation of order fulfillment conditioned on a task specification, where the goal is to transport all objects from each category into a specified shipping container, illustrated in Fig. 1. Each task can be composed from lower-level hierarchical sub-tasks such as pick, move, and place. Additionally, this problem has a rich task space with numerous task specifications, each with many ways to complete. This paper develops a hierarchical meta-learning approach Neural Task Programming (NTP). This approach generalizes well to three main characteristics of task structure: 1) task length: the growing 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., choosing a different ordering of picking objects); and 3) task semantics: the varying task definitions and success conditions (e.g., placing objects into a different container).

Many studies have analyzed the problem of knowledge transfer and generalization across tasks with deep networks exploring data randomization [20], reusing weights [9, 17], and decoupling task embeddings from environment dynamics [3, 13]. But these methods often fall short in the face of complex cognitive reasoning tasks that often exhibit a hierarchical and compositional structure. A common treatment for this problem class is reinforcement learning methods, especially hierarchical variants of RL that handle decomposition through multi-stage policies operating over options [2] [13] [19] [21]. However, RL often is limited to task-specific policy learning. To date, achieving efficient generalization across tasks is often an open area of research. Recent methods have adopted a learning-to-learn or “meta-learning” approach, which instantiates a task-specific policy based on a goal description or a task specification [7, 8, 15, 13, 22]. While a promising direction, few meta-learning results address learning and transfer in hierarchical tasks.

At the same time, research on modularized deep learning has opened up to ideas of information hiding, modularization, and abstraction, which were adopted in computer programming. Recent development in neural programming aims at marrying these nice properties of computer programs with a neural network backend. This line of work has shown great potential to handle multimodal data [16], guarantee symbolic generalization [4], and learn well from noisy input/output examples [6].

Inspired by ideas in meta-learning, hierarchical RL, and neural programming, NTP is a general and 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 (Fig. 1[1] left) and instantiates a hierarchical policy as a neural program (Fig. 1[1] middle), where the bottom-level programs are primitive actions that are executable in the environment. It decodes the objective of a task from an input specification and factorizes it into finer sub-tasks, interacting with the environment in a reactive manner until the goal is achieved. Each program call takes as input the environment observation and a task specification, producing the next sub-program and its corresponding sub-task specification. This hierarchical decomposition encourages information hiding and modularization, as lower-level modules only access their corresponding sub-task specifications that pertain to their functionalities. 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.

The primary contribution of this paper is a novel modeling framework NTP that enables meta-learning on hierarchical tasks. We evaluate NTP in modeling two single-arm manipulation tasks: block stacking and order fulfillment. We show that NTP enables knowledge transfer to novel tasks with increasing lengths, variable topologies, and changing objectives. We investigate NTP’s generalization power with respect to changes in environments (e.g., initial configurations and appearances) and changes in task structures, especially for the latter. NTP shows strong generalization results regarding three main aspects of the task structure, including task length, task topology, and task semantics.

II. Problem Formulation

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

Concretely, a task specification $\psi(t)$ can be a video sequence or a list of language instructions that describes the procedure of the sequential task. In many real-world tasks, the agent has no access to the underlying environment states. It only receives a sample of environment observation $o$ that corresponds to the state $s$, where $O$ is the observation space. Our goal is to learn a “meta policy” that instantiates an executable policy from a specification of a task, $\pi_{\text{meta}} : \Psi \rightarrow (O \rightarrow A)$. At test time, a specification of a new task $\psi(t)$ is fed to NTP. The meta policy generates a policy $\pi(a|o; \psi(t)) : O \rightarrow A$, which maps observations to actions that leads to a task-completion state $s_T$ when $g(s_T, t) = 1$.

Why is Neural Programming a Good Idea for Meta-Learning? Previous work has mostly used a monolithic network architecture to represent a goal-driven meta policy [7][18, [22]. These methods cannot exploit the compositional task structures to facilitate modularization and reusability. Instead, we represent our meta policy $\pi_{\text{meta}}$ as a neural program that takes a task specification as its input argument. NTP uses a task-agnostic core network to decide which sub-program to run next and adaptively feeds a subset of the task specification that it receives to the next program. Intuitively, it recursively decomposes a task specification and solves a hierarchical task by divide-and-conquer. Our method extends upon a special type of neural programming architectures, based on Neural Programmer-Interpreter (NPI) [4][16]. NPI generalizes well to increasing task sizes. However, it lacks a mechanism to parse a novel task specification and complete a novel task objective on demand. NTP combines the idea of meta-learning and NPI. The ability to understand task specifications and to construct neural programs on demand makes NTP generalize well across tasks.

Overview of NPI. Before introducing our NTP model, it is useful to briefly overview the Neural Programmer-Interpreter paradigm [16]. The core of NPI is a long-short memory (LSTM) [11] 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

Algorithm 1 NTP Inference Procedure

<table>
<thead>
<tr>
<th>Inputs:</th>
<th>task specification $\psi$, program id $i$, and environment observation $o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>function RUN($i$, $\psi$)</td>
<td></td>
</tr>
<tr>
<td>$r \leftarrow 0$, $p \leftarrow M^{prog}<em>i$, $s \leftarrow f</em>{ENC}(o)$, $c \leftarrow f_{TSE}(\psi)$</td>
<td></td>
</tr>
<tr>
<td>while $r &lt; \alpha$ do</td>
<td></td>
</tr>
<tr>
<td>$k$, $r \leftarrow f_{CN}(c, p, s)$, $\psi_2 \leftarrow f_{FSI}(\psi, p, s)$</td>
<td></td>
</tr>
<tr>
<td>$i_2 \leftarrow \text{arg max}_{j=1..N}(M^{key}_j, k)$</td>
<td></td>
</tr>
<tr>
<td>if program $i_2$ is primitive then</td>
<td></td>
</tr>
<tr>
<td>$a \leftarrow f_{TSI}(\psi_2, i_2, s)$</td>
<td></td>
</tr>
<tr>
<td>$\text{RUN API}(i_2, a)$</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>RUN($i_2, \psi_2$)</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>end while</td>
<td></td>
</tr>
<tr>
<td>end function</td>
<td></td>
</tr>
</tbody>
</table>
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 \( \alpha \) (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.

### III. NEURAL TASK PROGRAMMING

#### Overview.

NTP has three key components: Task Specification Interpreter \( f_{TSI} \), Task Specification Encoder \( f_{TSE} \), and a core network \( f_{CN} \). The Task Specification Encoder transforms a task specification \( \psi \) into a vector space. The core network takes as input the state \( s_i \), the program \( p_i \), and the task specification \( \psi_i \), producing the next sub-program to invoke \( p_{i+1} \) and an end-of-program probability \( r_i \). The program returns to the caller when \( r_i \) exceeds a threshold \( \alpha \) (set to 0.5). We detail the inference procedure as a function \( \text{RUN} \) in Algorithm 1. We highlight three main differences of NTP than the original NPI: 1) NTP takes hierarchical task specifications as the arguments to the programs, 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 and consequently more robust towards unexpected failures in real-world interactions. In addition to these three key components, NTP also implements two modules similar to previous NPI architectures [8, 16], including 1) the domain-specific task encoders that map an observation to a state representation \( s_i = f_{ENC}(o_i) \), and 2) the key-value memory that stores and retrieves embeddings:

\[
j^* = \arg \max_{j=1, \ldots, N} (M_j^{\text{prog}} k_i) \quad \text{and} \quad p_i = M_j^{\text{prog}},
\]

where \( k_i \) is the program key predicted from the core network.

#### Scaling Up NTP with APIs.

The bottom-level programs in NPI correspond to primitive actions that are executable in the environment. In scaling up neural programs to cope with the complexity of real-world tasks, it is desirable to use existing subroutines such that learning can be done at a high abstraction 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 primitive programs correspond to a set of APIs, which performs some form of computation (e.g., detecting an object) or interaction with the environment (e.g., moving the robot arm using inverse kinematics). Each API takes their own types of arguments, e.g., an object category or the end effector’s target position. NTP learns to select APIs and to generate their input arguments.

#### Task Specification Interpreter.

Our method learns a meta policy that takes as input a task specification. The Task Specification Interpreter performs one of the two things: 1) when the current program \( p_i \) is not primitive, it predicts the sub-task specification for the next sub-program; and 2) when \( p_i \) is primitive (i.e., an API), it predicts the arguments as input to the API.

Let \( \psi_i \) be the task specification of the \( i \)-th program call, where \( \psi_i \) is a sequence of random variables \( \psi_i = \{x_1, x_2, \ldots, x_N\} \). In practice, we use a video illustration as task specification, which describes the procedure of the target. In this case, each \( x_j \) corresponds to \( j \)-th frame in the video. However, our formulation is extensible to other forms of sequential specifications, e.g., language instructions or user manuals. The next task specification \( \psi_{i+1} \) is determined by three inputs: the environment state \( s_i \), the current program \( p_i \), and the current specification \( \psi_i \). When \( p_i \) is a primitive, TSI uses an API-specific decoder (i.e., an MLP) to predict the API arguments from the tuple \( (s_i, p_i, \psi_i) \).

We focus on the cases when \( p_i \) is not primitive. In this case, TSI needs to predict a sub-task specification \( \psi_{i+1} \) 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 stripping constraint, such that \( \psi_{i+1} \) is a contiguous subsequence of \( \psi_i \). Formally, given \( \psi_i = \{x_1, x_2, \ldots, x_N\} \), the goal is to find the optimal contiguous subsequence \( \psi_{i+1} = \{x_p, x_{p+1}, \ldots, x_{q-1}, x_q\} \), where \( 1 \leq p \leq q \leq N \).

#### Subsequence Selection.

We use a convolutional architecture to tackle the subsequence selection problem. First, we embed each input element \( \psi_i = \{x_1, x_2, \ldots, x_N\} \) into a vector space \( \phi_i = \{w_1, w_2, \ldots, w_N\} \), where each \( w_i \in \mathbb{R}^d \). We perform temporal convolution at every temporal location \( j \) of the sequence \( \phi_i \), 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 \( y^i_j \in \mathbb{R}^m \). We use \( \text{relu} \) as the nonlinearities. The outputs from all convolutional kernels \( y^i_j \) are concatenated with the program embedding \( p_i \) and the encoded states \( s_i \) into a single vector \( h_j = [p_i; y^i_j; s_i] \). Finally, we compute the softmax probability of four scoping labels \( \text{Pr}_j(l \in \{\text{Start}, \text{End}, \text{Inside}, \text{Outside}\}) \). These scoping labels indicate if this temporal location is the start/end point of the correct subsequence, or if it resides inside or outside the subsequence. We use these probabilities to decode the optimal
subsequence as the output sub-task specification \( \psi_{i+1} \).

The decoding process can be formulated as the maximum contiguous subsequence sum problem, which can be solved 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 \( \psi_{i+1} = \{x_{st}, x_{st+1}, \ldots, x_{ed}\} \), where \( st = \arg \max_{j=1, \ldots, N} \Pr_j(\text{Start}) \) and \( ed = \arg \max_{j=1, \ldots, N} \Pr_j(\text{End}) \). This process is illustrated in Fig. 2. In this example, the model factorizes a video sequence which illustrates the procedure of \textit{pick and drop} into a fraction that only illustrates \textit{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 supervision from program execution traces. Each execution trace is a list of tuples \( \{\xi_t = (\psi_t, p_t, s_t), t = 1 \ldots T\} \), where \( T \) is the length of the trace. Our training objective is to maximize the probability of the correct executions over all the tasks in the dataset \( D = \{\{\xi_t, \xi_{t+1}\}\} \), such that \( \theta^* = \sum_{t} \log \Pr[\xi_{t+1} | \xi_t; \theta] \). We collect a set of execution traces on a variety of tasks for training.

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. We use cross-entropy loss at every temporal location of the task specification to supervise the decoding labels. We also adopted the idea of adaptive curriculum from NPI [16], where the frequency of each mini-batch being fetched is proportional to the model’s prediction error with respect to the corresponding program.

### IV. Experiments

We evaluate NTP in two robot manipulation tasks, order fulfillment and block stacking. The goal of order fulfillment is to transport objects randomly scattered on a tabletop into their respective shipping containers stated in the task specification. The goal of block stacking is to stack a set of cube blocks into a target configuration, similar to the setup in Duan et al. [8]. Both tasks require prolonged and complex task specifications with the environment. At the same time, they can be recursively decomposed into simpler and repetitive sub-tasks. We use video sequences as the form of task specification, as it can be easily collected while describing the full details of the task procedure.

We conduct our experiments in a 3D physical environment simulated by the Bullet Physics engine [11]. The experiment environment consists of a single-armed Sawyer robot that interacts with other objects on a tabletop (see Fig. 3). We use an expert policy to generate program execution traces that complete the goals described in an input task specification. For each task, the expert policy invokes a hierarchy of programs to generate example execution traces for supervised training.

![Block Stacking](image1.png) ![Order Fulfillment](image2.png)

**Fig. 3:** We evaluate NTP and the baseline models in the order fulfillment and the block stacking tasks.

![Task Semantics Variation](image3.png) ![Task Topology Variation](image4.png) ![Task Length Variation](image5.png)

**Fig. 4:** The variability of a task structure consists of changing success conditions (task semantics), 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.

#### A. Evaluation Protocol

**Evaluation Metrics.** We evaluate NTP’s ability of task generalization regarding three main aspects of the task structure, including task length, task topology, and task semantics, depicted in Fig. 4. Previous works on neural programming [4, 10, 12, 16] have mostly focused on evaluating generalization of the model’s behavior with respect to tasks of variable sizes (e.g., length of the array for bubble sort). We call this metric \textit{task length generalization}. We evaluate this metric in the order fulfillment tasks. We train all models in tasks that transport a single object instance per category. They are evaluated on longer tasks that transport a varying number of objects (from 1 to 10 per category).

In addition to task length, a key difference of NTP than previous work is the meta policy that instantiates policies from task specifications. It enables NTP to generalize towards two additional aspects of task structure, i.e., task topology and task semantics. We evaluate these two metrics in the block stacking tasks. \textit{Task topology generalization} aims to complete unseen tasks stated by a novel task specification. We evaluate on a held-out set of task specifications that lead to unseen block configurations.

**Baselines.** We compare NTP with four baselines. To examine the effect of a hierarchical model, our first baseline is a flat model (Flat) that takes as input the task specification and the current observation and directly predicts the primitive APIs instead of calling hierarchical programs. We also investigate the value of the scoping constraint by comparing NTP with a baseline that feeds the entire specification to the subprograms.
Fig. 5: Results of the block stacking tasks. (a),(b): The $x$-axis is the number of tasks used for training. The $y$-axis is the overall success rate. NTP generalizes better to novel task specifications and new goals as the number of training tasks increases. (c) Results of topology generalization. NTP shows better performance in task topology generalization as the number of training tasks grows. In contrast, the flat baselines cannot handle topology variability.

B. Order Fulfillment

The task specification for order fulfillment is a video sequence that shows an object instance per category being transported to their target shipping containers. We use an ad-hoc object detector to pre-process the video to get a list of object locations. The environment observation consists of the $(x, y, z)$ location of the object closest to the gripper in the gripper frame per category as well as a binary value indicating if the gripper is closed. At test time, the agent receives a task specification and current observation as input and transports all objects to their corresponding shipping containers.

We use $N = 4$ object categories. This results in a total of $4^4 = 256$ category-container combinations. 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 trials.

We report the results of the order fulfillment tasks in Fig. 6. 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 declines from around %40 to around %25, which is close to random. The performance of the NTP (GRU) model also declines faster than that of the NTP model as the number of objects increases. This comparison illustrates NTP’s ability to generalize towards task length variations.

C. Block Stacking

The task specification for block stacking is a video sequence that illustrates the procedure of stacking blocks into towers. At test time, the agent receives a task specification and replicates the procedure starting from a random initial configuration.

We use a color-based object detector to track each block in 3D space. The environment observation consists of the $(x, y, z)$ locations of all blocks in the gripper frame along with a binary value indicating if the gripper is closed. We randomly generate 2000 distinct block stacking tasks. We consider two tasks equivalent if they have the same end configuration. We use a maximum of 1000 training tasks and 100 trials for each task. A task is considered successful if the end configuration of the blocks matches the task specification. 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.

Fig. 5 reports the results of the block stacking tasks. Fig. 5(a) 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 Fig. 5(b), 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 subtasks 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. Fig. 5(c) 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 more generalizable modules.

Next we evaluate task semantics generalization. The variability of real-world environment prevents any model from training for every possible task. NTP achieves semantics generalization by having the Task Specification Interpreter learn to parse unseen specifications. Fig. 5(b) illustrates that NTP generalizes well to novel task specifications 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 illustrate the benefit of our proposed scoping constraint. The performance of the NTP (no scope) baseline drops gradually as the task size grows. It implies that the programs in NTP learn modularized and reusable semantics due to information hiding, which is crucial to generalization towards new tasks.

V. CONCLUSION

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 two robot manipulation tasks that require prolonged and complex interactions with the environment. NTP achieves strong generalization results towards task length, topology, and semantics. This work opens up the opportunity to use generalizable neural programs for modeling hierarchical tasks. We intend to extend this framework to tackle a richer set of complex tasks on real robots in future work.

REFERENCES


