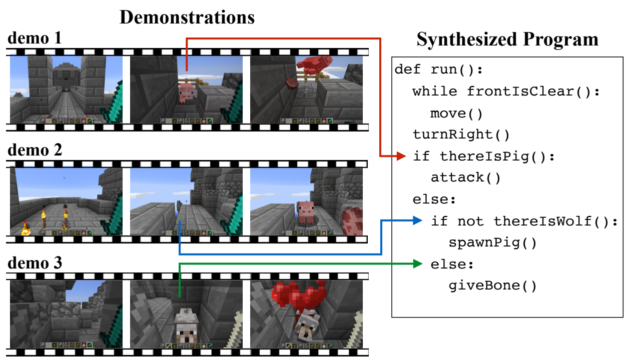
Neural Program Synthesis from Diverse Demonstration Videos

Shao-Hua Sun [*1] Hyeonwoo Noh [*2] Sriram Somasundaram [1] Joseph J. Lim [1]

Abstract

Interpreting decision making logic in demonstration videos is key to collaborating with and mimicking humans. To empower machines with this ability, we propose a neural program synthesizer that is able to explicitly synthesize underlying programs from behaviorally diverse and visually complicated demonstration videos. We introduce a summarizer module as part of our model to improve the network's ability to integrate multiple demonstrations varying in behavior. We also employ a multi-task objective to encourage the model to learn meaningful intermediate representations for end-to-end training. We show that our model is able to reliably synthesize underlying programs as well as capture diverse behaviors exhibited in demonstrations. The code is available at https://shaohua0116.github.io/demo2program.



```
def run():
  while frontIsClear():
    move()
  turnRight()
  if thereIsPig():
    attack()
  else:
    if not thereIsWolf():
      spawnPig()
    else:
      giveBone()
```

Figure 1. An illustration of neural program synthesis from demonstrations. Given multiple demonstration videos exhibiting diverse behaviors, our neural program synthesizer learn to produce interpretable and executable underlying programs. Divergence above occurs based on perception in the second frame.

1. Introduction

Imagine you are watching others driving cars. You will easily notice many common behaviors even if you know nothing about driving. For example, cars stop when the traffic light turns to red and move again when the light turns to green. Cars also slow down when pedestrians are seen jay-walking. Through observation, humans can abstract behaviors and understand the reasoning behind behaviors – especially extracting the structural relationship between actions (*e.g*. start, slow down, stop) and perception (*e.g*. light, pedestrian).

Can machines also reason decision making logic behind behaviors? There has been tremendous effort and success in understanding behaviors such as recognizing actions (Simonyan & Zisserman, 2014), describing activities in languages (Venugopalan et al., 2015), and predicting future outcomes (Srivastava et al., 2015). Yet, interpreting reasons behind behaviors is relatively unexplored and is a crucial skill for machines to collaborate with and mimic humans. Hence, our goal is to step towards developing a method that can interpret perception-based decision making logic from diverse behaviors seen in multiple visual demonstrations.

Our insight is to exploit declarative programs, structured in a formal language, as representations of decision making logics. The formal language is composed of action blocks, perception blocks, and control flow (*e.g.* if/else). Programs written in such a language can explicitly model the connection between an observation (*e.g.* traffic light, biker) and an action (*e.g.* stop). An example is shown in Figure 1[1]. Described in a formal language, programs are logically interpretable and executable. Thus, the problem of interpreting decision making logic from visual demonstrations can be reduced to extracting an underlying program.

In fact, there have been many neural network frameworks proposed recently for program induction or synthesis. First, a variety of frameworks (Kaiser & Sutskever, 2016; Reed & De Freitas, 2016; Xu et al., 2018; Devlin et al., 2017a) propose to induce latent representations of underlying programs. While they can be efficient at mimicking desired behaviors, they do not explicitly yield interpretable programs, resulting

[*]Equal contribution [1]Department of Computer Science, University of Southern California, California, USA [2]Department of Computer Science and Engineering, POSTECH, Pohang, Korea. Correspondence to: Shao-Hua Sun <shaohuas@usc.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

[1]The illustrated environment is not tested in our experiments.

in inexplicable failure cases. On the other hand, another line of work (Devlin et al., 2017b; Bunel et al., 2018) directly synthesize programs from input/output pairs, giving full interpretability. While successful, the limited information in the input/output pairs restricts applicability in synthesizing programs with rich expressibility. Hence, in this paper, we develop a model that synthesizes programs from visually complex and sequential inputs that demonstrate more branching conditions and long term effects, increasing the complexity of the underlying programs.

To this end, we develop a program synthesizer augmented with a summarizer module that is capable of encoding the interrelationship between multiple demonstrations and summarizing them into compact aggregated representations. In addition, to enable efficient end-to-end training, we introduce auxiliary tasks to encourage the model to learn the knowledge that is essential to infer an underlying program.

We extensively evaluate our model in two environments: a fully observable, third-person environment (Karel) and a partially observable, egocentric game (ViZDoom). Our experiments in both environments with a variety of settings present the strength of explicitly modeling programs for reasoning underlying conditions and the necessity of the proposed components (the summarizer module and the auxiliary tasks).

In summary, in this paper, we introduce a novel problem of program synthesis from diverse demonstration videos and a method to address it. This substantially enables machines to explicitly interpret decision making logic and interact with humans. We also demonstrate that our algorithm can synthesize programs reliably on multiple environments.

2. Related Work

Program Induction Learning to perform a specific task by inducing latent representations of underlying task-specific programs is known as *program induction*. Various approaches have been developed: designing end-to-end differentiable architectures (Graves et al., 2014; 2016; Zaremba & Sutskever, 2015; Kaiser & Sutskever, 2016; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Neelakantan et al., 2015), learning to call subprograms using step-by-step supervision (Reed & De Freitas, 2016; Cai et al., 2017), and few-shot program induction (Devlin et al., 2017a). Contrary to our work, those method do not return explicit programs.

Program Synthesis The line of work in *program synthesis* focuses on explicitly producing programs that are restricted to certain languages. (Balog et al., 2017) train a model to predict program attributes and used external search algorithms for inductive program synthesis. (Parisotto et al., 2017; Devlin et al., 2017b) directly synthesize simple string transformation programs. (Bunel et al., 2018) employ reinforcement learning to directly optimize the execution of generated programs. However, those methods are limited to synthesizing programs from input-output pairs, which substantially restricts the expressibility of the programs that are considered; instead, we address the problem of synthesizing programs from full demonstrations videos.

$$
\begin{aligned}
\text{Program } m &\coloneqq \text{def run}() : s \\
\text{Statement } s &\coloneqq \text{while}(b) : (s) \mid s_1; s_2 \mid a \mid \text{repeat}(r) : (s) \\
&\quad \mid \text{if}(b) : (s) \mid \text{ifelse}(b) : (s_1) \text{ else} : (s_2) \\
\text{Repetition } r &\coloneqq \text{Number of repetitions} \\
\text{Condition } b &\coloneqq \text{percept} \mid \text{not } b \\
\text{Perception } p &\coloneqq \text{Domain dependent perception primitives} \\
\text{Action } a &\coloneqq \text{Domain dependent action primitives}
\end{aligned}
$$

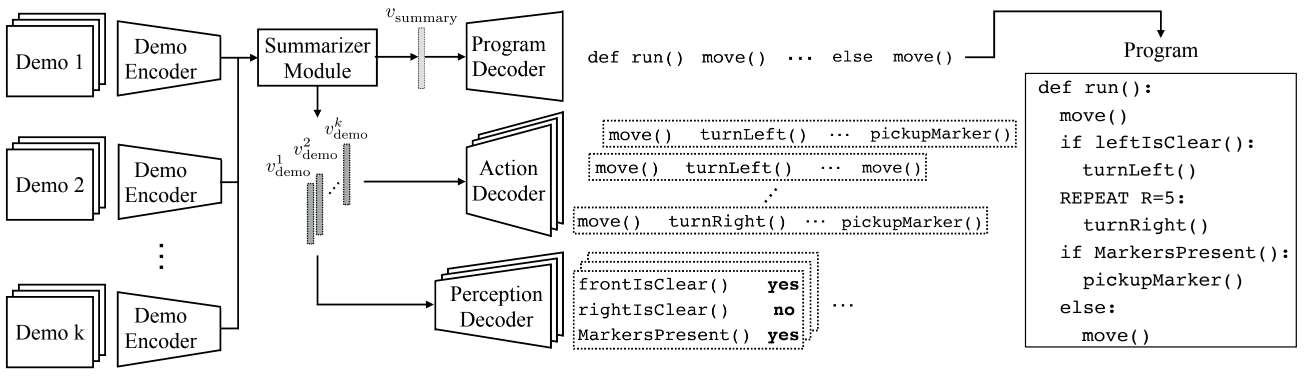
Figure 2. Domain specific language for the program representation. The program is composed of domain dependent perception and action primitives and control flows.

Imitation Learning The methods that are concerned with acquiring skills from expert demonstrations, dubbed *imitation learning*, can be split into *behavioral cloning* (Pomerleau, 1989; 1991; Ross et al., 2011) which casts the problem as a supervised learning task and *inverse reinforcement learning* (Ng et al., 2000) that extracts estimated reward functions given demonstrations. Recently, (Duan et al., 2017; Finn et al., 2017; Xu et al., 2018) have studied the task of mimicking given few demonstrations. This line of work can be considered as *program induction*, as they imitate demonstrations without explicitly modeling underlying programs. While those methods are able to mimic given few demonstrations, it is not clear if they could deal with multiple demonstrations with diverse branching conditions.

3. Problem Overview

In this section, we define our formulation for program synthesis from diverse demonstration videos. We define programs in a domain specific language (DSL) with perception primitives, action primitives, and control flows. Action primitives define the way that agents can interact with an environment, while perception primitives describe how agents can percept it. Control flow can include if/else statements, while loops, repeat statements, and simple logic operations. An example of control flow introduced in (Pattis, 1981) is shown in Figure 2. Note that we focus on perceptions with boolean types in this paper, although a more generic perception type constraint is possible.

A program η is a deterministic function that outputs an action $a \in \mathcal{A}$ given a history of states at time step t, $H_t = (s_1, s_2, ..., s_t)$, where $s \in \mathcal{S}$ is a state of the environment. The generation of an action given the history of states is


Demo 1
Demo 2
Demo k
Demo Encoder
Summarizer Module
v_{summary}
Program Decoder
v^1_{demo} v^2_{demo} v^k_{demo}
Action Decoder
Perception Decoder
def run() move() ··· else move()
move() turnLeft() ··· pickupMarker()
move() turnLeft() ··· move()
move() turnRight() ··· pickupMarker()
frontIsClear() yes
rightIsClear() no
MarkersPresent() yes
Program

```
def run():
  move()
  if leftIsClear():
    turnLeft()
  REPEAT R=5:
    turnRight()
  if MarkersPresent():
    pickupMarker()
  else:
    move()
```

Figure 3. Model Architecture. The demonstration encoder encodes each of the k demonstrations separately and the summarizer network aggregates them to construct a summary vector. The summary vector is used by the program decoder to produce program tokens sequentially. The encoded demonstrations are used to decode the action sequence and perception conditions as additional supervision.

represented as $a_t = \eta(H_t)$. In this paper, we focus on programs that can be represented in DSL by a code $C = (w_1, w_2, ..., w_N)$, which consists a sequence of tokens w.

A demonstration $\tau = ((s_1, a_1), (s_2, a_2), ..., (s_T, a_T))$ is a sequence of state and action tuples generated by an underlying program η^* given an initial state s_1. Given an initial state s_1 and its corresponding state history H_1, the program generates new action $a_1 = \eta^*(H_1)$. The following state s_2 is generated by a state transition function T: $s_2 \sim T(s_1, a_1)$. The newly sampled state is incorporated into the state history $H_2 = H_1 \frown (s_2)$ and this process is iterated until the end of file action EOF $\in \mathcal{A}$ is returned by the program. A set of demonstrations $D = \{\tau_1, \tau_2, ..., \tau_K\}$ can be generated by running a single program η^* on different initial states $s_1^1, s_1^2, ..., s_1^K$, where each initial state is sampled from an initial state distribution (*i.e.* $s_1^k \sim P_0(s_1)$).

While we are interested in inferring a program η^* from a set of demonstrations D, it is preferable to predict a code C^* instead, because it is a more accessible representation while immediately convertible to a program. Formally, we formulate the problem as a sequence prediction where the input is a set of demonstrations D and the output is a code sequence $\hat{C}$. Note that our objective is not about inferring a code perfectly but instead generating a code that can infer the underlying program, which models the diverse behaviors appearing in the demonstrations in an executable form.

4. Approach

Inferring a program behind a set of demonstrations requires (1) interpreting each demonstration video (2) spotting and summarizing the difference among demonstrations to infer the conditions behind the taken actions (3) describing the understanding of demonstrations in a written language, Based on this intuition, we design a neural architecture composed of three components:

- **Demonstration Encoder** receives a demonstration video as input and produces an embedding that captures an agent's actions and perception.
- **Summarizer Module** discovers and summarizes where actions diverge between demonstrations and upon which branching conditions subsequent actions are taken.
- **Program Decoder** represents the summarized understanding of demonstrations as a code sequence.

The details of the three main components are described in the Section 4.1, and the learning objective of the proposed model is described in Section 4.2. Section 4.3 introduces auxiliary tasks for encouraging the model to learn the knowledge that is essential to infer a program.

4.1. Model Architecture

Figure 3 illustrates the overall architecture of the proposed model, The details of each component are described in the following sections.

4.1.1. Demonstration Encoder

The demonstration encoder receives a demonstration video as input and produces an latent vector that captures the actions and perception of an agent. At each time step, to interpret visual input, we employ a stack of convolutional layers, to encode a state s_t to its embedding as a state vector $v^t_{\text{state}} = \text{CNN}_{\text{enc}}(s_t) \in \mathbb{R}^d$, where $t \in [1, T]$ is the time-step.

Since the demonstration encoder needs to handle demonstrations with variable numbers of frames, we employ an LSTM (Long Short Term Memory) (Hochreiter & Schmidhuber, 1997) to encode each state vector and summarized representation at the same time.

$$c^t_{\text{enc}}, h^t_{\text{enc}} = \text{LSTM}_{\text{enc}}(v^t_{\text{state}}, c^{t-1}_{\text{enc}}, h^{t-1}_{\text{enc}}), \quad (1)$$

where, $t \in [1, T]$ is the time step, while c^t_{enc} and h^t_{enc} denote the cell state and the hidden state. While final state tuples $(c^T_{\text{enc}}, h^T_{\text{enc}})$ encode the overall idea of the demonstration, intermediate hidden states $\{h^1_{\text{enc}}, h^2_{\text{enc}}, ..., h^T_{\text{enc}}\}$ contain high level understanding of each state, which are used as an input to the following modules. Note that these operations are applied to all K demonstrations while the index k is dropped in the equations for simplicity.

4.1.2. Summarizer Module

Inferring an underlying program from demonstrations that exhibits different behaviors requires the ability to discover and summarize where actions diverge between demonstrations and upon which branching conditions subsequent actions are taken. The summarizer module first re-encodes each demonstration with the context of all encoded demonstrations to infer branching conditions. Then, the module aggregates all encoded demonstration vectors to obtain the summarized representation. An illustration of the summarizer is shown in Figure 4.

The first summarization is performed by a reviewer module, an LSTM initialized with the average-pooled final state tuples of the demonstration encoder outputs, which can be written as follows:

$$c^0_{\text{review}} = \frac{1}{K}\sum_{k=1}^{K} c^{T,k}_{\text{enc}}, \quad h^0_{\text{review}} = \frac{1}{K}\sum_{k=1}^{K} h^{T,k}_{\text{enc}}, \qquad (2)$$

where $(c^{T,k}_{\text{enc}}, h^{T,k}_{\text{enc}})$ is the final state tuple of the kth demonstration encoder. Then the reviewer LSTM encodes the hidden states by

$$c^{t,k}_{\text{review}}, h^{t,k}_{\text{review}} = \text{LSTM}_{\text{review}}(h^{t,k}_{\text{enc}}, c^{t-1,k}_{\text{review}}, h^{t-1,k}_{\text{review}}), \qquad (3)$$

where the final hidden state becomes a demonstration vector $v^k_{\text{demo}} = h^{T,k}_{\text{review}} \in \mathbb{R}^d$, which includes the summarized information within a single demonstration.

The final summarization, which is performed across multiple demonstrations, is performed by an aggregation module, which gets K demonstration vectors and aggregates them into a single compact vector representation. To effectively model complex relations between demonstrations, we employ a relational network (RN) module (Santoro et al., 2017). The aggregation process is formally written as follows.

$$v_{\text{summary}} = \text{RN}\left(v^1_{\text{demo}}, ..., v^K_{\text{demo}}\right) = \frac{1}{K^2}\sum_{i,j}^{K} g_\theta(v^i_{\text{demo}}, v^j_{\text{demo}}), \qquad (4)$$

where $v_{\text{summary}} \in \mathbb{R}^d$ is the summarized demonstration vector and g_θ is an MLP parameterized by θ jointly trained with the summarizer module.

We show that employing the summarizer module significantly alleviates the difficulty of handling multiple demonstrations and improve generalization over different number of generations in Section 5 .

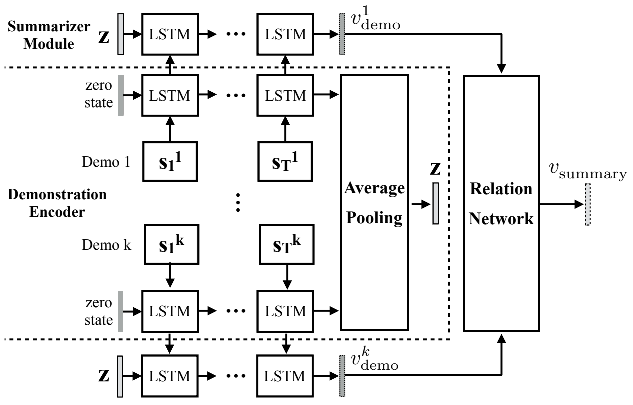


Figure 4. Summarizer Module. The demonstration encoder (inner layer) encodes each demonstration starting from a zero state. The summarizer module (outer layer) aggregates the outputs of the demonstration encoder with a relation network to provide context from other demonstrations.

4.1.3. Program Decoder

The program decoder synthesizes programs from a summarized representation of all the demonstrations. We use LSTMs similar to (Sutskever et al., 2014; Vinyals et al., 2015) as a program decoder. Initialized with the summarized vector v_{summary}, the LSTM at each time step gets the previous token embedding as an input and outputs a probability of the following program tokens as in the Eq. 5. During training, the previous ground truth token is fed as an input, and during inference, the predicted token in the previous steps is fed as an input.

4.2. Learning

The proposed model learns a conditional distribution between a set of demonstrations D and a corresponding code $C = \{w_1, w_2, ..., w_N\}$. By employing the LSTM program decoder, this problem becomes an autoregressive sequence prediction (Sutskever et al., 2014). For a given demonstration and previous code token w_{i-1}, our model is trained to predict the following ground truth token w^*_i, where the cross entropy loss is optimized.

$$\mathcal{L}_{\text{code}} = -\frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} \log p(w^*_{m,n}|W^m_{m,n-1}, D_m), \quad (5)$$

where M is the total number of training examples, $w_{m,n}$ is the nth token of the mth training example and D_m are mth training demonstrations. $W_{m,n} = \{w_{m,1}, ..., w_{m,n}\}$ is the history of previous token inputs at time step n.

4.3. Multi-task Objective

To reason an underlying program from a set of demonstrations, the primary and essential step is recognizing actions and perceptions happening in each step of the demonstration. However, it can be difficult to learn meaningful representations solely from the sequence loss of programs when environments increase in visual complexity. To alleviate this issue, we propose to predict action sequences and perception vectors from the demonstrations as auxiliary tasks. An overview of the auxiliary tasks are illustrated in Figure 3.

Predicting action sequences Given a demo vector v^k_{demo} encoded by the summarizer, an action decoder LSTM produces a sequence of actions. During training, a sequential cross entropy loss similar to Equation 5 is optimized:

$$\mathcal{L}_{\text{action}} = -\frac{1}{MKT}\sum_{m=1}^{M}\sum_{k=1}^{K}\sum_{t=1}^{T}\log p(a^{k*}_{m,t}|A^k_{m,t-1}, v^k_{\text{demo}}), \tag{6}$$

where, $a^k_{m,t}$ is the t-th action token in $k-$th demonstration of m-th training example, $A^k_{m,t} = \{a^k_{m,1}, ..., a^k m, t\}$ is the history of previous actions at time step t.

Predicting perceptions We denote a perception vector $\mathbf{\Phi} = \{\phi_1, ..., \phi_L\} \in \{0, 1\}^L$ as an L dimensional binary vector obtained by executing L perception primitives *e.g.* `frontIsClear()` on a given state s. Specifically, we formulate the perception vector prediction as a sequential multi-label binary classification problem and optimizes the binary cross entropy:

$$\mathcal{L}_{\text{perception}} = \\ -\frac{1}{MKTL}\sum_{m=1}^{M}\sum_{k=1}^{K}\sum_{t=1}^{T}\sum_{l=1}^{L}\log p(\phi^{k*}_{m,t,l}|P^k_{m,t-1}, v^k_{\text{demo}}), \tag{7}$$

where $P^k_{m,t} = \{f(\Phi^k_{m,1}), ..., f(\Phi^k_{m,t})\}$ is the history of encoded previous perception vectors and $f(\cdot)$ is an encoding function.

The aggregated multi-task objective is as follows: $\mathcal{L} = \mathcal{L}_{\text{code}} + \alpha\mathcal{L}_{\text{action}} + \beta\mathcal{L}_{\text{perception}}$, where α and β are hyperparameters controlling the importance of each loss. We set $\alpha = \beta = 1$ to equally optimize the objectives for all the experiments.

5. Experiments

We perform experiments in different environments: Karel (Pattis, 1981) and ViZDoom (Kempka et al., 2016). We first describe the experimental setup and then present the experimental results.

5.1. Evaluation Metric

To verify whether a model is able to infer an underlying program η^* from a given set of demonstrations D, we evaluate accuracy based on the synthesized codes and the underlying program (*sequence accuracy* and *program accuracy*) as well as the execution of the program (*execution accuracy*).

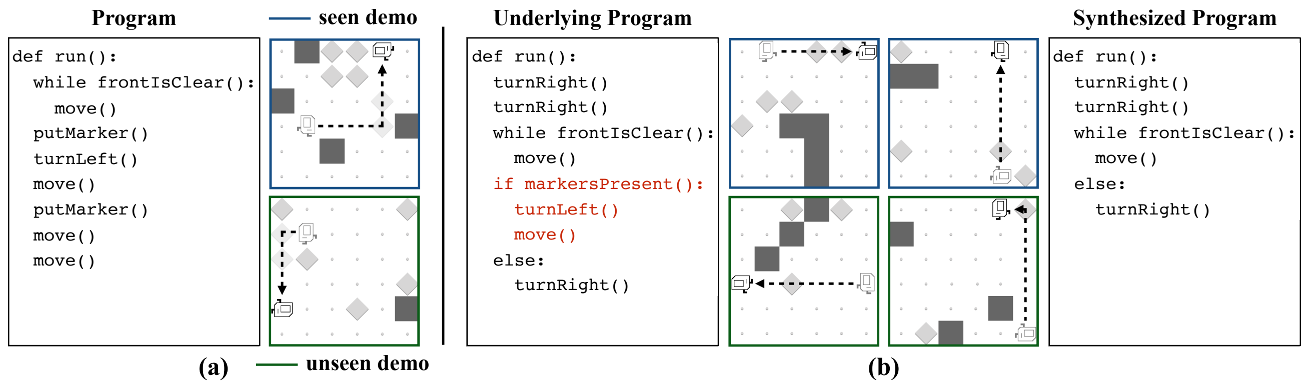
Sequence accuracy Comparison in the code space is based on the instantiated code C^* of a ground truth program and the synthesized code $\hat{C}$ from a program synthesizer. The *sequence accuracy* counts exact match of two code sequences, which is formally written as: $\text{Acc}_{\text{seq}} = \frac{1}{M}\sum_{m=1}^{M}\mathbb{1}_{\text{seq}}(C^*_m, \hat{C}_m)$, where M is the number of testing examples and $\mathbb{1}_{\text{seq}}(\cdot,\cdot)$ is the indicator function of exact sequence match.

Program accuracy While the *sequence accuracy* is simple, it is a pessimistic estimation of *program accuracy* since it does not consider *program aliasing* – different codes with identical program semantics (*e.g.* `repeat(2):(move())` and `move() move()`). Therefore, we measure the *program accuracy* by enumerating variations of codes. Specifically, we exploit the syntax of DSL to identify variations: *e.g.* unfolding repeat statements, decomposing if-else statement into two if statements, etc. Formally, the *program accuracy* is $\text{Acc}_{\text{program}} = \frac{1}{M}\sum_{m=1}^{M}\mathbb{1}_{\text{prog}}(C^*_m, \hat{C}_m)$, where $\mathbb{1}_{\text{prog}}(C^*_m, \hat{C}_m)$ is an indicator function that returns 1 if any variations of $\hat{C}_m$ match any variations of C^*_m. Note that the *program accuracy* is only computable when the DSL is relatively simple and some assumptions are made *i.e.* termination of loops. The details of computing program accuracy are presented in the supplementary material.

Execution accuracy To evaluate how well a synthesized program can capture the behaviors of an underlying program, we compare the execution results of the synthesized program code $\hat{C}$ and the demonstrations D^* generated by a ground truth program η^*, where both are generated from the same set of sampled initial states $I_K = \{s^1_1, ..., s^K_1\}$. We formally define the *execution accuracy* as: $\text{Acc}_{\text{execution}} = \frac{1}{M}\sum_{m=1}^{M}\mathbb{1}_{\text{execution}}(D^*_m, \hat{D}_m)$, where $\mathbb{1}_{\text{execution}}(D^*_m, \hat{D}_m)$ is the indicator function of exact sequence match. Note that when the number of sampled initial states becomes infinitely large, the *execution accuracy* converges to the *program accuracy*.

5.2. Evaluation Setting

For training and evaluation, we collect M_{train} training programs and M_{test} test programs. Each program code C^*_m is randomly sampled from an environment specific DSL and compiled into an executable form η^*_m. The corresponding demonstrations $D^*_m = \{\tau_1, ..., \tau_K\}$ are gener-



```
def run():
  while frontIsClear():
    move()
  putMarker()
  turnLeft()
  move()
  putMarker()
  move()
  move()
```

```
def run():
  turnRight()
  turnRight()
  while frontIsClear():
    move()
  if markersPresent():
    turnLeft()
    move()
  else:
    turnRight()
```

```
def run():
  turnRight()
  turnRight()
  while frontIsClear():
    move()
  else:
    turnRight()
```

Figure 5. Karel Results. Seen training examples are on top row (in blue) and unseen testing examples are on the bottom row (in green). (a) A successful case with a program sequence match (b) Due to a missing branch condition execution in training data (top images), the synthesized program doesn't incorporate the condition, resulting in execution mismatch in lower right testing image.

ated by running the program on $K = K_{\text{seen}} + K_{\text{unseen}}$ different initial states. The seen demonstrations are used as an input to the program synthesizer, and the unseen demonstrations are used for computing execution accuracy. We train our model on the training set $\Omega_{\text{train}} = \{(C_1^*, D_1^*), ..., (C_{M_{\text{train}}}^*, D_{M_{\text{train}}}^*)\}$ and test them on the testing set $\Omega_{\text{test}} = \{(C_1^*, D_1^*), ..., (C_{M_{\text{test}}}^*, D_{M_{\text{test}}}^*)\}$. Note that Ω_{train} and Ω_{test} are disjoint. Both sequence and execution accuracies are used for the evaluation. The training details are described in the supplementary material.

5.3. Baselines

We compare our proposed model (*ours*) against baselines to evaluate the effectiveness of: (1) explicitly modeling the underlying programs (2) our proposed model with the summarizer module and multi-task objective. To address (1), we design a program *induction baseline* based on (Duan et al., 2017), which bypasses synthesizing programs and directly predicts action sequences. We modified the architecture to incorporate multiple demonstrations as well as pixel inputs. The details are presented in the supplementary material. For a fair comparison with our model that gets supervision of perception primitives, we feed the perception primitive vector of every frame as an input to the *induction baseline* . To verify (2), we compose a program *synthesis baseline* simply consisting of a demonstration encoder and a program decoder without a summarizer module and multi-task loss. To integrate all the demonstration encoder outputs across demos, an average pooling layer is applied.

5.4. Karel

We first focus on a visually simple environment to verify the feasibility of program synthesis from demonstrations. We consider Karel (Pattis, 1981) featuring an agent navigating through a gridworld with walls and interacting with markers based on the underlying program.

5.4.1. Environment and Dataset

Karel has 5 action primitives for moving and interacting with markers and 5 perception primitives for detecting obstacles and markers. A gridworld of 8×8 size is used for our experiments. To evaluate the generalization ability of the program synthesizer to novel programs, we randomly generate 35,000 unique programs and split them into a training set with 25,000 program, a validation set with 5,000 program, and a testing set with 5,000 programs. The maximum length of the program codes is 43. For each program, 10 seen demonstrations and 5 unseen demonstrations are generated. The maximum length of the demonstrations is 20.

Methods	Execution	Program	Sequence
Induction baseline	62.8% (69.1%)	-	-
Synthesis baseline	64.1%	42.4%	35.7%
+ summarizer (ours)	68.6%	45.3%	38.3%
+ multi-task loss (ours-full)	**72.1%**	**48.9%**	**41.0%**

Table 1. Performance evaluation on Karel environment. *Synthesis baseline* outperforms *induction baseline* . The summarizer module and the multi-task objective introduce significant improvement.

5.4.2. Performance Evaluation

The evaluation results of our proposed model and baselines are shown in Table. 1. Comparison of execution accuracy shows relative performance of the proposed model and the baselines. *Synthesis baseline* outperforms *induction baseline* based on the execution accuracy, which shows the advantage of explicit modeling the underlying programs. *Induction baseline* often matches some of the K_{unseen} demonstration, but fails to match all of them from a single program. This observation is supported by the number in the parenthesis (69.1%), which counts the number of correct demonstrations while execution accuracy counts the number of program whose demonstrations match perfectly. This finding has also been reported in (Devlin et al., 2017b).

The proposed model shows consistent improvement over *synthesis baseline* for all the evaluation metrics. The sequence accuracy for our full model is 41.0%, which is a reasonable generalization performance given that none of the test programs are seen during training. We observe that our model often synthesizes programs that do not exactly match with the ground truth program but are semantically identical. For example, given a ground truth program `repeat(4):( turnLeft; turnLeft; turnLeft )`, our model predicts `repeat (12): ( turnLeft )`. These cases are considered correct for program accuracy. Note that comparison based on the execution and sequence accuracy is consistent with the program accuracy, which justifies using them as a proxy for the program accuracy when it is not computable.

The qualitative success and failure cases of the proposed model are described in Figure 5. The Figure 5(a) shows a correct case where a single program is used to generate diverse action sequences. Figure 5(b) show a failure case, where part of the ground truth program tokens are not generated due to missing seen demonstration hitting that condition.

Methods	k=3	k=5	k=10
Synthesis baseline	58.5%	60.1%	64.1%
+ summarizer (ours)	**60.6%**	**63.1%**	**68.6%**
Improvement	2.1%	3.0%	4.5%

Table 2. Effect of the summarizer module. Employing the proposed summarizer module brings more improvement as the number of seen demonstration increases over *synthesis baseline* .

5.4.3. Effect of Summarizer

To verify the effectiveness of our proposed summarizer module, we conduct experiments where models are trained on varying numbers of demonstrations and compare the execution accuracy in Table. 2. As the number of demonstrations increases, both models enjoy a performance gain due to extra available information. However, the gap between our proposed model and *synthesis baseline* also grows, which demonstrates the effectiveness of our summarizer module.

5.5. ViZDoom

Doom is a 3D first-person shooter game where a player can move in a continuous space and interact with monsters, items and weapons. We use ViZDoom (Kempka et al., 2016), an open-source Doom-based AI platform, for our experiments. ViZDoom's increased visual complexity and a richer DSL could test the boundary of models in state comprehension, demo summarization, and program synthesis.

5.5.1. Environment and Dataset

The ViZDoom environment has 7 action primitives including diverse motions and attack as well as 6 perception primitives checking the existence of different monsters and whether they are targeted. Each state is represented by an image with $120 \times 160 \times 3$ pixels. For each demonstration, initial state is sampled by randomly spawning different types of monsters and ammos in different location and placing an agent randomly. To ensure that the program behavior results in the same execution, we control the environment to be deterministic.

We generate 80,000 training programs and 8,000 testing programs. To encourage diverse behavior of generated program, we give a higher sampling rate to the perception primitives that has higher entropy over K different initial states. We use 25 seen demonstrations for program synthesis and 10 unseen demonstrations for execution accuracy measure. The maximum length of programs is 32 and the maximum length of demonstrations is 20.

5.5.2. Performance Evaluation

Table. 3 shows the result on ViZDoom environment. *Synthesis baseline* outperforms *induction baseline* in terms of the execution accuracy, which shows the strength of program synthesis for understanding diverse demonstrations. In addition, the proposed summarizer module and the multitask objective bring improvement in terms of all evaluation metrics. Also we found that the syntax of the synthesized programs is about 99.9% accurate. This tells that the program synthesizer correctly learn the syntax of the DSL.

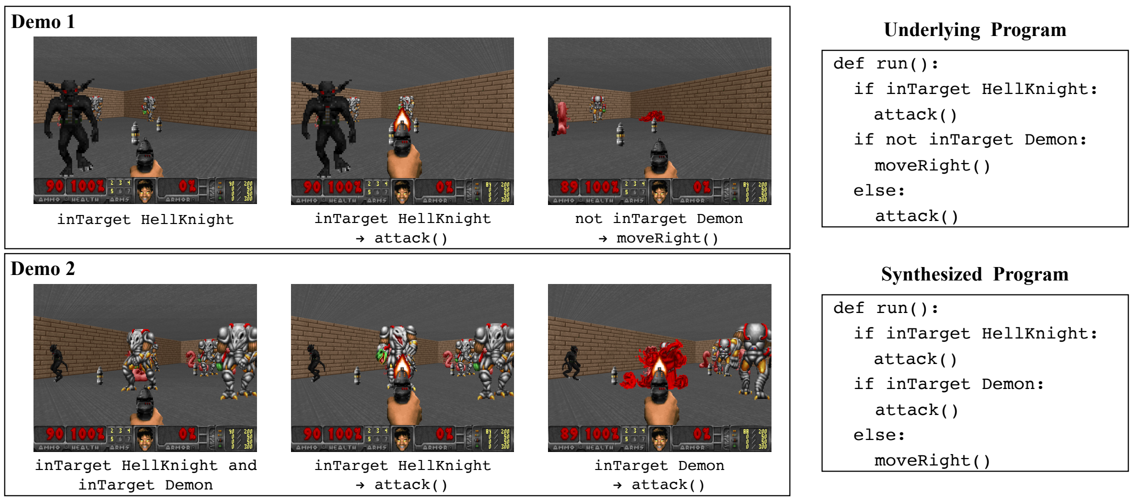
Figure 6 shows the qualitative result. It is shown that the generated program covers different conditional behavior in the demonstration successfully. In the example, the synthesized program does not match the underlying program in the code space, while matching the underlying program in the program space.

Methods	Execution	Program	Sequence
Induction baseline	35.1% (60.6%)	-	-
Synthesis baseline	48.2%	39.9%	33.1%
Ours-full	**78.4%**	**62.5%**	**53.2%**

Table 3. Performance evaluation on ViZDoom environment. The proposed model outperforms *induction baseline* and *synthesis baseline* significantly as the environment is more visually complex.

5.5.3. Analysis

To verify the importance of inferring underlying conditions, we perform evaluation only with programs containing a single if-else statement with two branching consequences. This setting is sufficiently simple to isolate other diverse factors that might affect the evaluation result. For the experiment,


Demo 1
inTarget HellKnight
inTarget HellKnight → attack()
not inTarget Demon → moveRight()
Demo 2
inTarget HellKnight and inTarget Demon
inTarget HellKnight → attack()
inTarget Demon → attack()
Underlying Program
Synthesized Program

Underlying Program

```
def run():
  if inTarget HellKnight:
    attack()
  if not inTarget Demon:
    moveRight()
  else:
    attack()
```

Synthesized Program

```
def run():
  if inTarget HellKnight:
    attack()
  if inTarget Demon:
    attack()
  else:
    moveRight()
```

Figure 6. ViZDoom results. Annotations below frames are the perception conditions and actions. Hellknight, Revenant, and Demon monsters are white, black, and pink respectively. The model is able to correctly percepts the condition and actions as well as synthesize a precise program. Note that the synthesized and the underlying program are semantically identical.

Methods	Execution	Program	Sequence
Induction baseline	26.5% (83.1%)	-	-
Synthesis baseline	59.9%	44.4%	36.1%
Ours-full	**89.4%**	**69.1%**	**58.8%**

Table 4. If-else experiment on ViZDoom environment. Single if-else statement with two branching consequences is used to evaluate ability of inferring underlying conditions.

we use 25 seen demonstrations to understand a behavior and 10 unseen demonstrations for testing. The result is shown in Table. 4. *Induction baseline* has difficulty inferring the underlying condition to match all unseen demonstrations most of the times. In addition, our proposed model outperforms *synthesis baseline* ,2 which demonstrates the effectiveness of the summarizer module and the multi-task objective.

Figure 7 illustrates how models trained with a fixed number (25) of seen demonstration generalize to fewer or more seen demonstrations during testing time. This shows our model and *synthesis baseline* are able to leverage more seen demonstrations to synthesize more accurate programs as well as achieve reasonable performance when fewer demonstrations are given. On the contrary, *Induction baseline* could not exploit more than 10 demonstrations well.

5.5.4. Debugging the Synthesized Program

One of the intriguing properties of the program synthesis is that synthesized programs are interpretable and interactable by human. This makes it possible to debug a synthesized program and fix minor mistakes to correct the behaviors. To verify this idea, we use edit distance between synthesized program and ground truth program as a number of minimum token that is required to get a exactly correct program. With

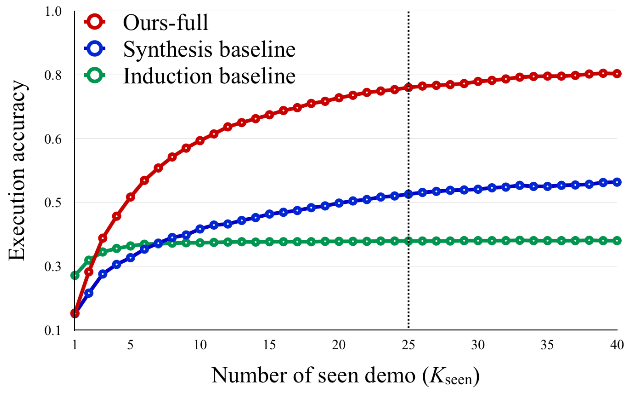


Figure 7. Generation over different number of K_{seen}. The baseline models and our model trained with 25 seen demonstration are evaluated with fewer or more seen demonstrations.

this setting, we found that fixing at most 2 program token provides 4.9% improvement in sequence accuracy and 4.1% improvement in execution accuracy.

6. Conclusion

We propose the task of synthesizing a program from diverse demonstration videos. To address this, we introduce a model augmented with a summarizer module to deal with branching conditions and a multi-task objective to induce meaningful latent representations. Our method is evaluated on a fully observable, third-person environment (Karel environment) and a partially observable, egocentric game (ViZDoom environment). The experiments demonstrate that the proposed model is able to reliably infer underlying programs and achieve satisfactory performances.

References

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

Bunel, Rudy R, Hausknecht, Matthew, Devlin, Jacob, Singh, Rishabh, and Kohli, Pushmeet. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*, 2018.

Cai, Jonathon, Shin, Richard, and Song, Dawn. Making neural programming architectures generalize via recursion. In *ICLR*, 2017.

Devlin, Jacob, Bunel, Rudy R, Singh, Rishabh, Hausknecht, Matthew, and Kohli, Pushmeet. Neural program meta-induction. In *NIPS*, 2017a.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. In *ICML*, 2017b.

Duan, Yan, Andrychowicz, Marcin, Stadie, Bradly, Ho, OpenAI Jonathan, Schneider, Jonas, Sutskever, Ilya, Abbeel, Pieter, and Zaremba, Wojciech. One-shot imitation learning. In *NIPS*, 2017.

Finn, Chelsea, Yu, Tianhe, Zhang, Tianhao, Abbeel, Pieter, and Levine, Sergey. One-shot visual imitation learning via meta-learning. In *CoRL*, 2017.

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

Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *NIPS*, 2015.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 1997.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, 2015.

Kaiser, Łukasz and Sutskever, Ilya. Neural gpus learn algorithms. In *ICLR*, 2016.

Kempka, Michał, Wydmuch, Marek, Runc, Grzegorz, Toczek, Jakub, and Jaśkowski, Wojciech. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *CIG*, 2016.

Neelakantan, Arvind, Le, Quoc V, and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. In *ICLR*, 2015.

Ng, Andrew Y, Russell, Stuart J, et al. Algorithms for inverse reinforcement learning. In *ICML*, 2000.

Parisotto, Emilio, Mohamed, Abdel-rahman, Singh, Rishabh, Li, Lihong, Zhou, Dengyong, and Kohli, Pushmeet. Neuro-symbolic program synthesis. In *ICLR*, 2017.

Pattis, Richard E. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

Pomerleau, Dean A. Alvinn: An autonomous land vehicle in a neural network. In *NIPS*, 1989.

Pomerleau, Dean A. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 1991.

Reed, Scott and De Freitas, Nando. Neural programmer-interpreters. In *ICLR*, 2016.

Ross, Stéphane, Gordon, Geoffrey, and Bagnell, Drew. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 2011.

Santoro, Adam, Raposo, David, Barrett, David G, Malinowski, Mateusz, Pascanu, Razvan, Battaglia, Peter, and Lillicrap, Tim. A simple neural network module for relational reasoning. In *NIPS*, 2017.

Simonyan, Karen and Zisserman, Andrew. Two-stream convolutional networks for action recognition in videos. In *NIPS*, 2014.

Srivastava, Nitish, Mansimov, Elman, and Salakhudinov, Ruslan. Unsupervised learning of video representations using lstms. In *ICML*, 2015.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

Venugopalan, Subhashini, Rohrbach, Marcus, Donahue, Jeffrey, Mooney, Raymond, Darrell, Trevor, and Saenko, Kate. Sequence to sequence-video to text. In *ICCV*, 2015.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. In *CVPR*, 2015.

Xu, Danfei, Nair, Suraj, Zhu, Yuke, Gao, Julian, Garg, Animesh, Fei-Fei, Li, and Savarese, Silvio. Neural task programming: Learning to generalize across hierarchical tasks. In *ICRA*, 2018.

Zaremba, Wojciech and Sutskever, Ilya. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.