UNSUPERVISED VISUAL PROGRAM INDUCTION WITH FUNCTION MODULARIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

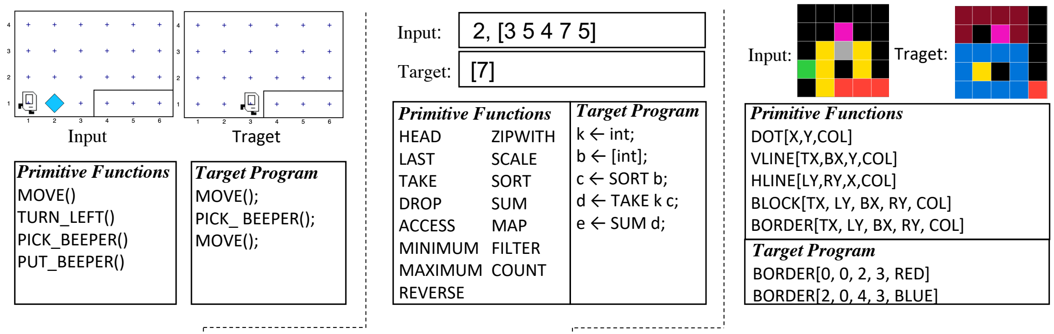
Program induction serves as one way to analog the ability of human thinking. However, existing methods could only tackle the task under simple scenarios (Fig 1(a),(b)). When it comes to complex scenes, e.g., the visual scenes, current program induction methods fail due to the huge program action space. In this paper, to the best of our knowledge, we are the first to tackle this problem. We propose a novel task named *unsupervised visual program induction* in complex visual scenes that require complex primitive functions. Solving this task faces two challenges: i) modeling complex primitive functions for complex visual scenes is very difficult, and ii) employing complex functions in the unsupervised program induction suffers from a huge and heterogeneous program action space. To tackle these challenges, we propose the Self-Exploratory-Modularized-Function (SEMF) model, which can jointly model individual function selection and its parameters through a unified modular block. Moreover, a Monto-Carlo-Tree-Search (MCTS) based Self-Exploratory algorithm is proposed to explore program space with modularized function as prior. The exploratory results, in turn, guide the training of these modularized functions. Our experiments demonstrate that the proposed SEFM model outperforms all the existing baselines in model performance, training efficiency, and model generalization.

1 INTRODUCTION

Program induction is regarded as an efficient way to analog human's thinking process (Fodor, 1975; Piantadosi, 2011), which could be described as: given a set of core primitive functions, the algorithm is required to organize these functions into a program to explain the rule hidden in the observed scenes (Koza & Koza, 1992; Piantadosi, 2011) without teacher guidance. However, when it comes to visual scenes, due to the complexity of the scene and the required complex primitive function, traditional program induction methods fail due to the exponential search complexity.

In this paper, we tackle the task of visual program induction with complex functions unsupervisedly for the first time. We propose a novel task named *unsupervised visual program induction* with *complex functions*. Fig. 1(c) provides an example of visual program induction: a complex visual scene would require complex functions with multiple parameters, thus resulting in a much larger action space. Solving this problem poses two challenges that lead to the failure of existing non-visual program induction approaches: i) modeling complex primitive functions for complex visual scenes is very difficult, and ii) employing complex functions in the unsupervised program induction suffers from a huge and heterogeneous program action space.

To solve these challenges, we propose the Self-Exploratory-Modularized-Function (SEMF) model which is capable of jointly modeling function selection together with its parameters through a unified modular block. Particularly, the whole action space is re-organized modularized functions, i.e., each function together with its parameters will be organized as a unified modular block. A modularized function can act automatically given different observations of visual patterns. Moreover, to train the modularized functions unsupervisedly, we design a function planning mechanism that focuses on the function-level program generation and propose a special two-stage Monto-Carlo-Tree-Search (MCTS) based search algorithm to efficiently conduct visual program induction. The Self-Exploratory algorithm is able to search the program space with those modularized functions as prior, and the explored search results will in turn guide the training of these modularized functions.



(a) ***move*** one step; ***pick*** the blue beeper; and ***move*** one step.

(b) assign inputs to ***k*** and ***b***; sort ***b***, store the result in ***c***; take the ***k***-th element from ***c*** into ***d***; assign the sum of ***d*** to ***e***.

(c) draw a ***rectangle border*** from ***pos (0, 0)*** to ***pos (2, 3)*** with color ***crimson***; draw a ***rectangle border*** from ***pos (2, 0)*** to ***pos (4, 3)*** with color ***blue***.

Figure 1: Program induction task examples. (a) The famous Karel task (Pattis et al., 1981), which includes four primitive functions with no parameter, i.e., the total action space is **4**; (b) The "Sequence Manipulation" task (Balog et al., 2017), which includes 15 primitive functions with limited parameters, and the total action space is **34**. (c) The Visual Program Induction task in this paper, where there are five primitive functions with rich parameters, and the total action space is **2,349**. (See Appendix A to see how these numbers are calculated.)

We conduct extensive experiments to test the proposed algorithm with respect to model performances, training efficiency, and generalization ability compared to several state-of-the-art baselines. The results demonstrate the superiority of our proposed SEMF model over existing methods for solving the task of unsupervised visual program induction with complex functions.

To summarize, we make the following contributions in this paper.

- To the best of our knowledge, we are the first to tackle visual program induction with complex functions in an unsupervised manner, and formally propose the novel problem of *unsupervised visual program induction* with *complex functions*.
- We propose the Self-Exploratory-Modularized-Functions (SEMF) model which is able to jointly optimize individual function selection and its parameters by decoupling the learning process into modularized functions learning and Monto-Carlo-Tree-Search (MCTS) based function planning. The decoupling procedure makes exploring the huge and heterogeneous action space possible and efficient.
- We conduct extensive experiments to compare our proposed SEMF model with several state-of-the-art baselines on a novel and specialized-designed dataset. Our proposed SEMF model could gain magnitude improvement compared with the baselines in terms of model performance, training efficiency, and model generalization ability.

2 Program Induction with Complex Functions

In this section, we formally define the task of program induction with complex functions, and compare it with simple functions studied in the literature.

2.1 Definition

The task of program induction aims to generate programs to explain the input-output pairs from a set of given primitive functions (Koza & Koza, 1992; Piantadosi, 2011). In particular, we focus on complex primitive functions that may have arbitrary and heterogeneous parameters for the first time.

First, we provide some intuitions for modeling complex primitive functions. From the perspective of human cognition, human skills and actions are mostly modularized and complex such that we could act adaptively depending on different environments (Chollet, 2019; Johnson et al., 2021). As the goal of program induction is to analogize and approach human's thinking system (Fodor, 1975;

Piantadosi, 2011), the ability to model complex functions is indispensable. From the perspective of programming language, a parameterized function could represent expediently a group of actions that share the same basic functionality but with different properties to adapt to diverse conditions. However, the modeling of complex functions has long been overlooked in the literature. Therefore, we aim to explicitly model complex functions to meet the demand of LOT and general program induction.

In this paper, a complex function is defined as $F^{\mathbf{\Theta}} = (F, \mathbf{\Theta}_F)$, where $F \in \mathbf{F}$ is the function class, $\mathbf{F}$ denotes all possible function classes, and $\mathbf{\Theta}_F = [\theta_{F,1}, \theta_{F,2}, ..., \theta_{F,n_F}]$ are n_F the function parameters for F. The parameters range is denoted as $\mathbf{\Xi}_F = \Xi_{F,1} \times \Xi_{F,2} ... \times \Xi_{F,n_F}$, i.e., $\theta_{F,i} \in \Xi_{F,i}$. Notice that the parameters and their ranges depend on F. We use $F^{\mathbf{\Theta}}(\mathcal{O}_t) \rightarrow \mathcal{O}_{t+1}$ to represent the mapping of $F^{\mathbf{\Theta}}$ from $\mathcal{O}_t$ to $\mathcal{O}_{t+1}$. We adopt subscripts to denote multiple functions in a sequence, e.g., $F_i^{\mathbf{\Theta}_i}$, and use $\mathcal{F} = \{F^{\mathbf{\Theta}} | F \in \mathbf{F}, \mathbf{\Theta}_F \in \mathbf{\Xi}_F\}$ to denote the possible action space for the complex function.

Formally, given an input $\mathcal{O}_{\text{in}}$ and output $\mathcal{O}_{\text{out}}$, the action space $\mathcal{F}$, the task of program induction with complex function is to find a sequence of functions $\mathcal{P} = \{F_t^{\mathbf{\Theta}_t} \in \mathcal{F}\}_{t=1}^T$ such that:

$$F_T^{\mathbf{\Theta}_T} \circ F_{T-1}^{\mathbf{\Theta}_{T-1}} \cdots \circ F_1^{\mathbf{\Theta}_1} (\mathcal{O}_{\text{in}}) \rightarrow \mathcal{O}_{\text{out}}. \tag{1}$$

Notice that T depends on the underlying ground-truth program in generating the input-output pair and is not observed in the data. Moreover, since we do not want $\mathcal{P}$ to have an infinity length, we transform the task into a constrained optimization problem by minimizing the cost of $\mathcal{P}$ as follows:

$$\begin{aligned} &\underset{\mathcal{P}}{min}\, \mathcal{C}(\mathcal{P}) \\ s.t. \quad & F_T^{\mathbf{\Theta}_T} \circ F_{T-1}^{\mathbf{\Theta}_{T-1}} \cdots \circ F_1^{\mathbf{\Theta}_1} (\mathcal{O}_{\text{in}}) \rightarrow \mathcal{O}_{\text{out}}, \end{aligned} \tag{2}$$

where $\mathcal{C}(\cdot)$ is a metric that measures the cost of a sequence of functions. In the following, for notation convenience, we also write $\mathcal{O}_{\text{in}}$ as $\mathcal{O}_0$ and $\mathcal{O}_{\text{out}}$ as $\mathcal{O}_T$.

2.2 Comparison with simple functions

The general goal of our paper is aligned with previous program induction tasks, but we consider complex functions with a huge and heterogeneous action space, which cannot be solved by traditional program induction models.

As shown in Fig 1(a), the Karel task (Pattis et al., 1981) is a typical example of program induction and includes only four parameter-free primitive functions, i.e., the robot *Karel* can only choose from these four actions. In Fig. 1(b), Balog et al. (2017) consider the sequence manipulation problem which includes 15 primitive functions. It is also worth mentioning that these primitive functions contain none or a very limited number of parameters. For convince, Balog et al. (2017) directly flatten all the primitive functions and associated parameters, obtaining a action space of 34.

In contrast, as shown in Fig 1(c), we consider visual program induction problems where all the primitive functions contain at least one position parameter and an extra color parameter. Therefore, a task with five primitive functions already results in the whole action space being 2,349, which is two orders of magnitude larger than previous works (Appendix A explains how these numbers are calculated in details).

2.3 Problem Analysis and Literature Solutions

The core of solving program induction tasks relies on the fact that all functions are executable. Then, our goal turns into finding the following transition dynamics:

$$F_t^{\mathbf{\Theta}_t} \sim P(F^{\mathbf{\Theta}} | \mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F}),\ \mathcal{O}_t = F_t^{\mathbf{\Theta}_t}(\mathcal{O}_{t-1}). \tag{3}$$

The transition dynamic is executed iteratively until we reach the target state $\mathcal{O}_T$ or reach a preset max iteration step T_{max}. Then, functions along the success trace are treated as a valid program. Next, we review the literature for solving the transition dynamics.

Solving via Exhaustive Search. Search-based methods are common solutions when the action space is not large as we can simply "try" all the possible programs until finding $\mathcal{O}_T$. Naively, by uniformly exploring the action space, i.e., $P(F^{\mathbf{\Theta}}|\mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F}) = 1/|\mathcal{F}|, \forall F^{\mathbf{\Theta}} \in \mathcal{F}$, we can define an order to enumerate all the possible programs of length T_{max}. Though search-based methods can guarantee a valid solution when we try every possible program, when the action space becomes huge, naively trying out all feasible candidate programs is not feasible in practice.

Solving via Learning. In the modern AI era, we can also learn the transition dynamics using neural networks, i.e., $P(F^{\mathbf{\Theta}}|\mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F}) = \frac{1}{\mathbf{Z}} q(F^{\mathbf{\Theta}}, \mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F})$, where $q(\cdot)$ is a neural network to predict the probability distribution of $F^{\mathbf{\Theta}}$ to be the next function and $\mathbf{Z}$ is the normalization term.

Though achieving some successes, however, those learning-based solutions require ground-truth programs $\{F_t^{*\mathbf{\Theta}_t^*}\}_{t=1}^{T}$ as supervision signals during train (Lezama, 2008; Tian et al., 2019), which is expensive or even infeasible to obtain for most program induction tasks.

Solving via Searching and Learning. More recently, Balog et al. (2017) investigate the possibility of combing exhaustive search and learning-based methods where a neural network $q(\cdot)$ is also utilized to learn $P(F^{\mathbf{\Theta}}|\mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F})$. But instead of using ground-truth programs as supervision, the neural network $q(\cdot)$ is directly trained from final search results. This idea is shown successful in (Balog et al., 2017). However, when it comes to the complex functions considered in this paper, this basic search-and-learn framework suffers from the problem of the huge and heterogenous action space, and the learning of $q(\cdot)$ becomes fragile and easy to collapse into trivial local minima.

3 SEMF: Self-Exploratory-Function-Modularization

As explained in Section 2.3, the existing methods cannot model complex functions and suffer from the huge and heterogeneous action space. In this section, we solve this problem by reformulating the program induction task with complex functions as learning function planning and function parameter prediction, where the former decides which function class should we use and the latter decides what parameters should be learned. The intuition is that functions and learnable parameters should be considered separately to better explore the action space.

This section is organized as follows: in Section 3.1, we reformulate the problem into two stages of "function planing" and "parameter prediction"; in Section 3.2, we introduce Modularized Functions proposed in this paper; in Section 3.3, we introduce the Self-Exploratory Learning Framework to efficiently explore this two-stage action space.

3.1 Task Reformulation

To handle complex functions considered in this paper, we first split Eq. (3) into two parts: function planing and parameter prediction, where the former focus on predicting the function class F_t while the latter focus on predicting parameters $\mathbf{\Theta}_t$:

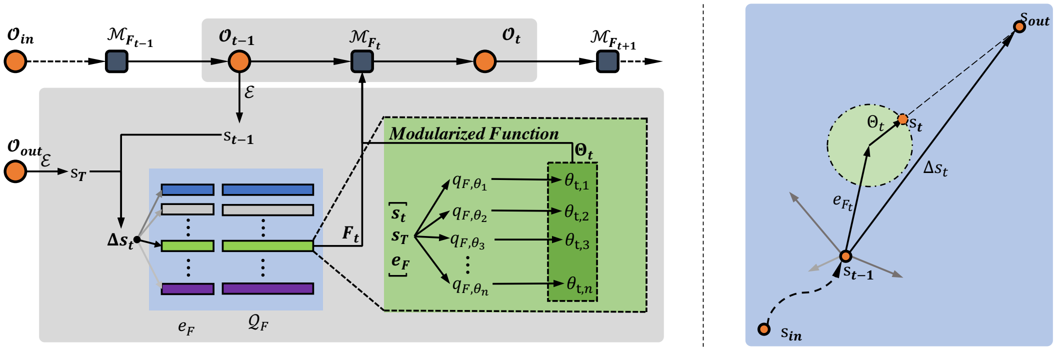
$$F_t \sim P_F(F|\mathcal{O}_{t-1}, \mathcal{O}_T; \mathbf{F}),\ \mathbf{\Theta}_t \sim P_{\mathbf{\Theta}}(\mathbf{\Theta}|\mathcal{O}_{t-1}, \mathcal{O}_T; F, \Xi_F),\ \mathcal{O}_t = F_t^{\mathbf{\Theta}_t}(\mathcal{O}_{t-1}). \tag{4}$$

The decoupling of function class prediction and parameter prediction facilitates exploring the huge action space: we first determine the function and then find the best parameters, rather than directly selecting from the whole action space. From another perspective, the function class prediction can be regarded as modeling function's long-term planing dynamic since the function class usually determines the main functionality, while the parameter prediction models function's short-term transition dynamic by selecting the best parameters.

3.2 Modularized Functions

Instead of implementing Eq. (4) with two separated models as in (Tian et al., 2019), in this paper, we propose the idea of modularized functions, where a modularized function for $F^{\mathbf{\Theta}}$ is defined as $\mathcal{M}_F = (\boldsymbol{e}_F, \mathcal{Q}_F)$ where $\boldsymbol{e}_F$ is a special designed embedding used in function planning and $\mathcal{Q}_F$ is a set of neural networks used to predict function parameters.

Shared Visual Encoder. Due to the complexity of visual scenarios, we firstly introduce a shared encoder $\mathcal{E}$ to encode the raw input space $\mathcal{O}$ into an embedding space, i.e., $\mathbf{s}_i = \mathcal{E}(\mathcal{O}_i) \in \mathbb{R}^h$, where h is the dimensionality of the embedding space.



(a)

(b)

Figure 2: Model illustration. (a) The process of computing function class F_t and parameters $\mathbf{\Theta}_t$ from $\mathcal{O}_{t-1}$ to $\mathcal{O}_t$. (b) An illustration from $\mathcal{O}_{t-1}$ to $\mathcal{O}_t$ in the embedding space, where $\mathbf{s}_i = \mathcal{E}(\mathcal{O}_i)$ is the state embedding vector and $\mathbf{e}_F$ is the function class embedding. F_t determines main direction while $\mathbf{\Theta}_t$ further revises the direction of the function towards $\mathcal{O}_{out}$.

Function Planning. Each function $F \in \mathbf{F}$ is assigned with a vector $\boldsymbol{e}_F \in \mathbb{R}^h$ which is used for the long-term function planning. The probability for function F_t is defined as:

$$F_t \sim P_F(F|\mathcal{O}_{t-1}, \mathcal{O}_T; \mathbf{F}) = \frac{1}{\boldsymbol{Z}_F}\sigma\left(\Delta\mathbf{s}_t^\top \mathbf{e}_F\right), \forall F \in \mathbf{F}, \tag{5}$$

where $\Delta\mathbf{s}_t = \mathcal{E}(\mathcal{O}_T) - \mathcal{E}(\mathcal{O}_{t-1})$, $\mathbf{Z}_F$ is the normalization term, and σ is an activation function. In a nutshell, we calculate the difference $\Delta\mathbf{s}_t$ between our desired output $\mathcal{O}_T$ and the current state $\mathcal{O}_{t-1}$ and select functions with large inner product similarities in the embedding space.

Parameter Prediction. After predicting F_t using $P_F(F|\mathcal{O}_{t-1}, \mathcal{O}_T; \mathbf{F})$, we use $\mathcal{Q}_{F_t}$ to calculate the probability of function parameters $\mathbf{\Theta}_t$ based on $(\mathcal{O}_{t-1}, \mathcal{O}_T)$. For function F with parameters $\mathbf{\Theta}_F = [\theta_{F,1}, \theta_{F,2}, .., \theta_{F,n_F}]$, we use n_F neural networks $\mathcal{Q}_F = \{q_{F,\theta_i}\}_{i=1}^{n_F}$ to model the marginal distribution of $\mathbf{\Theta}_F$ as:

$$\begin{aligned}\theta_{F,i} &\sim P_{F,\theta_i}\left(\theta|\mathcal{O}_{t-1}, \mathcal{O}_T; \Xi_{F,i}\right) \\ &= \frac{1}{\boldsymbol{Z}_\Theta}\sigma\left(q_{F,\theta_i}\left(\mathcal{E}(\mathcal{O}_T), \mathcal{E}(\mathcal{O}_{t-1}), \mathbf{e}_F\right)\right)_{\theta_{F,i}}.\end{aligned} \tag{6}$$

Based on F_t and $\mathcal{Q}_{F_t}$, the probability distribution for $\mathbf{\Theta}_t$ is:

$$\mathbf{\Theta}_t \sim P_{\mathbf{\Theta}}(\mathbf{\Theta}|\mathcal{O}_{t-1}, \mathcal{O}_T; F_t, \mathbf{\Xi}_{F_t}) = \prod_{i=1}^{n_{F_t}} P_{F_t,\theta_i}\left(\theta|\mathcal{O}_{t-1}, \mathcal{O}_T; \Xi_i\right). \tag{7}$$

Using the above definition of modularized function $\mathcal{M}_F$, we further illustrate the computing process of our proposed method in Fig. 2. As shown in Fig. 2(a), given $\mathcal{O}_T$ as the target, the inner product between $\mathbf{e}_F$ and $\Delta\mathbf{s}_t = \mathcal{E}(\mathcal{O}_T) - \mathcal{E}(\mathcal{O}_{t-1})$ is computed to select a function class F_t. After that, $\{\theta_{F_t,i}\}_{i=1}^{n_{F_t}}$ is computed and gathered into $\mathbf{\Theta}_t$. Notice that, during function planning, we do not consider how to learn the parameters. Similarly, during parameter prediction, $\mathcal{Q}_F$ tries to find the best parameters assuming the function class F is already selected. In Fig. 2(b), a brief illustration of the transition dynamic is given. In the embedding space, the function class F_t is used to determine the main direction using $\mathbf{e}_{F_t}$, while $\mathbf{\Theta}_t$ is selected from the parameter space to further revise this direction conditioned on $\mathbf{e}_{F_t}$ to find the best $F_t^{\mathbf{\Theta}_t}$.

3.3 The Self-Exploratory Learning Framework

Next, we introduce our Self-Exploratory Learning Framework, which includes a search phase and a training phase.

Our search method is based on the modularized function and the Monto-Carlo-Tree-Search (MCTS, see (Browne et al., 2012) for more details) algorithm. Due to the huge action space and the non-zero-sum-game nature of this task, vanilla MCTS would still meet extra challenges:

1. The parameter space is still huge even with our proposed modularization, limiting the search efficiency.
2. The successful traces are extremely sparse, i.e., most of the searched programs from $\mathcal{O}_{in}$ cannot lead to $\mathcal{O}_{out}$, and thus could not provide very informative supervision.

To tackle these two challenges, we introduce the following two improvements into MCTS:

1. Increase the parameter search width, i.e., select multiple candidates when searching for parameters rather than search one trace in the standard MCTS;
2. Utilize failed traces as an extra supervison.

Building the Search Tree. One key of MCTS is to build a search tree that enumerates program traces with high probabilities, which includes **Select Child**, **Expand Node**, and **Backup**:

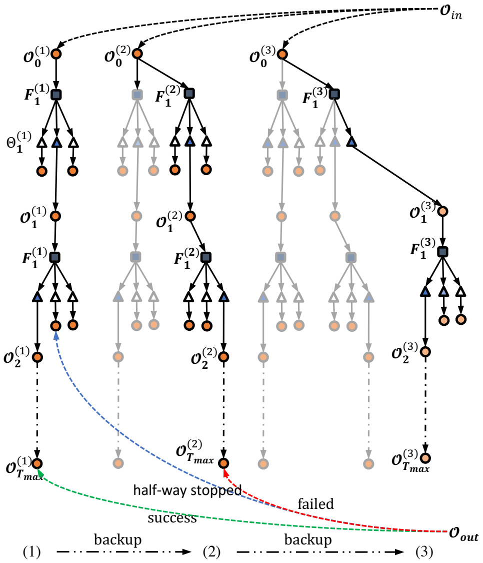


Figure 3: An illustrating example of the modified MCTS algorithm to build a search tree in this paper. We assume three parameters are selected for each function and show the first three traces (best view in colors).

Select Child. Starting from the root node indicating $\mathcal{O}_0^{(1)} = \mathcal{O}_{in}$, we select a function $F_t^{\boldsymbol{\Theta}_t}$ with highest scores to attach to the search tree. Specifically, since there are two types of distributions for function planning and parameter prediction, we make some modification compared to standard MCTS. When choosing a function node F_t, we directly select the function with highest score and append it to the search tree. For the parameter prediction, rather than selecting a single set of parameters, we select the top-k sets of parameters with highest scores to better explore the huge parameter space.
Expand Node. In the "expansion" process, given one function class F_t and multiple sets of parameters $\{\boldsymbol{\Theta}_{t,i}\}$, all the possible output are computed as $\{F_t^{\boldsymbol{\Theta}_{t,i}}(\mathcal{O}_{t-1})\}$. Then we randomly select one of them to expand for the next round of **Select Child.**
Backup. After the expansion process reaches the maximum depth T_{max}, we compare $\mathcal{O}_{T_{max}}^{(i)}$ and all other intermediate $\mathcal{O}_t^{(i)}$ with $\mathcal{O}_{out}$, and backup the statistics, which is the visit count in our scenario. After statistics backup, We continue next round **Select Child** from the root, i.e., $\mathcal{O}_0^{(i+1)} = \mathcal{O}_{in}$.

The score function for selecting a child node $u \in \Omega(v)$, where $\Omega(\cdot)$ denotes all possible children of the current node v, is defined as:

$$score(u) = \frac{1}{\sqrt{1 + \beta \cdot \alpha(u)}} \cdot P(u), \tag{8}$$

where $\alpha(u)$ is the visit count of the child node u, β is a scaling hyper-parameter, and $P(\cdot)$ is the learnable distribution defined as Eq. (5) (for function planing) or Eq. (7) (for parameter prediction). In essence, the learnable distribution encourages searching most potential candidates, while penalizing frequently visiting the same nodes within a tree and thus helps exploration. More details could be found in Appendix C.

Transform into Training Data. After building the search tree, as shown in Fig 3, there are three type of possible traces: (1) successful traces that reach the target (represented as **green** dash line), i.e., $\mathcal{O}_t^{(i)} = \mathcal{O}_{out}$; (2) traces that reach the maximum depth but still do not reach the target (represented as **red** dash line), i.e., $\mathcal{O}_{T_{max}}^{(i)} \neq \mathcal{O}_{out}$; (3) other half-way stopped traces that no dot reach the target (represented as **blue** dash line), i.e., $\mathcal{O}_{t'}^{(i)} \neq \mathcal{O}_{out}, t' \neq T_{max}$. We denote these traces as $\mathcal{T}_s$, $\mathcal{T}_f$, $\mathcal{T}_o$, respectively, where each search step in these traces is in the form of $(\mathcal{O}_{t-1}, \mathcal{O}_t, \mathcal{O}_{out}, F_t, \boldsymbol{\Theta}_t)$ where $F_t^{\boldsymbol{\Theta}_t}(\mathcal{O}_{t-1}) \rightarrow \mathcal{O}_t$.

Training the Model. Given the traces above as training data, the function planing model is optimized using the following loss function:

$$\mathcal{L}_F = \sum_{(\mathcal{O}_{t-1},\mathcal{O}_t,\mathcal{O}_{out},F_t)\in\mathcal{T}_s} P_F(F_t)\cdot\log P_F(F_t) + \sum_{(\mathcal{O}_{t-1},\mathcal{O}_t,\mathcal{O}_{out},F_t)\in\mathcal{T}_f} (1-P_F(F_t))\cdot\log(1-P_F(F_t)), \tag{9}$$

where $P_F(F_t)$ is defined in Eq. (5).

For parameter prediction, since we do not have supervision and the action space is large, we propose a method to optimize the per-step improvement, i.e., only considering the relative improvement compared to the last step. Formally, we design a function $\mathcal{D}(\cdot)$ to measure the quality of a searched state $\mathcal{O}_t$ compared to our goal $\mathcal{O}_{out}$. We realize $\mathcal{D}(\cdot)$ as the edit distance in this paper, which can be easily generalized. Then, the per-step improvement is measured as $\Delta\mathcal{D}(\mathcal{O}_t, \mathcal{O}_{t-1}) = \mathcal{D}(\mathcal{O}_t, \mathcal{O}_{out}) - \mathcal{D}(\mathcal{O}_{t-1}, \mathcal{O}_{out})$. The parameter prediction is trained with the following cross-entropy loss:

$$\mathcal{L}_{\mathbf{\Theta}} = \sum_{(\mathcal{O}_{t-1},\mathcal{O}_t,\mathcal{O}_{out},F_t,\mathbf{\Theta}_t)\in\mathcal{T}_s\cup\mathcal{T}_f\cup\mathcal{T}_p} Q(\mathbf{\Theta}_t)\log P_{\mathbf{\Theta}}(\mathbf{\Theta}_t) + (1-Q(\mathbf{\Theta}_t))\log(1-P_{\mathbf{\Theta}}(\mathbf{\Theta}_t)), \tag{10}$$

where $P_{\mathbf{\Theta}}(\mathbf{\Theta}_t)$ is defined in Eq. (7) and $Q(\mathbf{\Theta}_t) = \rho\left(\Delta\mathcal{D}\left(F_t^{\mathbf{\Theta}_t}(\mathcal{O}_{t-1}), \mathcal{O}_{t-1}\right)\right)$ is the normalized relative improvement when choosing parameter $\mathbf{\Theta}_t$ at the current step.

4 Experiments

In this section, we conduct experiments for the visual program induction task. We first introduce the data generation process and experimental setup in Sec. 4.1. In Sec. 4.2, we report the results of our proposed model and state-of-the-art search-based and learning-based baselines. In Sec. 4.3, we analyze the training efficiency and generalization ability. Besides, we give some results illustration in Appendix. D.

4.1 Data Generation and Experimental Setup

Data Generation. We specifically design a visual program induction dataset inspired by the visual patterns from Abstraction and Reasoning Corpus (Chollet, 2019) and focus on pixel modification primitive functions on 5×5 grids with ten colors (see Fig. 1(c) for an example). The training set and testing set are generated by randomly selecting primitive functions to form a program, and then applying the program to a random input observation $\mathcal{O}_{in}$ to obtain $\mathcal{O}_{out}$. In this paper, we generate $10,000$ instances for training and $1,000$ instances for testing. We fix the maximum length of the program as $T = 3$ unless stated otherwise. Ground-truth program is used to measure the model performance. More details and examples about the dataset could be found in Appendix A.

Model Instantiation. The instantiation of our proposed method is as follows. The encoder $\mathcal{E}$ is a convolutional neural network (CNN) with four convolutional layers and one MLP layer. The neural networks for parameter prediction q_{F,θ_i} has 4 MLP layers, which are shared across different parameters, and one parameter-specific MLP layer with the output dimensionality the same as $|\Xi_{F,i}|$. The model is optimized with Adam (Kingma & Ba, 2014) optimizer and the learning rate is 0.001. The training are conducted with 1 GTX 3090 GPU and 60 CPU cores for two days.

Comparing Methods. We compare the following methods. (1) Basic search-based algorithms, including **Depth-First-Search** (DFS), **Beam Search**, and A specifically-designed **MCTS** algorithm for this task. These baselines do not have a training process. (2) Our proposed SEMF model. Since SEMF is a general learning framework, we equip it with different search methods mentioned above, i.e., DFS, Beam Search, and MCTS, *in the testing phase*. (3) Supervised baseline, which is inspired by program synthesis (Sun et al., 2018; Tian et al., 2019) and uses ground-truth programs as supervision. Notice that the supervised baseline should be considered as an upper bound since it utilizes ground-truth programs. More details about these methods could be found in Appendix. B.

Metrics. We consider two metrics in this paper: **Hit@N**, i.e., whether a successful trace is found within N trials, which measures the inference accuracy, and **AvgTime**, i.e., the average inference time to solve a problem, which measures the model efficiency.

Table 1: The overall performance Comparison. For training and testing, “AvgTime” measures the average search time for a maximum of 50 trails. For search-based baselines, since they do not have a training stage, we directly apply them to the training data and testing data, respectively. For SEMF and Supervised methods, different search methods only affect the testing phase.

Model	Search Method	Training		Testing			
		Hit@50↑	AvgTime↓	Hit@10↑	Hit@50↑	Hit@200↑	AvgTime↓
Search	Beam	1.7%	$>1000s$	0.2%	1.8%	3.5%	$>1000s$
	DFS	1.1%	$>1000s$	0.3%	1.1%	2.6%	$>1000s$
	MCTS	2.4%	$>1000s$	0.5%	2.3%	2.7%	$>1000s$
SEMF	+Beam	85.5%	$200s$	61.0%	74.4%	86.1%	$358ms$
	+DFS	85.5%	$200s$	66.3%	67.3%	87.8%	$191ms$
	+MCTS	85.5%	$200s$	72.5%	82.1%	90.7%	$108ms$
Supervised	+MCTS	93.8%	$23ms$	85.1%	92.1%	94.4%	$79ms$

4.2 Results

We report the results of all the comparing methods in Table 1 and make the following observations.

First, search-based methods fail miserably to solve most of the tasks within an acceptable time, indicating that these classical methods cannot solve the visual program induction tasks with complex functions. The search space of exhaustive search grows exponentially with respect to the length of the program. By setting $T_{max} = 3$, which is the ground-truth length of the dataset, the search space would have a size of $2049^3 > 8 \times 10^9$, which is infeasible in practice. The results also motivate combining search-based methods with learning-based solutions.

Our proposed SEMF model significantly outperforms search-based methods. For example, SEMF+MCTS achieves more than 70%, 80%, and 90% accuracy for Hit@10, Hit@50, and Hit@200, respectively. Besides, the average inference time for SEMF+MCTS is around 100 milliseconds, which is more than 1000x faster than search-based methods.

Comparing different search methods, our proposed MCTS significantly outperforms DFS and Beam search. For example, MCTS improves the accuracy by about 8% for Hit@50 while reducing the average inference time by more than 43%. We attribute the strong performance of MCTS to our tailed design for the visual program induction task.

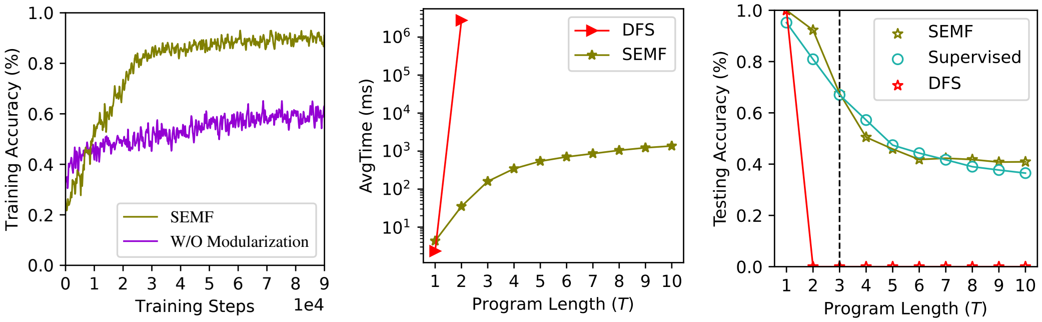
Our proposed method achieves comparable performance compared to the supervised baseline. Notice that the supervised baseline needs ground-truth programs of the training data, which is expensive or infeasible to collect for most real visual program induction tasks.

4.3 Training Efficiency and Generalization Ability

In this section, we analyze the training efficiency model robustness and generalization Ability of our proposed method.

Training efficiency First, we compare a variant of our proposed method without function modularization, i.e., directly choosing from the 2,349 functions in the action space. The results are shown in Fig. 4(a). Clearly, without function modularization, the training accuracy is much lower and the convergence rate is much slower. Next, we compare our proposed method with DFS when the program length varies. The results are shown in Fig. 4(b). DFS can barely handle when the program length if larger than 1. In contrast, the average inference time of our proposed method only grows marginally as the program length increases, demonstrating the superior efficiency.

Generalization ability. Here we test the generalization ability of our proposed method with respesct to the program length T. Specifically, the model is trained with $T_{max} = 3$, but tested with varying program lengths of $T = 1, ..., 10$. The results are shown in Fig. 4(c). DFS could not handle programs when $T > 1$ with a time budget of 1,000 seconds. On the other hand, our model and the supervised baseline can fairly generalize to data with a larger program length. Interestingly, when



(a) (b) (c)

Figure 4: Training efficiency and generalization ability analysis results.

the program length exceeds 7, our proposed method even slightly outperforms the supervised baseline. The results clearly show that our proposed method has a good generalization ability without using ground-truth programs as supervisions.

5 Related Work

Program Induction. Program induction analogs the ability of human thinking, which has regained research interests recently (Lake et al., 2015; Chollet, 2019; Qi et al., 2021). Classical program induction methods adopt search-based or other enumeration-based methods like Depth-First-Search (DFS) and SMT-based Solvers (Lezama, 2008; Feser et al., 2015). Recently, the combination of deep learning and program induction is also a trending direction (Balog et al., 2017; Irving et al., 2016). Besides technical advancements, there are also new datasets like ARC (Chollet, 2019), PQA (Qi et al., 2021), etc. In this paper, we focus on visual program induction tasks that require complex primitive functions, which is mostly inspired by ARC that aims to solve complex visual abstract reasoning tasks. Compared with the tasks considered by (Balog et al., 2017; Irving et al., 2016), our task is much more complicated especially considering the complex functions and visual domains.

Program Synthesis. Program Synthesis could be seen as the supervised version of program induction where the annotated programs are provided as supervision, also known "using machine learning to write programs". Parisotto et al. (2017); Bunel et al. (2018); Devlin et al. (2017) have applied program synthesis on the Karel environment or string transformation tasks. Ling et al. (2017) solve the task of algebraic word problems, Sun et al. (2018) use similar techniques to learn programs for 2D visual games, and Tian et al. (2019) consider synthesizing 3D programs for 3D shapes. All these methods rely on the supervised setting and require expensive program annotation costs.

Program Induction with latent Programs. As neural networks are good at handling high-dimensional vector data, the modeling of program states rather than explicit programs is also appealing. Famous examples include Neural Turing Machines and its follow-up works (Graves et al., 2014). Neural Logic Machines (Dong et al., 2019) and Neural GPU (Li et al., 2016; Devlin et al., 2017) are also related in the latent program space. In comparison, we focus on explicit program induction which is more explainable and the obtained programs are understandable to humans.

6 Conclusion

In this paper, we are the first to tackle the task of program induction with complex functions for the visual abstract reasoning tasks. We model a function and its corresponding parameters as a unified modules. Moreover, we propose a two-stage MCTS algorithm to efficiently search the action space to provide training supervision. Our method could outperform several strong baselines, especially has a higher efficiency. We believe this method would help us to explore a wider program induction scenarios towards modeling much more complex functions and to achieve machines' intelligence and mental thinking system.

REFERENCES

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. 2017.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2018.

Hyeong Soo Chang, Michael C Fu, Jiaqiao Hu, and Steven I Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, 2005.

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

Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. *Neural Information Processing Systems (NIPS)*, 2017.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2019.

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

Jerry A Fodor. *The language of thought*, volume 5. Harvard university press, 1975.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. *Advances in Neural Information Processing Systems*, 29:2235–2243, 2016.

Aysja Johnson, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. Fast and flexible: Human program induction in abstract reasoning tasks. *arXiv preprint arXiv:2103.05823*, 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

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

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2016.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*, 2017.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *Proceedings of the 4th International Conference on Learning Representations(ICLR)*, 2017.

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

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

Yonggang Qi, Kai Zhang, Aneeshan Sain, and Yi-Zhe Song. Pqa: Perceptual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12056–12064, 2021.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pp. 4790–4799. PMLR, 2018.

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. *International Conference on Learning RepresentationsarXiv preprint arXiv:1901.02875*, 2019.

A Data Generation and Result Illustration

A.1 The primitive Functions

We listed the considered primitive functions in Table. 2.

Table 2: Primitive Functions.

Primitive Functions	***Descriptions***
DOT[X,Y,COL]	draw a **dot** at position **(X, Y)** with color **COL**
VLINE[TX,BX,Y,COL]	draw a **vertical line** from **(TX, Y)** to **(BX, Y)** with color **COL**
HLINE[LY,RY,X,COL]	draw a **horizontal line** from **(X, LY)** to **(X, RY)** with color **COL**
BLOCK[TX, LY, BX, RY, COL]	draw a **block** from **(TX, LY)** to **(BX, RY)** with color **COL**
BORDER[TX, LY, BX, RY, COL]	draw a **rectangle border** from **(TX, LY)** to **(BX, RY)** with color **COL**

In this table, each primitive function is implemented as a small python function. the function and its parameters are then organized by a `pytorch.nn.Module`. An examples of the `DOT` primitive function and its modularized form `Dot` are shown as follows:

```python
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
        self.embedding = torch.rand(size=(H,))
        # ParamNet is another mlp to predict parameters.
        self.params['X'] = ParamNet("X", H, range=MAXSIZE[0])
        self.params['Y'] = ParamNet("Y", H, range=MAXSIZE[0])
        self.params['C'] = ParamNet("C", H, 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: The primitive function DOT and its Modularized Functions

A.2 Generate the dataset

In the main experiment, we test the task under a canvas of shape $(5, 5)$ with ten colors. This setting results in awhole action space as 2349 (See Table. 3), which is much more larger than other tasks ever before (Pattis et al., 1981; Balog et al., 2017).

To generate a dataset, we randomly sample a list of functions and parameters to form a program $\mathcal{P} = \left[F_1^{(\mathbf{\Theta}_1)}, F_2^{(\mathbf{\Theta}_2)}, \cdots F_T^{(\mathbf{\Theta}_T)}\right]$ with $T < T_{max}$, and generate a random Input O_0, and apply the program to $\mathcal{O}_0$ to obtain $\mathcal{O}_T$. Then $(\mathcal{O}_0, \mathcal{O}_T)$ is used as an input-output pairs. Moreover, due to that some function would overwrite previous one (e.g., F_j=`DOT(X,Y,1)` will always overwrite F_i =`DOT(X,Y,2)` if $j > i$), we carefully compare the generating trace $[\mathcal{O}_0, \mathcal{O}_1, \cdots, \mathcal{O}_T]$ to delete those functions that have been overwritten.

Table 3: Function Action Size.

Primitive Functions	***Overall Action Space***
DOT[X,Y,COL]	$C(5,1)\times C(5,1)\times 9 = 5\times 5\times 9 = 225$
VLINE[TX,BX,Y,COL]	$C(5,1)\times C(5,2)\times 9 = 5\times 10\times 9 = 450$
HLINE[LY,RY,X,COL]	$C(5,2)\times C(5,1)\times 9 = 10\times 5\times 9 = 450$
BLOCK[TX, LY, BX, RY, COL]	$C(5,2)\times C(5,2)\times 9 = 10\times 10\times 9 = 900$
BORDER[TX, LY, BX, RY, COL]	$(C(5,2)-4)^2\times 9 = 6\times 6\times 9 = 324$

The training set in this paper consists of 20% programs with length 1, 20% programs with length 2, and 60% programs with length 3. The alpha testing set (Table. 1) has the same distribution as training set. While the beta test set (Figure 4) consists only those datas with a program of exactly length T_{max}. In this paper, the training set contains of 10,000 input-output pairs, all the testing set contains 1000 input-output pairs.

B Implementation and Baselines

B.1 More details on MCTS

The model in this paper is implemented with `pytorch`. Basically, the encoder is a 2-layer CNN neural network while the `ParamNet` (a.k.a, g_θ^F in Eq. 6) is a three layer MLP. The hidden size in this paper is set as 400. At each round of selection, we select 200 parameters, and using GPU to parallelly computing all the output observations. And the whole model is optimized with an `Adam` optimizer with learning rate `1e-3`.

Moreover, to accelerate the training speed, we follow the setting of MuZero (Silver et al., 2016) to implement a muti-processing system based on Ray[1]. The system consists of 60 explorers to continuously explore the function space via the Self-Exploratory algorithm (Alg.1), and explored traces are used to train the model via an online trainer. The trained model weight are used in the following exploration (Fig. 5). The whole framework is launched on a GPU server with two `Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz` CPU processors and two `GTX 3090` GPU processors. The whole learning process takes about two days.

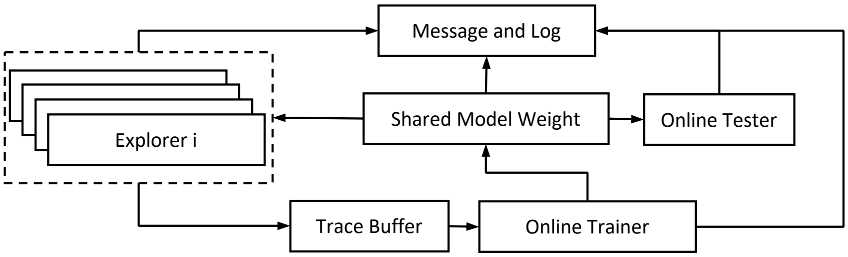


Figure 5: The multi-processing learning framework of this paper.

. We also provide more details about the Self-Exploratory Algorithm in Appendix C.

B.2 Baselines

In this paper, we consider a **supervised** baseline, and several of the search-based baseline **DFS**, **BeamSearch**, and **MCTS**. More details about them are as follows:

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

1. **Supervised**. This baseline is inspired by Lezama (2008); Tian et al. (2019). Basically, based on s_t, two neural networks are used to predict F_t and $\mathbf{\Theta}_t$. This model is supervised by the program $\mathcal{P}$ as described in Sec. A. We make sure that this model has similar parameters with our SEMF. Similar implementation could also be found in Sun et al. (2018).
2. **DFS**. Depth-First-Search(DFS) is one of the most famous search-based method, which has been widely used. Despite those specialized-designed search-method (Lezama, 2008; Gulwani et al., 2011), DFS would still be competitive the general scenarios (Balog et al., 2017). As DFS explore the candidates according to a give order, it would help if we could rerank the candidates and provide a better order for the algorithm.
3. **BeamSearch**. BeamSearch is widely used in nature-language-processing community to avoid local minima by tracing the top K candidates simultaneously. Rather than DFS, BeamSearch would be seen as a constrained version of **Breadth-First-Search (BFS)** by constraining the breadth to K.
4. **MCTS**. MCTS is widely used to make decisions. While in this paper, we found that it could also be used to do searching. We call this algorithm as **MCTS**, which iteratively search from the root node until reach the target or reach maximum depth as described in Alg. 1. **MCTS** would be more efficient in this paper for its consistency with the training process.

C Overall Algorithm

The overall algorithm in this paper includes the **Exploring** phase and the **Training** phase, which could be found in Alg. 2.

Algorithm 1: MCTS-based Exploration

```
Input: $(\mathcal{O}_{in}, \mathcal{O}_{out})$, Action Space $\mathcal{S}$, Maximum Depth $T_{max}$, Number of Traces $N$
Output: Training Traces $\mathcal{T}_s, \mathcal{T}_f, \mathcal{T}_o$
Function Backup ($node$):
    while $node \neq root$ do
        $node.visit$ = $node.visit + 1$
        $node = node.parent$
Function SelectChild($v$):
    return $\arg\max_{u \in \Omega(v)} score(u)$
Function Select-k-Children($v$):
    $\mathbf{u} = []$
    while $len(\mathbf{u}) < k$ do
        $u_i = \arg\max_{u \in \Omega(v), u \notin \mathbf{u}} score(u)$
        $\mathbf{u}$.append($u_i$)
    return $\mathbf{u}$
count = 0 while $count < N$ do
    $node$ = $\mathcal{O}_{in}$
    while $node.depth < T_{max}$ and $node \neq \mathcal{O}_{out}$ do
        $F$ = SelectChild ($node$)
        $\mathbf{\Theta}_1 ... \mathbf{\Theta}_k$ = Select-k-Children($F$)
        Add $F, \mathbf{\Theta}_1, ..., \mathbf{\Theta}_k$ to the tree
        Compute $\mathcal{O}_{t,i} = F^{\mathbf{\Theta}_i}(node)\},\ i = 1, ...k$
        $node = \mathcal{O}_{t,i}, i \sim$ Uniform$(1, k)$
    Backup ($node$)
    count ← count + 1
Transform traces into $\mathcal{T}_s, \mathcal{T}_f, \mathcal{T}_o$ and return
```

C.1 Detailed MCTS algorithm in this paper

A search tree in this task aims to find the programs that enables the transition from $\mathcal{O}_{in}$ to $\mathcal{O}_{out}$. To achieve this, we would start from $\mathcal{O}_{in}$, iteratively select next function and parameter, and repeat this

Algorithm 2: The Self-Explore-Learnable-Functions Algorithm

```
Input: $\mathcal{D} = \{(\mathcal{O}_{in}, \mathcal{O}_{out})\}, \mathcal{F}$;                // Datas and Function Primitives
Output: $P_F(F|\mathcal{O}_{t-1}, \mathcal{O}_{out}, \mathbf{F}); P_{F,\theta_i}(\mathbf{\Theta}|\mathcal{O}_{t-1}, \mathcal{O}_T, \Xi_i)$;     // Eq. 5, and Eq. 7
TrainingQueue = Queue()
DataQueue = Queue()
Function Explorer($P_\theta, \{P_{F,\theta_i}\}$):
    repeat
        $(\mathcal{O}_{in}, \mathcal{O}_{out}) \leftarrow$ DataQueue.get()
        $\mathcal{T}_s, \mathcal{T}_f, \mathcal{T}_o \leftarrow$ SelfExplore($(\mathcal{O}_0, \mathcal{O}_T)$) ;            // ALg. 1
        TrainingQueue.put($\mathcal{T}_s, \mathcal{T}_f, \mathcal{T}_o$)
    until Model Converged;
Function Trainer(TrainingQueue):
    repeat
        $\mathcal{T}_s, \mathcal{T}_f, \mathcal{T}_o$ = TrainingQueue.get()
        $\mathcal{L} \leftarrow \mathcal{L}_F + \mathcal{L}_{\mathbf{\Theta}}$ ;                              // Eq. 9, Eq. 10
        Parameter Update with Adam ;      // Update the model weights with Adam
    until Model Converged;
repeat
    trainer = NewRayProcess(Trainer);
    for rank in range(N) do
        new_explorer = NewRayProcess(Explorer)
        new_explorer.run()
    trainer.run()
    for $(\mathcal{O}_{in}, \mathcal{O}_{out}) \in \mathcal{D}$ do
        DataQueue.put($(\mathcal{O}_{in}, \mathcal{O}_{out})$)
until Model Converged;
```

process until we reach $\mathcal{O}_{out}$. This process is very similar with a DFS search, except that the search order will dynamically change along the tree building process. Following MCTS, we introduce this process as:

- **Child Selection**. Given current observation $\mathcal{O}_t$, the goal of child selection is to select a proper function F_{t+1} and $\mathbf{\Theta}_{t+1}$ for next step. Unlike those scenarios there are a unified action space, in this paper, we have two sequentially action space. Note that the parameter action space is much bigger, we use the following strategy:

$$\begin{aligned} F_{t+1} &= \texttt{SelectCchild}(P(\ \cdot\ |\mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F})) \\ \{\mathbf{\Theta}_{t+1}^{(k)}\} &= \texttt{Select-k-Children}(P(\ \cdot\ |\mathcal{O}_{t-1}, \mathcal{O}_T; \mathcal{F})) \end{aligned} \tag{11}$$

  where the SelectChild function is to compute the next next node according to their score as:

$$score(u) = \frac{1}{\sqrt{1 + \beta \cdot \alpha(u)}} \cdot P(u), \tag{12}$$

  where $\alpha(u)$ is the visit count of the child node u, β is a scaling hyper-parameter, and $P(\cdot)$ is the learnable distribution defined as Eq. (5) (for function planing) or Eq. (7) (for parameter prediction). Eq. 12 is a simplified version of the UCB Chang et al. (2005) score that only remains the most important characteristic: *the score is proportional to the negative square root of visit number.*

- **Node Expansion**. With the select F_i and $\{\mathbf{\Theta}_i\}$. The search tree is expanded to a set of new node $\mathcal{O}_{t+1}^{(1)}, \mathcal{O}_{t+1}^{(2)}, \cdots \mathcal{O}_{t+1}^{(k)}$, where k is the superscripts corresponding to $\mathbf{\Theta}_t^{(k)}$. In the original MCTS algorithm, at each step, only one action is selected. But for our scenarios, we select multiple parameters.

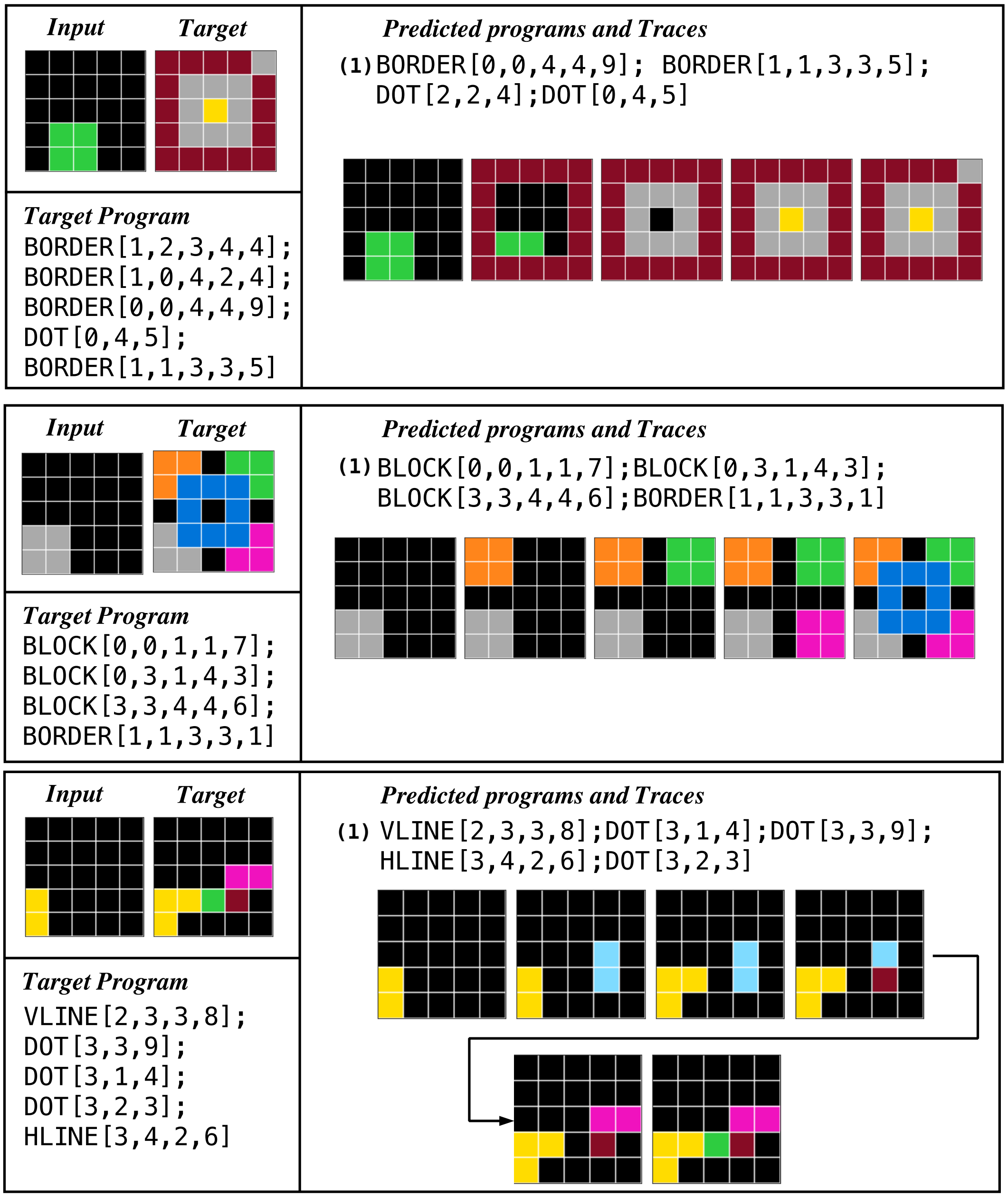
- **Statistics Backup**. After the search reach max search step or reach target $\mathcal{O}_T$, the next task is to backup the statics. In our setting, for those success node, all of its parents receives 1. and the visit count is added 1. For those failed traces, we do not need to backup any statics.

Compared with the vanilla MCTS method. Our algorithm has two major difference:

1. The search process includes two types of distribution. Firstly, the function planning distribution on $\boldsymbol{F}$ defined in Eq. 5. Secondly, a group of distribution for parameter prediction, each of which corresponds to one of the primitive function, defined in Eq. 7 on Ξ_F. This two types of distribution guides the sampling process. Compared with a unified action distribution on $\mathcal{F}$, this two-stage sampling introduces extra sampling complexity, but gains more explainability and robustness for training.
2. Within the two-stage sampling, the function planning is a discrete distribution with $|\boldsymbol{F}|$ dimensions while the parameter prediction is a discrete distribution with $|\Xi_F|$ dimension. The latter dimension is much more bigger than the first one. In this paper, we tackle this problem by selecting more than one child when it comes to the parameter selection parts. An extra

D Result Illustrations

Fig. 6 illustrates some of the input output observations and successful model predictions. Also In Fig. 7, we show an case that the prediction is not the same with the ground-truth. This is usually the case: *The ground-truth program always is not the unique answer.* This would provide more inspiration that the supervised training model could be further improved if considering the uniqueness of the program.


Input
Target
Target Program
Predicted programs and Traces

```
BORDER[1,2,3,4,4];
BORDER[1,0,4,2,4];
BORDER[0,0,4,4,9];
DOT[0,4,5];
BORDER[1,1,3,3,5]
```

```
(1) BORDER[0,0,4,4,9]; BORDER[1,1,3,3,5];
    DOT[2,2,4];DOT[0,4,5]
```

Input
Target
Target Program
Predicted programs and Traces

```
BLOCK[0,0,1,1,7];
BLOCK[0,3,1,4,3];
BLOCK[3,3,4,4,6];
BORDER[1,1,3,3,1]
```

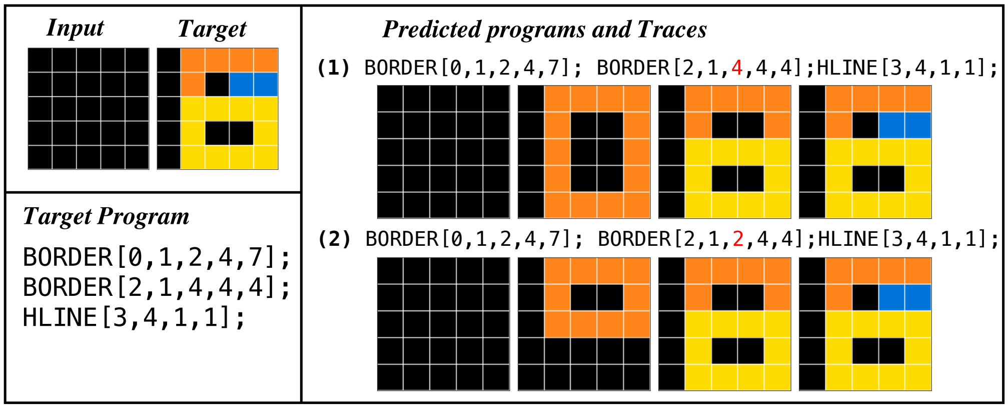
```
(1) BLOCK[0,0,1,1,7];BLOCK[0,3,1,4,3];
    BLOCK[3,3,4,4,6];BORDER[1,1,3,3,1]
```

Input
Target
Target Program
Predicted programs and Traces

```
VLINE[2,3,3,8];
DOT[3,3,9];
DOT[3,1,4];
DOT[3,2,3];
HLINE[3,4,2,6]
```

```
(1) VLINE[2,3,3,8];DOT[3,1,4];DOT[3,3,9];
    HLINE[3,4,2,6];DOT[3,2,3]
```

Figure 6: Results Illustrations.



```
BORDER[0,1,2,4,7];
BORDER[2,1,4,4,4];
HLINE[3,4,1,1];
```

```
(1) BORDER[0,1,2,4,7]; BORDER[2,1,4,4,4];HLINE[3,4,1,1];
(2) BORDER[0,1,2,4,7]; BORDER[2,1,2,4,4];HLINE[3,4,1,1];
```

Figure 7: Results Illustrations.