Parametric Visual Program Induction with Function Modularization

Xuguang Duan [1] **Xin Wang** [1] **Ziwei Zhang** [1] **Wenwu Zhu** [1]

Abstract

Generating programs to describe visual observations has gained much research attention recently. However, most of the existing approaches are based on non-parametric primitive functions, making them unable to handle complex visual scenes involving many attributes and details. In this paper, we propose the concept of parametric visual program induction. Learning to generate parametric programs for visual scenes is challenging due to the huge number of function variants and the complex function correlations. To solve these challenges, we propose the method of function modularization, capable of dealing with numerous function variants and complex correlations. Specifically, we model each parametric function as a multi-head self-contained neural module to cover different function variants. Moreover, to eliminate the complex correlations between functions, we propose the hierarchical heterogeneous Monto-Carlo tree search (H2MCTS) algorithm which can provide high-quality uncorrelated supervision during training, and serve as an efficient searching technique during testing. We demonstrate the superiority of the proposed method on three visual program induction datasets involving parametric primitive functions. Experimental results show that our proposed model is able to significantly outperform the state-of-the-art baseline methods in terms of generating accurate programs.

1. Introduction

Studying how to generate computer-executable programs is one of the core interests of the AI community (Waldinger & Lee, 1969; Manna & Waldinger, 1975), and has drawn lots of recent interests in the visual domain thanks to deep learning (Ellis et al., 2020). By leveraging powerful deep models, these works can successfully describe the logic behind visual games (Sun et al., 2018), learn spatial patterns hidden in images (Young et al., 2019), or conduct neural-symbolic reasoning (Yi et al., 2018).

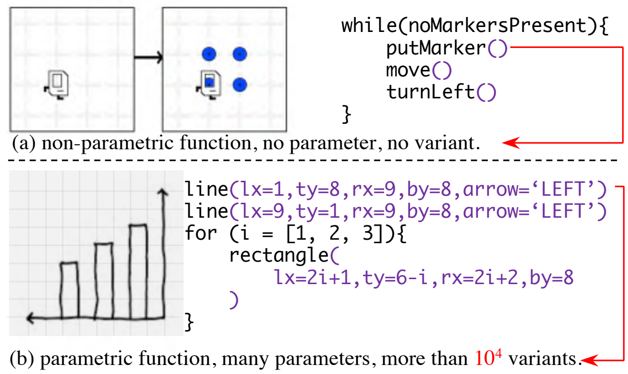


Figure 1. (a): An example of the visual program induction task that only generates non-parametric programs, within which, each function has only one variant, and could be modeled as a symbolic token. (b): An example of the parametric visual program induction task studied in this paper, where parametric primitive functions with many more variants are needed to describe the complex visual scene. However, it is hard to tackle such many function variants.

Despite their enormous success, most of the existing approaches are based on non-parametric primitive functions, failing to meet the requirement of the increasing complexity of visual observations, as well as the increasing elaboration of programs. In this paper, we are the first to propose the concept of **Parametric Visual Program Induction**, *i.e.*, *generating programs with parametric primitive functions for complex visual observations*, to the best of our knowledge. By leveraging parametric primitive functions, we can generate much more detailed programs to describe both the hidden logic and visual details.

However, the challenges for solving parametric program induction are two folds. First, the action space for a single function can be huge. Compared with basic non-parametric primitive functions, the parametric primitive functions always have several heterogeneous parameters, resulting in a huge number of function variants. For example, in Figure 1(a), a basic visual program induction task may contain simple primitive functions such as `move()`,

[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China. Correspondence to: Xin Wang <xin_wang@tsinghua.edu.cn>, Wenwu Zhu <wwzhu@tsinghua.edu.cn>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

`turnLeft()`; while in Figure 1(b), a parametric function studied in this work tend to have more than 10^4 variants due to different parameter combinations.

Second, the function space for the whole program is also very huge. Given that parametric functions may contain multiple parameters, and these parameters and functions are correlated together, it becomes very challenging to model the long-range function transitions within a program. This problem is also known as *program aliasing* (Bunel et al., 2018) in the non-parametric scenario, and becomes more severe for parametric functions.

These two challenges make non-parametric visual program induction methods hard to extend to the parametric domain. To address these challenges, we propose the concept and method of **Function Modularization**, which can model numerous and complex parametric functions. In particular, we treat each function along with its parameters as a self-contained module and learn the module to predict the correct parameters given visual contexts, which is able to solve the challenge of the huge action space. Furthermore, based on the modularized functions, we propose a Hierarchical Heterogeneous Monto-Carlo Tree-Search (H2MCTS) algorithm that can traversal all the program aliases, thus providing uncorrelated training data during training and serving as a powerful search method during inference. To verify the superiority of the concept of **function modularization** and the efficiency of the H2MCTS algorithm, we conduct extensive experiments on a small hand-craft dataset and two well-known datasets (Ellis et al., 2018; Dong et al., 2019). Experimental results show that a modularized function is easier to learn and has higher accuracy compared with vanilla baselines. Also, the proposed H2MCTS algorithm is able to efficiently search over different function combinations and reduce the inference time significantly. In summary, we make the following contributions:

- To the best of our knowledge, we are the first to investigate the problem of parametric visual program induction by proposing the concept and method of **Function Modularization**, which decouples the learning of function parameters and function transitions, resulting in accurate and efficient learning of the parametric programs.

- We propose the H2MCTS algorithm to assist the learning of modularized functions. Our proposed algorithm can provide uncorrelated data to train modularized functions and serve as an efficient search method during inference.

- We conduct extensive experiments to demonstrate that our proposed model can significantly outperform state-of-the-art baselines on all three datasets.

2. Related Work

Learning to generate programs has a long history in AI (Waldinger & Lee, 1969; Manna & Waldinger, 1975; 1980). Traditionally, the process of generating programs is based on search-based induction, and one of the most famous works is the Excel *FlashFill* system (Gulwani, 2011). These methods rely on syntax-based pruning (Feser et al., 2015), or use satisfiability modulo theories-based solvers (Lezama, 2008; Feser et al., 2015). With the development of deep learning, this area has gained new attention as learning to generate a program from data directly (Parisotto et al., 2017; Devlin et al., 2017; Ling et al., 2017; Chollet, 2019), including previously unsolvable visual domain tasks (Bunel et al., 2018; Sun et al., 2018; Shin et al., 2019). Besides, the combination of search and learning is also appealing by leveraging advantages from both sides by combining learning and searching (Balog et al., 2016; Irving et al., 2016; Ellis et al., 2020). Balog et al. (2016) and Irving et al. (2016) propose to use neural networks to predict the probability of the next word, and lead into s guided-search schema; Ellis et al. (2020) propose the EC2 algorithm to iteratively learn and search over a Domain-Specific Language. Despite the success of these methods, most existing approaches work with non-parametric or few-parameter primitive functions and solve the task by treating programs as "a sequence of tokens" to learn the token transition dynamics, which cannot effectively handle parametric programs. Besides, Nye et al. (2019) had also tried to solve the problem of generating complex programs, which focus on generating longer programs with complex control flows by proposing a series of control-flow sketches and learning to fill the "sketch-hole".

Besides, as visual scenes become prevalent, researchers start to work with much more complex visual scenes like LaTeX drawings and computer-aided design objects (Eslami et al., 2016; Ellis et al., 2018; Young et al., 2019; Tian et al., 2019; Zhou et al., 2021). Most of these tasks are based on parametric functions and thus make the traditional view of "*treating the program as a sequence of tokens*" collapse due to a large number of variants of parametric functions. Ellis et al. (2018) uses STN (Jaderberg et al., 2015) to model multiple parameters, Tian et al. (2019) aligns all the function parameters such that they could be modeled with the same neural network, while Zhou et al. (2021) uses a grammar-encoded LSTM model. Though obtained remarkable results, these methods are not easy to generalize.

Compared with existing methods, we follow the combination of learning and searching, while, at the same time, tackling those parametric primitive functions. We propose to model each parametric function along with its parameters as a module and propose the H2MCTS algorithm that could benefit both training and inference.

3. Problem Formulation

3.1. Notations and Problem Formulation

Following Piantadosi (2011), we define a program as a logical collection of primitive functions. Specifically, given a set of primitive functions $\mathbb{F}$, a program $\mathcal{P} = (f_1^{\mathbf{\Theta}_1}, f_2^{\mathbf{\Theta}_2}, \cdots, f_T^{\mathbf{\Theta}_T})$, where $f_i^{\mathbf{\Theta}_i} \in \mathbb{F}$ is a primitive function f with parameters $\mathbf{\Theta}_i = (\Theta_{i,0}, \Theta_{i,1}, \cdots, \Theta_{i,n_f})$, n_f is the number of parameters for f, and T is a program-dependent parameter that indicates the length of the program $\mathcal{P}$. Besides, in the main text of this paper, we focus on the parametric functions and simplify our program syntax as context-free grammar (CFG) (Zhou et al., 2021), *i.e.*, programs without `loops` and other control commands, and we show in the experiments (Section 6.3) and Appendix B that our method could be easily extended to context-based scenarios.

The task of parametric visual program induction is defined as: given an input-output observation pair $(\mathcal{O}_I, \mathcal{O}_O)$, find a parametric program $\mathcal{P}$ to transform the input to the output:

$$\mathcal{P}(\mathcal{O}_I) \rightarrow \mathcal{O}_O. \tag{1}$$

Moreover, based on CFG, Eq. (1) can be rewritten as

$$f_T^{\mathbf{\Theta}_T} \circ f_{T-1}^{\mathbf{\Theta}_{T-1}} \cdots \circ f_1^{\mathbf{\Theta}_1}(\mathcal{O}_I) \rightarrow \mathcal{O}_O, \tag{2}$$

where $f_i^{\mathbf{\Theta}_i} \circ f_j^{\mathbf{\Theta}_j}$ is the composition of two functions, *i.e.*, $f_i^{\mathbf{\Theta}_i} \circ f_j^{\mathbf{\Theta}_j}(O_{in}) \doteq f_i^{\mathbf{\Theta}_i}\left(f_j^{\mathbf{\Theta}_j}(O_{in})\right)$.

3.2. The Existing Methods

To generate the desired program $\mathcal{P}$ in Eq. (1), most of the existing works adopt the method of tokenization, *i.e.*, transforming $(f_1^{\mathbf{\Theta}_1}, f_2^{\mathbf{\Theta}_2}, \cdots, f_T^{\mathbf{\Theta}_T})$ into $(t_1, t_2, ...t_N)$, where t_i is a token and N is the number of tokens. The probability of program $\mathcal{P}$ is calculated by assuming the Markov property:

$$\Pr[\mathcal{P}|\mathcal{O}_I, \mathcal{O}_O] = \prod_{i=1}^{N} P(t_i|t_{<i}; \mathcal{O}_I, \mathcal{O}_O), \tag{3}$$

where P is a conditional Markov transition probability. To tackle the problem, traditional rule-based search methods (Manna & Waldinger, 1980) adopt syntax-pruned search strategies, while recent neural program synthesis methods (Bunel et al., 2018) learn the probability function with language models (Figure 2 (a) and (b)). Though this Markov chain modeling works well with zero-parameter functions by treating each function as a token, such approaches encounter great difficulties in modeling parametric functions. Considering an example function of "`dot(x,y,col)`". If each function variant is treated as a token, *e.g.*, "`dot(x=1,y=2,col=red)`" is a token, the set of tokens becomes too large considering different parameter combination. On the other hand, if each function fragment is treated as a token, e.g., "`dot`", "`(`", "`x=1`", "`y=2`", "`col=red`", "`)`" as 6 tokens, the generated program may be very long ($N \gg T$) and suffer syntax-error.

To train P in Eq. (3), synthetic data is generated and utilized (Shin et al., 2019). Specifically, a random program $\mathcal{P}$ is first generated. Then, by inputting a random visual observation $\mathcal{O}_I$, the corresponding $\mathcal{O}_O$ is obtained as $\mathcal{O}_O = \mathcal{P}(\mathcal{O}_I)$. $\mathcal{O}_I, \mathcal{O}_O, \mathcal{P}$ are used as ground-truths to train the model by maximizing the posterior probability. However, training from such synthetic data is biased due to the distribution mismatch between the random program and true programs (Shin et al., 2019). Besides, due to the program aliasing problem (Bunel et al., 2018), naively using the generated data will lead to inefficient training.

4. Function Modularization

In this section, we tackle the problem of learning parametric functions by function modularization. Specifically, we transform the parametric program induction problem as learning inter-function transition and intra-function parameter prediction. The former focuses on selecting which function should be used, and the latter models each function along with its parameters as a self-contained module to obtain the most suitable parameters for that function.

4.1. Function Transition and Parameter Prediction

The goal of function modularization is to separate the learning of the program into two parts: **inter-function transition** and **intra-function parameter prediction** as:

$$\Pr[\mathcal{P}|\mathcal{O}_I, \mathcal{O}_O] = \prod_{i=1}^{T} P(f_i|\hat{\mathcal{O}}_{i-1}) \cdot Q_{f_i}(\mathbf{\Theta}_i|\hat{\mathcal{O}}_{i-1}), \tag{4}$$

where $\hat{\mathcal{O}}_i = (\mathcal{O}_i, \mathcal{O}_O)$ is the pair of the observation at the i-th step and the target output, $\mathcal{O}_i = f_i^{\mathbf{\Theta}_i}(\mathcal{O}_{i-1})$, and P and Q are two learnable probabilities. By this transformation, we could decouple the learning objective into:

$$\begin{aligned} &\text{(Function Transition) } P \ \ : (\mathcal{O}_t, \mathcal{O}_O) \rightarrow f, \\ &\text{(Parameter Prediction) } Q_f : (\mathcal{O}_t, \mathcal{O}_O) \rightarrow \mathbf{\Theta}, \end{aligned} \tag{5}$$

i.e., P proposes the next function f_i, and the corresponding Q_{f_i} is used to determine the parameters for the function f_i. This process iterates until we reach some terminal states or a preset maximum step.

Function Transition. P controls the function transition dynamics. In the literature, to simplify the learning process, most works use a predefined order (e.g., canonical orders as from left to right, top to down (Ellis et al., 2018)) to constrain the execution of functions. However, this strategy is inefficient and problematic because a context-free program should have the flexibility to execute freely as needed. Thus,

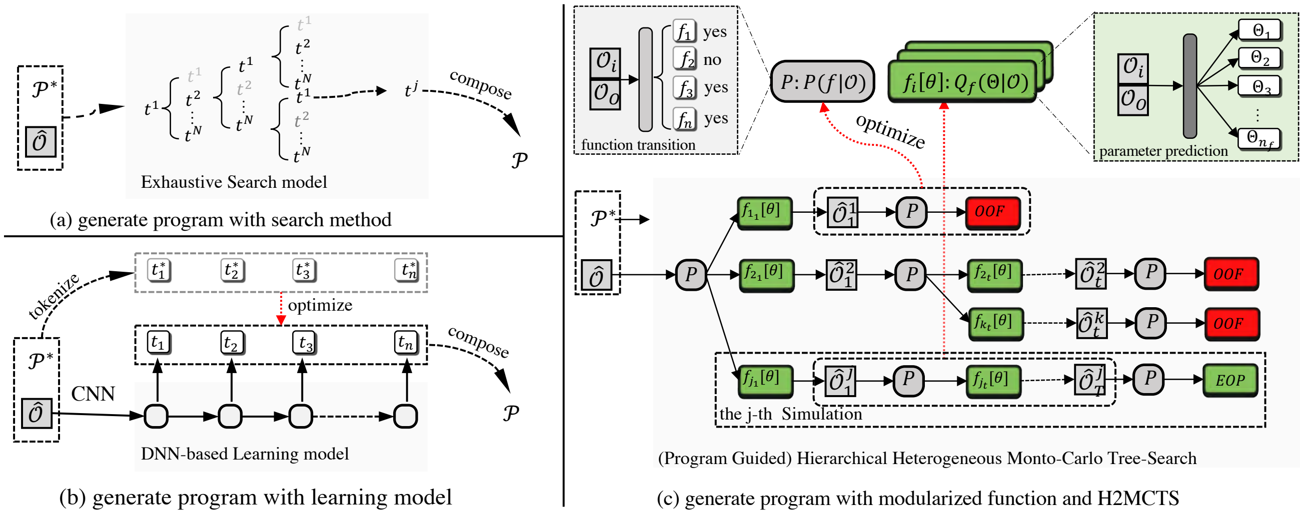


Figure 2. An illustration of (a) the search method; (b) the learning model; and (c) the proposed Function Modularization and H2MCTS. Basically, our proposed method learns the **function transition** model to propose possible new functions, and the **parameter prediction** to generate parameters for the function. Moreover, we use our proposed H2MCTS algorithm to effectively generate supervisions. Failed search processes (end with red `OOF`, "out of function") as well as successful search processes (end with green `EOP`, "end of program") can help to train the model. (Best view in color.)

our function transition model aims to capture the probability that f is a suitable function for the i-th step as follows:

$$P(f|\hat{\mathcal{O}}_i) \doteq \Pr\left[\exists \boldsymbol{\Theta} : d(f^{\boldsymbol{\Theta}}(\mathcal{O}_i), \mathcal{O}_O) < d(\mathcal{O}_i, \mathcal{O}_O)\right], \tag{6}$$

where $d(\cdot,\cdot)$ is a distance metric (e.g., program distance (Ellis et al., 2018) or image IoU (Tian et al., 2019)). Intuitively, we aim to find f such that applying f could make our observation more similar to the target with some parameters $\boldsymbol{\Theta}$. Besides modeling primitive functions in $\mathbb{F}$, we have two extra functions to determine the terminal states of programs: "End of Program"(`EOP`) and "Out of Function"(`OOF`) where `EOP` means that the program has successfully reached the target $\mathcal{O}_O$, while `OOF` means that it is impossible to generate the target observation $\mathcal{O}_O$ with the current program.

Parameter prediction. After obtaining the function f from Equation 6, we model each function with its parameters $\boldsymbol{\Theta}$ as a self-contained module Q_f. The goal of Q_f is to predict the best $\boldsymbol{\Theta}$ given the context $\hat{\mathcal{O}}_i$:

$$Q_f(\boldsymbol{\Theta}|\hat{\mathcal{O}}_i) \doteq \arg\min_{\boldsymbol{\Theta}} d(f^{\boldsymbol{\Theta}}(\mathcal{O}_i), \mathcal{O}_O). \tag{7}$$

Next, we introduce the instantiation of P and Q_f.

4.2. Instantiation

In our considered setting, the raw observations come from the visual domain. Therefore, we first encode the visual observations with a convolutional-based encoder $\mathcal{E}$, which transforms the visual observations into a hidden state. Then, the function transition and parameter prediction are performed in the hidden state.

To instantiate the transition model, we set P as the following function:

$$P(f|\hat{\mathcal{O}}_i) = \sigma\left(\mathcal{E}(\mathcal{O}_i, \mathcal{O}_O)^\top \cdot \mathbf{e}_f\right), \tag{8}$$

where $\mathbf{e}_f$ is a vector representation of the function f, σ is the normalization function.

To instantiate the parameter prediction model , we set Q_f as a multi-head self-contained deep neural network where each head corresponds to one parameter:

$$Q_{f,j}(\Theta_j|\hat{\mathcal{O}}_i) = \mathrm{MLP}_{f,j}(\mathcal{E}(\mathcal{O}_i, \mathcal{O}_O)), j = 1, 2\cdots, n_f. \tag{9}$$

$\mathrm{MLP}_{f,j}$ corresponds to the parameter $\Theta_{f,j}$ for function f's j-th parameter, with an appropriate activation function, *e.g.*, sigmoid for numerical parameters and softmax for categorical parameters. Finally, for each $f \in \mathbb{F}$, the parameter prediction function is as follows:

$$Q_f(\boldsymbol{\Theta}|\hat{\mathcal{O}}_i) = \prod_{j=1}^{n_f} Q_{f,j}(\Theta_j|\hat{\mathcal{O}}_i). \tag{10}$$

5. Learning and Inference with H2MCTS

In this section, we propose the Hierarchical-Heterogeneous Monto-Carlo tree search (H2MCTS) algorithm in conjunction with the modularized function. Compared to using the naive synthetic data introduced in Section 3.2, our proposed H2MCTS can provide high-quality uncorrelated training data during training, and serves as an efficient search technique during inference.

5.1. H2MCTS

Following the standard Monto-Carlo tree search, we start from the initial observation $\mathcal{O}_I$ and aim to find the target observation $\mathcal{O}_O$ by hierarchically running the search algorithm on two types of tree node: `function` node for function transition and `parameter` node for parameter prediction. The probability of visiting node v depends on the following score function:

$$score(v) = \frac{1}{\sqrt{1 + \beta \cdot \alpha(v)}} p(v), \tag{11}$$

where $\alpha(v)$ is the visit count of of node v, β is a scaling hyper-parameter, and $p(\cdot)$ is $P(\cdot)$ for `function` node and $Q_f(\cdot)$ for `parameter` node. In essence, the learnable distribution encourages searching most potential candidates, while penalizing frequently visiting the same nodes within a tree and thus helps exploration. Concretely, in the l-th round of search, starting from $\hat{\mathcal{O}}_0 = (\mathcal{O}_I, \mathcal{O}_O)$, H2MCTS includes the following steps:

- **Select function**. Given the current observation $\hat{\mathcal{O}}_{l_i}$, we choose the next function f_{l_i} based on the score of each `function` node.
- **Select parameter**. After selecting the function f_{l_i}, we choose $\mathbf{\Theta}_{l_i}$ from the corresponding `parameter` nodes.
- **Expand Node**. We obtain $\mathcal{O}_{l_{i+1}} = f_{l_i}^{\mathbf{\Theta}_{l_i}}(\mathcal{O}_{l_i})$, and add $\hat{\mathcal{O}}_{l_{i+1}} = (\mathcal{O}_{l_{i+1}}, \mathcal{O}_O)$ into the search tree.
- **BackUp**. Repeat the above steps until we reach the terminal state `EOP`, `OOF` or the maximum step. If we reach `EOP` then return the observation and the program from the current iteration as the output. If we reach `OOF` or the maximum step, which indicates that this search process fails, we update the visit count of each node and start the next search process.

More details of H2MCTS are provided in Appendix 1.

5.2. Learning with H2MCTS

In this subsection, we introduce how to transform the synthetic data as introduced in Section 3.2 using H2MCTS and adopt the transformed data to more effectively train the model. We mainly need to train the transition model P and the parameter prediction function $\{Q_f\}_{f \in \mathbb{F}}$.

Specifically, consider a randomly generated training data $(\mathcal{O}_I, \mathcal{O}_O, \mathcal{P})$ so that $\mathcal{P}(\mathcal{O}_I) = \mathcal{O}_O$. We show how $\mathcal{P}$ could be used to guide H2MCTS. In the i-th search step, we define the **program-guided** probability P^* and Q_f^* with $\mathcal{P}_0 = \mathcal{P}$:

$$P^*(f|\hat{\mathcal{O}}_i) = \mathbf{1}\left(f \in \mathcal{P}_i\right),$$
$$Q_f^*(\mathbf{\Theta}|\hat{\mathcal{O}}_i) = \mathbf{1}(\mathbf{\Theta} = \mathtt{Top}(\mathcal{P}_i, f)),$$

where $\mathbf{1}(\cdot)$ is the indicator function and $\mathtt{Top}(\cdot)$ finds the first parameter $\mathbf{\Theta}$ of the function f in $\mathcal{P}_i$. After selecting a function f and parameter $\mathbf{\Theta}$ in the i-th step, we update the program as $\mathcal{P}_{i+1} = \mathcal{P}_i \backslash \{f^{\mathbf{\Theta}}\}$. We emit the `EOP` token when $\mathcal{P}_i$ is empty. Using P^* and Q_f^* in the H2MCTS algorithm, we actually obtain an alias of program $\mathcal{P}$ which could transform $\mathcal{O}_I$ into $\mathcal{O}_O$ as $\mathcal{P}$ does. We then form the training data by collecting the pair along the $(\mathcal{O}_{l_i}, \mathcal{O}_O, f_{l_i}, \mathbf{\Theta}_{l_i}), \forall 1 \leq i \leq T$. The same procedure can be performed for other search traces and the overall training set is $\mathcal{T} = \bigcup_{l,\mathcal{P}} \{(\mathcal{O}_{l_i}, \mathcal{O}_O, f_{l_i}, \mathbf{\Theta}_{l_i})\}_{i=1}^T$.

Finally, we optimize the following objective functions

$$\begin{aligned} &\max_P \mathop{\mathbb{E}}_{(\mathcal{O}_i, \mathcal{O}_O, f_i, \mathbf{\Theta}_i) \in \mathcal{T}} \left[P(f_i|\mathcal{O}_i, \mathcal{O}_O)\right] \\ &\max_{Q_f} \mathop{\mathbb{E}}_{(\mathcal{O}_i, \mathcal{O}_O, f_i, \mathbf{\Theta}_i) \in \mathcal{T}} \left[Q_f(\mathbf{\Theta}_i|\mathcal{O}_i, \mathcal{O}_O)\right], f \in \mathbb{F} \end{aligned} \tag{12}$$

Notice that in the above learning process, instead of only using functions with the generated order, we enumerate and use all possible functions variants using H2MCTS as long as we can approach the target observation, which greatly enriches our training examples.

5.3. Inference with H2MCTS

With learned $P(f|\hat{\mathcal{O}})$ and $\{Q_f(\mathbf{\Theta}|\hat{\mathcal{O}})|f \in \mathbb{F}\}$ (Sec. 5.2), the H2MCTS algorithm could be used as a search technique directly by using the score function to calculate visiting probabilities of nodes. Therefore, we also adopt H2MCTS during inference. In experiments, we show that H2MCTS empirically outperforms other search methods in both search efficiency and accuracy.

6. Experiments

In this section, we demonstrate the superiority of our method through three experiments: firstly, we compare our modularized function with the token-based models on a small hand-crafted dataset, which is designed as small as possible such that token-based methods would still work; secondly, we compare our method on the LaTeX 2D drawing dataset (Ellis et al., 2018), which only includes control-free commands; lastly, we examine our model on the complex 3D shape-synthesis dataset (Tian et al., 2019), which is more difficult with control flows and more primitive functions.

6.1. 5×5 Pixel Grid: Token vs. Modular

Dataset. In the first experiment, we aim to compare our modularized function and the token-based model. We create a small dataset on the 5×5 pixel grid with ten colors (one background and nine foregrounds) with relatively simple parametric functions such that those token-based methods still work. We consider five primitive function `DOT`, `VLINE`, `HLINE`, `BLOCK`, and `BORDER`, corresponding to drawing a

Table 1. Primitive functions for the Pixel Grid dataset

Primitive Functions	**Descriptions**	**Overall Action Space**
DOT[X,Y,COL]	draw a **dot** at position **(X, Y)** with color **COL**	$C(5,1)\times C(5,1)\times 9 = 5\times 5\times 9 = 225$
VLINE[TX,BX,Y,COL]	draw a **vertical line** from **(TX, Y)** to **(BX, Y)** with color **COL**	$C(5,1)\times C(5,2)\times 9 = 5\times 10\times 9 = 450$
HLINE[LY,RY,X,COL]	draw a **horizontal line** from **(X, LY)** to **(X, RY)** with color **COL**	$C(5,2)\times C(5,1)\times 9 = 10\times 5\times 9 = 450$
BLOCK[TX, LY, BX, RY, COL]	draw a **block** from **(TX, LY)** to **(BX, RY)** with color **COL**	$C(5,2)\times C(5,2)\times 9 = 10\times 10\times 9 = 900$
BORDER[TX, LY, BX, RY, COL]	draw a **rectangle border** from **(TX, LY)** to **(BX, RY)** with color **COL**	$(C(5,2)-4)^2\times 9 = 6\times 6\times 9 = 324$

single pixel dot, a vertical line, a horizontal line, a rectangle block, and a rectangle border, respectively. For example, the definition of the primitive function `DOT` is:

> `DOT(X, Y, COL)`: draw a `DOT` at position `(X,Y)` with color `COL`. On the 5×5 grid, this function has $5 \times 5 \times 9 = 225$ function variants.

We summarize all the primitive functions of this dataset in Table 1. Considering all the function variants, there are 2349 actions in total, which is much larger than the common program learning tasks used tasks (Pattis et al., 1981; Balog et al., 2016).

An example is shown in the left of Figure 4. Since this dataset does not contain control-flow, we only need to learn a sequence of parametric primitive functions to transform $\mathcal{O}_I$ to $\mathcal{O}_O$.

Baselines. We compare our proposed function modularization with two token-based baselines:

- Word-as-Token (Balog et al., 2016): The method treats each word as a token, e.g., `DOT(X, Y, COL)` in transformed into six tokens: [‘`DOT`’, ‘`(`’, ‘`X`’, ‘`Y`’, ‘`COL`’, ‘`)`’]. This is similar to the sentence tokenization preprocessing in the NLP community.
- Function-as-Token: The method treats each function-parameter variant as a token, e.g., ‘`DOT(1, 2, RED)`’ and ‘`DOT(2, 2, RED)`’ are considered as two tokens. This is similar to Reinforcement Learning (RL) which learns when to conduct which symbolic action.

We conceptually compare different methods in Table. 2. Word-as-Token results in a relatively small number of tokens, but these tokens do not contain syntax information. Function-as-Token contains syntax by itself, but results in a large number of tokens. As for our proposed modularized function, the total number of tokens equals the number of primitive functions, while ensuring the syntax.

We compare different methods in both the unsupervised and the supervised settings. For the unsupervised setting, we train the Word-as-Token model as in Balog et al. (2016), which is a state-of-the-art model on program induction. The

Table 2. A comparison of different methods on the token setting.

TOKENIZING METHOD	# TOKENS	SYNTAX
WORD-AS-TOKEN	29	NO
FUNCTION-AS-TOKEN	2,349	YES
MODULARIZED(OUR)	7	YES

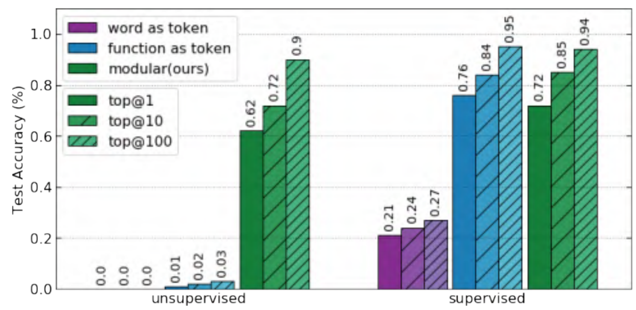


Figure 3. The testing accuracy (%) on the 5×5 Pixel Grid dataset.

Function-as-Token model is trained with Reinforcement Learning (Sutton et al., 2000). For the supervised settings, following (Ellis et al., 2018), we provide ground-truth programs to all the models.

Results. As shown in Figure 3, for the unsupervised setting, our proposed method reports impressive results by reporting accuracy of 0.9 for the top100 metric, while Word-as-Token (Balog et al., 2016) and Function-as-Token both fail due to the huge search space. The results show that our proposed function modularization can effectively reduce the number of tokens and relieve the burden of supervision signals in the unsupervised setting.

For the supervised setting, Word-as-Token also fails to capture the complex parameters and syntax and performs poorly, while our method and Function-as-Token show satisfactory performance. In particular, both methods could find 95% of the valid program within 100 searches. The results show that Function-as-Token can work reasonably well if enough supervised data is provided, which, however, is expensive or infeasible in practice. In contrast, our proposed function modularization can work with both supervised and unsupervised settings.

Finally, we provide a showcase of different models in Fig. 4. The failure of Word-as-Token is mainly due to the syntax

Table 3. The results on the 2D LaTeX Drawing dataset. All the models are trained on the synthesized dataset. The testing dataset includes both synthesized data and another 100 real hand-drawn images.

Model	MLP Output Dimension	Synthesized		Real Hand-Drawn		
		Train	Test	Top@1	Top@10	Top@100
Seq2Seq(F) + CE Loss	$\sum_f \prod_j^{n_f} \lvert\mathbf{\Theta}_j\rvert > 30{,}000$	66.94%	6.30%	0%	0%	0%
Seq2Seq(F) + RL Loss		44.20%	2.10%	0%	0%	0%
Seq2Seq(F) + CE-RL Loss		76.44%	15.41%	0%	0%	0%
STN + SMC(Ellis et al., 2018)	$\sum_f n_f + \lvert\mathbb{F}\rvert + 1 = 16$	—	—	63%	70%	70%
STN + SMC(our implement)		90.74%	71.21%	64%	72%	74%
FM + Canonical	$\sum_f \mathbf{O}(n_f) + \lvert\mathbb{F}\rvert + 2 = 20$	91.81%	74.45%	67%	72%	72%
FM + H2MCTS		93.91%	83.13%	76%	84%	86%

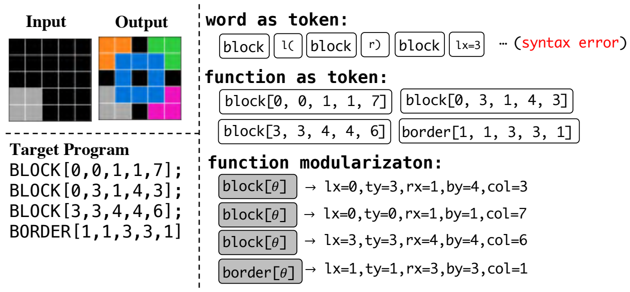
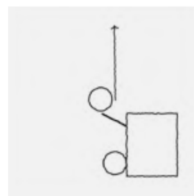
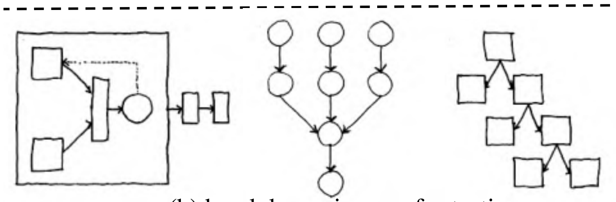


Figure 4. An illustration of the 5×5 Pixel Grid dataset. We show the input-output observation and the target program in the left. In the right, we show the generated programs of three methods under the supervised setting.



```
circle(x=8,y=8,radius=1),
circle(x=9,y=14, radius=1),
line(x1=9,y1=8,x2=9,y2=1,
     arrow=true,solid=true),
line(x1=8,y1=9,x2=10,y2=10,
     arrow=false,solid=true),
rectangle(x1=10,y1=9,x2=15,y2=15)
```

(a) synthesized observation and its ground truth program.



(b) hand drawn images for testing.

Figure 5. Top: an example of the synthesized image and the program that generated it. Bottom: examples of the true hand-drawn images which correspond to model diagram, flowing chart, and tree structures. The learned programs should create legible figures by rendering in LaTeX.

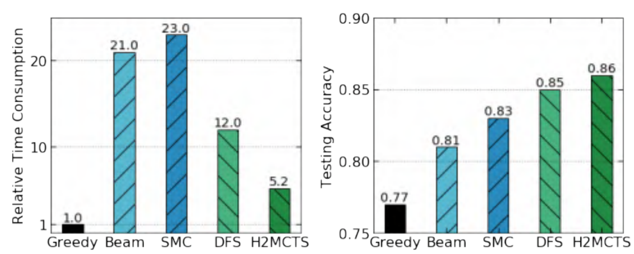


Figure 6. A comparison between different search methods. Left: the relative time consumption, from which we could find that our H2MCTS algorithm consumes much less time than other methods; Right: the testing accuracy, which demonstrate that our H2MCTS obtain the best performance.

problem. Function-as-Token could correctly predict the program, with the cost of having to select from the 2,349 candidates tokens (Table 2). In contrast, our proposed method can easily generate the program by function modularization.

6.2. 2D LaTeX Drawing: Control-free Program Learning

Dataset. In the second experiment, we adopt the LaTeX 2D drawing dataset (Ellis et al., 2018). The goal is to learn LaTeX executable programs with visual observations. Following (Ellis et al., 2018), the training set includes 95,000 synthesized data, and the testing set includes 5,000 synthesized data as well as 100 real hand-drawn images.

Specifically, $\mathcal{O}_I$ is an empty canvas, and $\mathcal{O}_O$ is an image with size 256×256. This dataset contains 3 primitive functions: `Circle`, `Line`, and `Rectangle`, each of which draws on a discrete 16×16 grid coordinates. The synthesized data contains randomly generated programs, while the real hand-drawn images aim to show certain structures. Figure 5 shows some examples of the dataset.

Baselines. We compare the following methods. (1) A LSTM-based Seq2Seq language model, which has achieved successes in language translation and image captioning tasks (Vinyals et al., 2015). Based on the results from 5×5 Pixel Grid in Section 6.1, we only consider the Function-as-Token tokenizing method and denote it as Seq2seq(F). We compare three versions where the first two use Cross-Entropy (CE) loss, Reinforcement Learning (RL) loss during training respectively, and the third one is pretrained

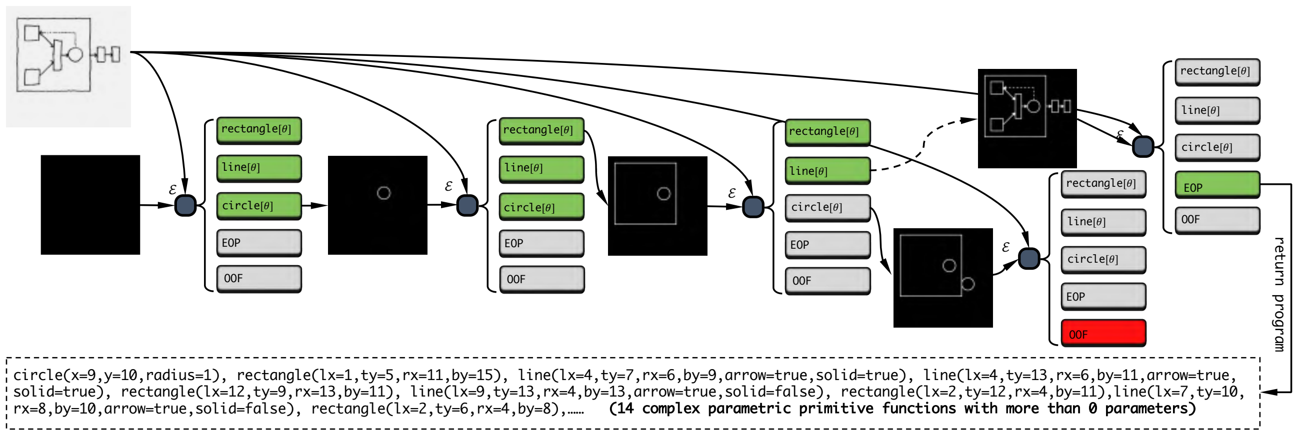


Figure 7. An example of the 2D LATEXDrawing dataset. In the illustration, the function transition model proposes green function modules as executable ones, and the H2MCTS algorithm selects and executes one function. This process is repeated until we reach the `OOF` terminal state, marking the failure of this search process, or the `EOP` terminal state, which marks the success of this search. Then, the programs along with their parameters are returned as the final program. The color of the rendered images is inverted for better visualization.

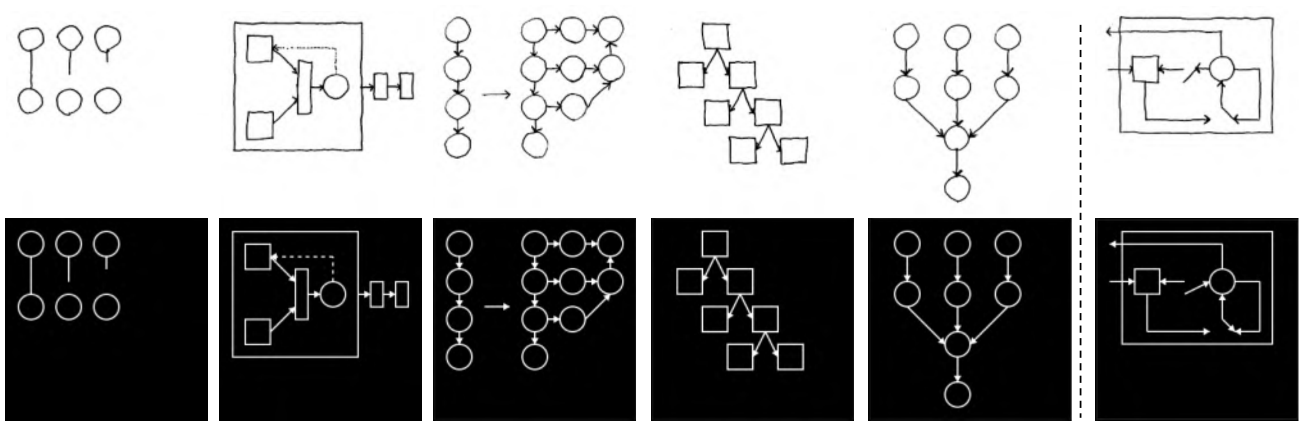


Figure 8. More showcases for the LATEX 2D drawing datasets. Top: the hand drawings; Bottom: the LATEXrendered output with our generated program.

with the CE loss and further fine-tuned with the RL loss. (2) Spatial Transformer Network model with Sequential Monto-Carlo search (STN+SMC)(Ellis et al., 2018), which achieves the state-of-the-art result. We present two versions: the original results reported in the paper and our own implementation. (3) Our proposed function modularization and H2MCTS model (FM+H2MCTS). We also include an ablation study, which uses the standard canonical function order to replace the H2MCTS training, denoted as FM+Canonical. For all methods except the original result from (Ellis et al., 2018), we use ResNet-18 as the encoder $\mathcal{E}$ to ensure a fair comparison.

Results. The results are shown in Table 3. We make the following observations. Overall, our proposed FM+H2MCTS model reports the best results, consistently and greatly outperforming the most competitive baseline by more than 10% in both the synthesized test set and real hand-drawn images. The results demonstrate the effectiveness of our proposed method in handling the 2D LATEXdataset. We attribute the reasons into two folds. First, our proposed multi-head self-contained neural module is more flexible to handle different parameters adaptively. For example, we could model `coordinates` with a regression head and model `arrow state` with a binary classification head. On the other hand, compared to using the canonical approach, our H2MCTS algorithm also contributes to the performance by getting rid of the predefined function execution order during both training and inference which is extremely inflexible.

The Seq2Seq model shows poor results even in the training dataset, not to mention handling real hand-drawn images. A plausible reason is that this baseline is difficult to converge due to the huge number of parameters in MLP (more than 30,000 dimension outputs). The results are consistent with Ellis et al. (2018) that pure DNN-based approaches could not tackle the complicated visual program induction tasks.

We also compare H2MCTS and SMC with other search

Table 4. The results on the 3D Shape dataset. Following (Tian et al., 2019), we compute the intersection over union (IoU) between the target observation and the rendered output of the generated program as the evaluation metric.

Model	Training Set (IoU ↑)		Testing Set (IoU ↑)			
	Table	Chair	Bed	Sofa	Cabinet	Bench
Tian et al. (2019)	0.492	0.469	0.283	0.365	0.345	0.248
FM + H2MCTS (ours)	0.641	0.642	0.501	0.670	0.661	0.602

strategies including greedy search, beam search, and depth-first search (DFS) regarding their search accuracy and efficiency during inference time based on our trained model. The results are shown in Figure 6. We observe that H2MCTS outperforms other search methods with respect to both search time and testing accuracy. Among baselines, DFS is most competitive based on our well-trained function transition model and parameter prediction model.In comparison, beam search and SMC perform slightly worse. The differences mainly come from the searching strategy: Beam Search and SMC are optimized Breadth-First-Search that keep a number of partials (bandwidth) at every iteration which consumes much more computation; while H2MCTS is an optimized Depth-First-Search method that stores the search statics at each iteration for next round. With a well-trained model, a small bandwidth could be sufficient to tackle most of the problem and thus leads to the high efficiency of H2MCTS and DFS.

The results demonstrate that H2MCTS serves as an efficient inference technique, as discussed in Section 5.3.

Finally, to provide a more intuitive understanding, we present a showcase of our method in Figure 7 and Figure 8, including the raw observation, the learned programs, and the LaTeX rendered images. The figure clearly shows the workflow and the effectiveness of our proposed method in learning parametric programs from real visual scenes. We include more examples in Figure 8, where the first five drawings could be solved perfectly.

6.3. 3D Shape: Control-based Program Learning

Datasets. In the last experiment, we adopt the 3D-Shape dataset (Tian et al., 2019) containing 18 primitive functions and `for loops`, *i.e.*, control-based programs. A showcase of the dataset is provided in Figure 9.

Settings and Baselines. To enable our proposed method to handle the controls, we add one extra type of **Node** to determine the control flow, and extend the bi-level modeling in Eq. (5)) into a tri-level modeling as follows:

$$\begin{aligned}\text{(Control Transition) } C \ &: (\mathcal{O}_i, \mathcal{O}_O) \rightarrow C_i. \\ \text{(Function Transition) } P \ &: (\mathcal{O}_i, \mathcal{O}_O) \rightarrow f_i, \\ \text{(Parameter Prediction) } Q_f &: (\mathcal{O}_i, \mathcal{O}_O, C_i) \rightarrow \mathbf{\Theta}.\end{aligned} \tag{13}$$

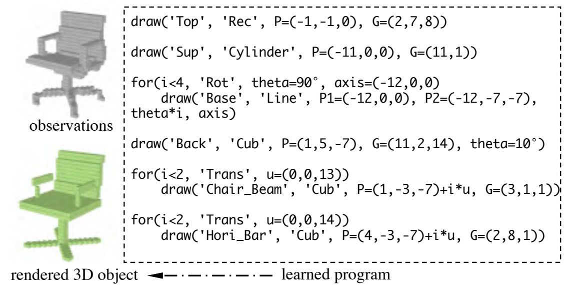


Figure 9. An illustration of the 3D Shape dataset.

With this tri-level modeling, functions within different control blocks are still context-free and therefore our proposed H2MCST algorithm still applies. See Appendix B for more details. We mainly compare our proposed method with Tian et al. (2019), a state-of-the-art baseline for this dataset. Notice that guided adaptation used in (Tian et al., 2019) is not available in our considered setting.

Results As shown in Table 4, the program generated by our method achieves better performance for all categories of objects, demonstrating the general applicability of our method on control-based programs. In Figure 9, we provide a showcase of our method, which successfully generates a program to describe the visual observation.

7. Conclusion

In this paper, we investigate the parametric visual program induction task by decoupling the learning of parametric function as learning function transition and function parameter prediction. We propose the concept of function modularization and the H2MCTS algorithm. Our method outperforms state-of-the-art baselines with higher accuracy and efficiency. Future works include exploring more visual program induction scenarios using our proposed method.

Acknowledgement

This work is supported by the National Key Research and Development Program of China No. 2020AAA0106300 and National Natural Science Foundation of China No. 62102222.

References

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2016.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2018.

Chollet, F. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Devlin, J., Bunel, R., Singh, R., Hausknecht, M., and Kohli, P. Neural program meta-induction. *Neural Information Processing Systems (NIPS)*, 2017.

Dong, H., Mao, J., Lin, T., Wang, C., Li, L., and Zhou, D. Neural logic machines. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2019.

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. B. Learning to infer graphics programs from hand-drawn images. *Neural Information Processing Systems (NIPS)*, 2018.

Ellis, K., Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

Eslami, S., Heess, N., Weber, T., Tassa, Y., Szepesvari, D., Hinton, G. E., et al. Attend, infer, repeat: Fast scene understanding with generative models. *Advances in Neural Information Processing Systems (NIPS)*, 29:3225–3233, 2016.

Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

Gulwani, S. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46 (1):317–330, 2011.

Irving, G., Szegedy, C., Alemi, A. A., Eén, N., Chollet, F., and Urban, J. Deepmath-deep sequence models for premise selection. *Advances in Neural Information Processing Systems*, 29:2235–2243, 2016.

Jaderberg, M., Simonyan, K., Zisserman, A., et al. Spatial transformer networks. *Advances in neural information processing systems (NIPS)*, 28:2017–2025, 2015.

Lezama, A. S. *Program synthesis by sketching*. PhD thesis, PhD thesis, EECS Department, University of California, Berkeley, 2008.

Ling, W., Yogatama, D., Dyer, C., and Blunsom, P. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pp. 158–167, 2017.

Manna, Z. and Waldinger, R. Knowledge and reasoning in program synthesis. *Artificial intelligence*, 6(2):175–208, 1975.

Manna, Z. and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. Learning to infer program sketches. In *International Conference on Machine Learning*, pp. 4861–4870. PMLR, 2019.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2017.

Pattis, R., Roberts, J., and Stehlik, M. Karel the robot. *A gentele introduction to the Art of Programming*, 1981.

Piantadosi, S. T. *Learning and the language of thought*. PhD thesis, Massachusetts Institute of Technology, 2011.

Shin, R., Kant, N., Gupta, K., Bender, C., Trabucco, B., Singh, R., and Song, D. Synthetic datasets for neural program synthesis. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2019.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Sun, S.-H., Noh, H., Somasundaram, S., and Lim, J. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pp. 4790–4799. PMLR, 2018.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems (NIPS)*, pp. 1057–1063, 2000.

Tian, Y., Luo, A., Sun, X., Ellis, K., Freeman, W. T., Tenenbaum, J. B., and Wu, J. Learning to infer and execute 3d shape programs. In *International Conference on Learning Representations*, 2019.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3156–3164, 2015.

Waldinger, R. J. and Lee, R. C. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pp. 241–252, 1969.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. B. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Neural Information Processing Systems (NIPS)*, 2018.

Young, H., Bastani, O., and Naik, M. Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning*, pp. 7144–7153. PMLR, 2019.

Zhou, C., Li, C.-L., and Póczos, B. Unsupervised program synthesis for images by sampling without replacement. In *Uncertainty in Artificial Intelligence*, pp. 408–418. PMLR, 2021.

A. Implementation and Training

In this section, we briefly introduce more details about the modularized functions, as well as our training framework.

A.1. Function Modular

Parametric primitive functions are easy to implement, but how to determine their parameters is hard. This is one of the motivation of this paper. To achieve this, we organize each primitive function and its parameter into a module with a multi-head MLP (each head is a two-layer MLP as `ParamNet`), leaving each head corresponding to one parameter. An examples of the `DOT` primitive function and its modularized form `Dot` are shown as follows:

```
import numpy as np
import torch

def DOT(In, X, Y, COL):
    Out = copy.deepcopy(In)
    Out[X, Y] = COL
    return Out

class Dot(torch.nn.Module):
    def __init__(self, H):
        super().__init__()
        self.fn = DOT
        # ParamNet is another mlp to predict parameters.
        self.params['X'] = ParamNet("X", "REG", range=MAXSIZE[0])
        self.params['Y'] = ParamNet("Y", "REG", range=MAXSIZE[0])
        self.params['C'] = ParamNet("C", "CLS", range=NUM_OF_COLOR)

    def inference(In, Target):
      s = encoder(Target) - encoder(In)
      param_x = self.params['X'](s)
      param_y = self.params['Y'](s)
      param_c = self.params['C'](s)
      return self.fn(In, param_x, param_y, param_c)
```

Listing 1. An example of Function Modular.

A.2. Overall Network Architecture

With the idea of function modularization, the whole deep model is clear to go as a shared convolutional encoder, followed with **function transition** head to predict the transition dynamics for next function, while several function modular to predict the parameter for each function. As for the encoder, we adopt a 4-layer convolutional networks for 5×5 Pixel Grid, use ResNet18 for 2D LaTeX Drawing, use a shapenet as Tian et al. (2019) for 3D Shape.

Moreover, to accelerate the training speed, we follow the setting of MuZero (Silver et al., 2016) to implement a distributed system based on Ray[1]. The system consists of 60 explorers to continuously explore the function space via the H2MCTS algorithm (Alg.1), and explored traces are used to train the model via an online trainer. The whole framework is launched on a GPU server with two `Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz` CPU processors and two `Nvidia GeForce RTX 3090` GPU processors.

A.3. The H2MCTS algorithm

Our H2MCTS algorithm follows and generalizes the basic MCTS algorithm and (Silver et al., 2016). A basic MCTS algorithm includes four parts as: **Selection**, **Expansion**, **Simulation**, **BackUp**. Both Silver et al. (2016) and our work use the neural network to conduct the **Simulation** step. Moreover, we consider two different types of nodes in the simulation process, and thus our algorithm is called H2MCTS. The detailed algorithm is shown in Algorithm 1.

[1]https://www.ray.io/

Algorithm 1 H2MCTS

Input: The Function Transition model $P(\cdot|\cdot)$; A set of Parameter Prediction models $\{Q_f(\cdot|\cdot)\}$; {Eq. (5)}
Input: A raw observation input-output pair $\mathcal{O}_I, \mathcal{O}_O$.
init: $j \leftarrow 0$.
init: `ROOT` $\leftarrow (\mathcal{O}_I, \mathcal{O}_I, 0)$ {`ROOT` node with $\mathcal{O}_I$, $\mathcal{O}_O$, and visit count 0}
repeat
  `PNode` $\leftarrow$ `ROOT` {We starts from root at every round}
  repeat
    $f = \arg\max_f 1/(1 + \beta * \alpha(\texttt{PNode})) * P(f|\texttt{PNode.context})$ {**Select Function** node according to Eq. (11)}
    if f is `OOF` **then**
      `node` $\leftarrow$ `PNode` {**BackUp** node visit count}
      repeat
        α`(node)` $\leftarrow \alpha$(node) + 1
        `node` $\leftarrow$ `node.parent`
      until `node` is None
      break inner loop
    end if
    if f is `EOP` **then**
      break outer loop
    end if
    if $f \notin$ `PNode.children` **then**
      `FNode` $\leftarrow$ (f, 0) {create new function node with f, and visit count 0}
      `PNode.children.add( FNode(f, 0))` {Add f to `PNode`'s children}
    end if
    $\mathbf{\Theta} = \arg\max_{\mathbf{\Theta}} 1/(1 + \beta * \alpha(\texttt{PNode})) * Q(\mathbf{\Theta}|\texttt{PNode.context})$ {**Select Parameter Node** for f according to Eq. (11)}
    $\mathcal{O}_{new} \leftarrow f^{\mathbf{\Theta}}(\texttt{PNode.I})$ {Obtain new observations}
    if $\mathbf{\Theta} \notin$ `FNode.children` **then**
      $\mathcal{O}_{new} \leftarrow f^{\mathbf{\Theta}}(\texttt{PNode.Input})$ {render the new observation and **Expand** the search tree}
      `PNode` $\leftarrow (\mathcal{O}_{new},$ `PNode.Output`$, 0)$ {A new `PNode`}
      `FNode.children.add(PNode)`
    end if
  until K exceed maximum search depth
until j exceed maximum number of simulation
$\mathcal{P} \leftarrow$ `[]`
repeat
  $\mathcal{P}$`.add(PNode.parent.`f`, PNode.`$\mathbf{\Theta}$`)`
  `PNode` $\leftarrow$ `PNode.parent.parent`
until `PNode` is None
Return $\mathcal{P}$

B. Extension to Context-based Scenarios

To extend our formulation to context-based scenarios, we firstly rewrite a program as $\mathcal{P} = (C_1, C_2, .., C_{N_P})$, where $C_i = (C^{\mathbf{\Theta}_C}, f_{i,1}^{\mathbf{\Theta}_{i,1}}, f_{i,2}^{\mathbf{\Theta}_{i,2}}, \cdots)$ is the i-th logic collection block and $C^{\mathbf{\Theta}_C}$ is the control unit (e.g., `loop`, `if-else`), and $f_{i,j}^{\mathbf{\Theta}_{i,j}}$ is primitive function f with parameters $\mathbf{\Theta}_{i,j} = (\Theta_{i,j,0}, \Theta_{i,j,}, \cdot, \Theta_{i,j,n_f})$, n_f is the number of parameters for f (e.g., `line`$^{(lx,ty,rx,by)}$). Then we define a tri-level modeling as:

$$\text{Pr}\left[\mathcal{P}|\mathcal{O}_I, \mathcal{O}_O\right] = \prod_{i=1}^{N_p} C(c_i|\hat{\mathcal{O}}_{i-1}) \cdot P(f_i|\hat{\mathcal{O}}_{i-1}) \cdot Q_{f_i}(\mathbf{\Theta}_i|\hat{\mathcal{O}}_{i-1}, c_i). \tag{14}$$

This means we firstly determine the control unit, then determine the function, and finally the parameter. Moreover, we add an extra control unit as `NUL` which directly executes its enclosed commands, and the H2MCTS algorithm is correspondingly extended with the extra Control node.

C. More Experiments Results

C.1. Pixel Grid

C.1.1. DATASET

To generate the dataset, we randomly sample a sequence of functions and parameters from the primitive functions $\mathcal{P} = \left[f_1^{(\mathbf{\Theta}_1)}, f_2^{(\mathbf{\Theta}_2)}, \cdots f_T^{(\mathbf{\Theta}_T)}\right]$ with $T < T_{max}$, and generate a random input $\mathcal{O}_I$, and apply the program to $\mathcal{O}_I$ to obtain $\mathcal{O}_O$. Then $(\mathcal{O}_I, \mathcal{O}_O)$ is used as an input-output pair. Moreover, as some functions could overwrite previous functions (e.g., f_j=`DOT(X,Y,C2)` will overwrite f_i =`DOT(X,Y,C1)` if $j > i$), we carefully compare the intermediate results $[\mathcal{O}_0, \mathcal{O}_1, \cdots, \mathcal{O}_T]$ to remove functions which have been overwritten.

C.1.2. COMPARISON WITH RULE-BASED METHODS.

In our experiments, the training set contains of 10,000 input-output pairs, which consists of 20% programs with length 1, 20% programs with length 2, and 60% programs with length 3. The testing set contains 1,000 input-output pairs with the same length distribution.

C.2. LaTeX 2D drawings

We show the training curve of our model in Figure 10. From the figure, we can see that most of the functions and their parameters could reach an accuracy of more than 95% after 50k iterations.

C.3. 3D shape-synthesis

We provide more showcases for this dataset in Figure 11.

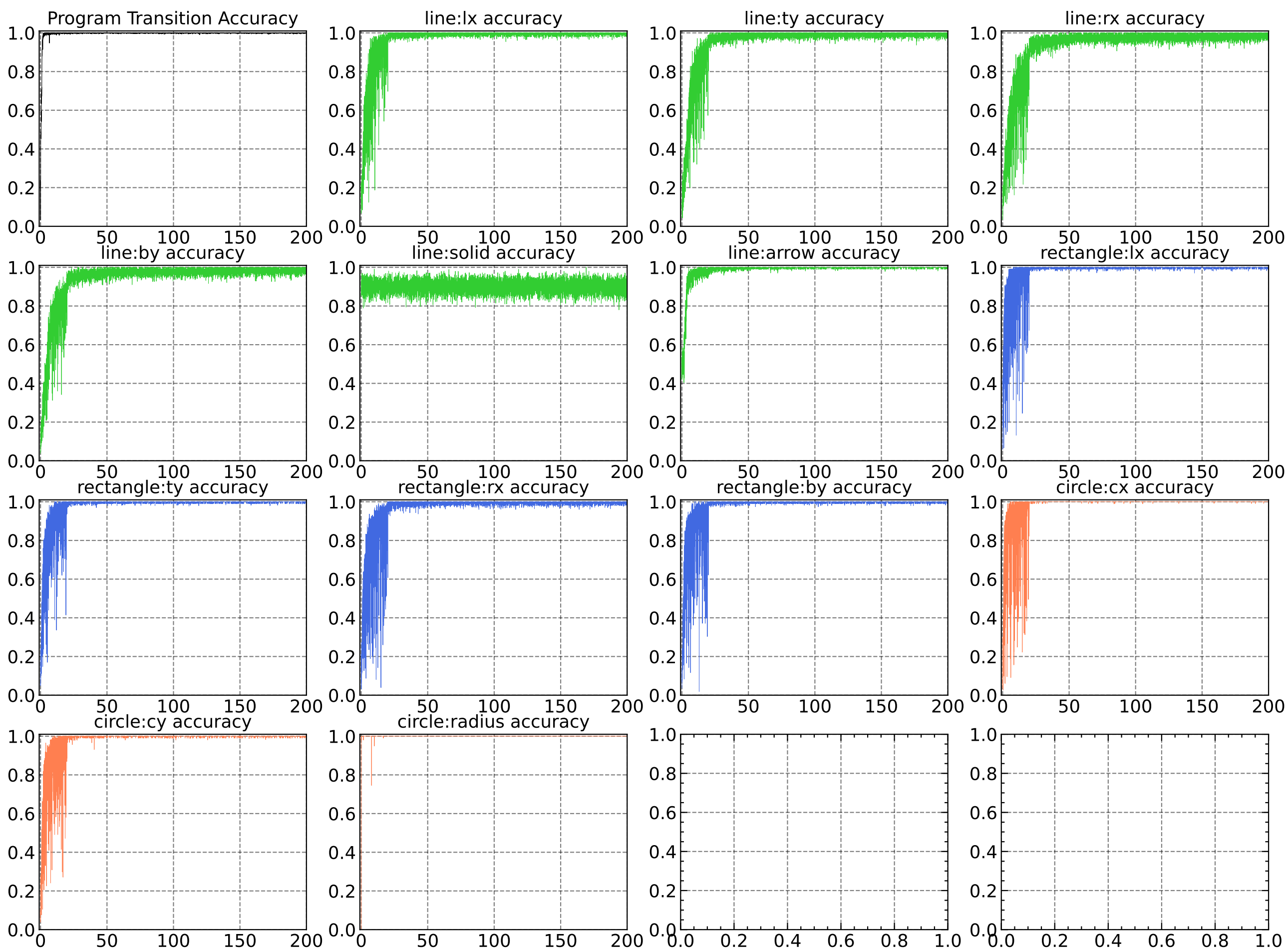


Figure 10. Training Curves for the LaTeX 2D drawing datasets. We can observe that most of the modules could converge within 50k iterations.

```
draw('Top', 'Rec', P=(-1,0,0), G=(3,9,9))
for(i<2, 'Trans', u1=(0,0,17))
    for(i<2, 'Trans', u2=(0,13,0))
        draw('Leg', 'Cub', P=(-12,-7,-9)+i*u1+j*u2, G=(14,2,1))
for(i<2, 'Trans', u=(0,0,17))
    draw('Hori_Bar', 'Cub', P=(-12,-7,-9)+i*u, G=(2,15,1))
draw('Back', 'Cub', P=(2,5,-9), G=(10,3,18), theta=5°)
for(i<2, 'Trans', u=(0,0,18))
    draw('Chair_Beam', 'Cub', P=(2,-7,-10)+i*u, G=(3,1,1))
for(i<2, 'Trans', u=(0,0,18))
    draw('Hori_Bar', 'Cub', P=(5,-7,-10)+i*u, G=(3,14,1))
```

```
draw('Top', 'Square', P=(10,0,0), G=(3,12))

for(i<2, 'Trans', u1=(0,0,16))
    for(i<2, 'Trans', u2=(0,17,0))
        draw('Leg', 'Cub', P=(-12,-10,-9)+i*u1+j*u2,
G=(24,3,2))

draw('Layer', 'Rec', P=(-2,0,0), G=(2,9,9))
```

```
draw('Top', 'Square', P=(-5,0,0), G=(5,10))

draw('Vert_Board', 'Cub', P=(-10,-8,-10), G=(11,1,19))

for(i<5, 'Rot', theta=72°, axis=(-11,1,0)
    draw('Base', 'Line', P1=(-11,1,0), P2=(-12,-8,-6), theta*i, axis)

draw('Back', 'Cub', P=(0,10,-10), G=(11,2,19), theta=0°)
```

Figure 11. More showcases for the 3D shape dataset. Left: the raw observations. Middle: the generated program. Right: the rendered results.