VirtualHome: Simulating Household Activities via Programs

Xavier Puig[1]*, Kevin Ra[2]*, Marko Boben[3]*, Jiaman Li[4], Tingwu Wang[4],
Sanja Fidler[4], Antonio Torralba[1]

[1]MIT   [2]McGill University   [3]University of Ljubljana   [4]University of Toronto

{xavierpuig,torralba}@csail.mit.edu   kevin.ra@mail.mcgill.ca   marko.boben@fri.uni-lj.si

{tingwuwang,ljm,fidler}@cs.toronto.edu

Abstract

In this paper, we are interested in modeling complex activities that occur in a typical household. We propose to use programs, *i.e., sequences of atomic actions and interactions, as a high level representation of complex tasks. Programs are interesting because they provide a non-ambiguous representation of a task, and allow agents to execute them. However, nowadays, there is no database providing this type of information. Towards this goal, we first crowd-source programs for a variety of activities that happen in people's homes, via a game-like interface used for teaching kids how to code. Using the collected dataset, we show how we can learn to extract programs directly from natural language descriptions or from videos. We then implement the most common atomic (inter)actions in the Unity3D game engine, and use our programs to "drive" an artificial agent to execute tasks in a simulated household environment. Our* VirtualHome *simulator allows us to create a large activity video dataset with rich ground-truth, enabling training and testing of video understanding models. We further showcase examples of our agent performing tasks in our* VirtualHome *based on language descriptions.*

1. Introduction

Autonomous agents need to know the sequences of actions that need to be performed in order to achieve certain goals. For example, we might want a robot to clean our room, make the bed, or cook dinner. One can define activities with procedural recipes or programs that describe how one can accomplish the task. A *program* contains a sequence of simple symbolic instructions, each referencing an atomic action (e.g. "sit") or interaction (e.g. "pick-up object") and a number of objects that the action refers to (e.g., "pick-up juice"). Assuming that an agent knows how to execute the atomic actions, programs provide an effective means of "driving" a robot to perform different, more complex tasks. Programs
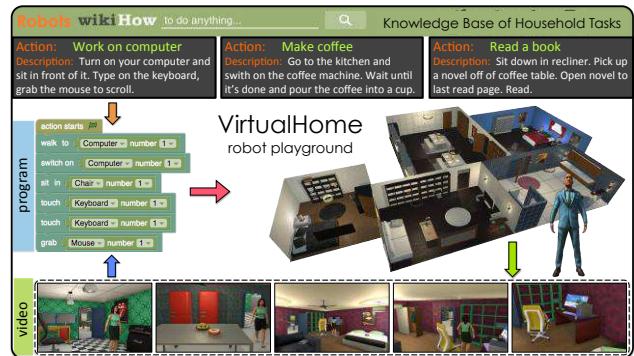

Figure 1: We first crowdsource a large knowledge base of household tasks, **(top)**. Each task has a high level name, and a natural language instruction. We then collect "programs" for these tasks, **(middle left)**, where the annotators "translate" the instruction into simple code. We implement the most frequent (inter)actions in a 3D simulator, called *VirtualHouse*, allowing us to drive an agent to execute tasks defined by programs. We propose methods to generate programs automatically from text **(top)** and video **(bottom)**, thus driving an agent via language and a video demonstration.

can also be used as an internal representation of an activity shown in a video or described by a human (or another agent). Our goal in this paper is to automatically generate programs from natural language descriptions, as well as from video demonstrations, potentially allowing naive users to teach their robot a wide variety of novel tasks.

Towards this goal, one important missing piece is the lack of a database describing activities composed of multiple steps. We first crowdsource common-sense information about typical activities that happen in people's homes, forming the natural language know-how of how these activities are performed. We then adapt the Scratch [1] interface used for teaching kids how to code in order to collect programs that formalize the activity as described in the knowledge base. Note that these programs include *all the steps* required for the robot to accomplish a task, even those that are not mentioned in the language descriptions. We then implement the most common atomic (inter)actions in the Unity3D game engine, such as *pick-up*, *switch on/off*, *sit*, *stand-up*. By exploiting the physics, navigation and kinematic models in the

*Denotes equal contribution

game engine we enable an artificial agent to execute these programs in a simulated household environment.

We first introduce our data collection effort and the program based representation of activities. In Sec. 5 we show how we can learn to automatically translate natural language instructions of activities into programs. In Sec. 4 we introduce the *VirtualHome* simulator that allows us to create a large activity video dataset with rich ground-truth by using programs to drive an agent in a synthetic world. Finally, we use the synthetic videos to train a system to translate videos of activities into the program being executed by the agent. Our *VirtualHome* opens an important "playground" for both vision and robotics, allowing agents to exploit language and visual demonstration to execute novel activities in a simulated environment. Our data is available online: http://virtual-home.org/.

2. Related Work

Actions as programs. A few works have defined activities as programs. In [26], the authors detect objects and actions in cooking videos and generate an "action plan" using a probabilistic grammar. By generating the plan, the robots were able to execute complex actions by simply watching videos. These authors further collected a tree bank of action plans from annotated cooking videos [25], creating a knowledge base of actions as programs for cooking. [16] tried to translate cooking recipes into action plans using an MRF. [20, 3] also argued for actions as a sequence of atomic steps. They aligned YouTube how-to videos with their narrations in order to parse videos into such programs. Most of these works were limited to either a small set of activities, or to a narrow domain (cooking). We go beyond this by creating a knowledge base about an exhaustive set of activities and tasks that people do in their homes.

Just recently, [22] crowd-sourced scripts of people's actions at home in the form of natural language. These were mostly comprised of one or two sentences describing a short sequence of actions. While this is valuable information, language is very versatile and thus hard to convert into a usable program on a robot. We show how to do this in our work.

Code generation. There is increased interest in generating and interpreting source code [11]. Work most relevant to ours produces code given natural language inputs. [4] retrieves code snippets from *Stackoverflow* based on language queries. Given a sentence describing conditions, [17] produces If-This-Then-That code. [12] generates a program specifying the logic of a card game given a short description of the rules. In [8], the authors inferred programs to answer visual questions about images. Our work differs in the domain, and works with text or video as input.

Robotics. A subfield of robotics aims at teaching robots to follow instructions provided in natural language by a human tutor. However, most of the existing literature deals

with a constrained problem, for example, they learn to translate navigational instructions into a sequence of robotic actions [24, 13, 10, 14]. These instructions are typically simpler as they directly mention what to do next, and the action space is small. This is not the case in our work which also considers interactions with objects, and everyday activities which are typically far more complex.

Simulation. Simulations using game engines have recently been developed to facilitate training visual models for autonomous driving [7, 19, 6], quadcopter flying [21], or other robotic tasks [5]. Recently, [29] released a simulator for target-driven indoor navigation, which however does not simulate activities. We are not aware of simulators at the scale of objects and actions in a home, like ours. Lastly, we give credit to the popular game Sims which we draw our inspiration from. Sims is a strategic video game mimicking daily household activities. Unfortunately, the source of the game is not public and thus cannot be used for our purpose.

3. KB of Household Activities for Robots

Our goal is to build a large repository of common activities and tasks that we perform in our households in our daily lives. These tasks can include simple actions like "turning on TV" or complex ones such as "make coffee with milk". What makes our effort unique is that we are interested in collecting this information for robots. Unlike humans, robots need more direct instructions. For example, in order to "watch tv", one might describe it (to a human) as "Switch on the television, and watch it from the sofa". Here, the actions "grab remote control" and "sit/lie on sofa" have been omitted, since they are part of the commonsense knowledge that humans have. In our work, we aim to collect **all the steps** required for a robot to successfully execute a task, including the commonsense steps. In particular, we want to collect programs that fully describe activities.

Describing actions as programs has the advantage that it provides a clear and non-ambiguous description of all the steps needed to complete a task. Such programs can then be used to instruct a robot or a virtual character. Programs can also be used as a representation of a complex task that involves a number of simpler actions, providing a way to understand and compare activities and goals.

3.1. Data Collection

In this section, we describe our dataset collection using crowdsourcing. Describing actions as programs can be a challenging task as most annotators have no programing experience. We split the data collection effort in two parts. In the first part, we ask AMT workers to provide verbal descriptions of daily household activities. In particular, each worker is asked to come up with a common activity/task, give it a high level name, eg "make coffee", and describe it in detail. In order to cover a wide spectrum of activities we

Description: Bring some juice to the coffee table, and relax by watching television from the sofa in the living room.

0:00 [Walk] <JUICE> 0:10 [Grab] <JUICE> 0:12 [Walk] <COFFEE_TABLE> 0:25 [Put] <JUICE> <C.TABLE> 0:27 [Walk] <TV> 0:36 [Switch-On] <TV> 0:38 [Walk] <Sofa> 0:42 [Sit] <Sofa> 0:50

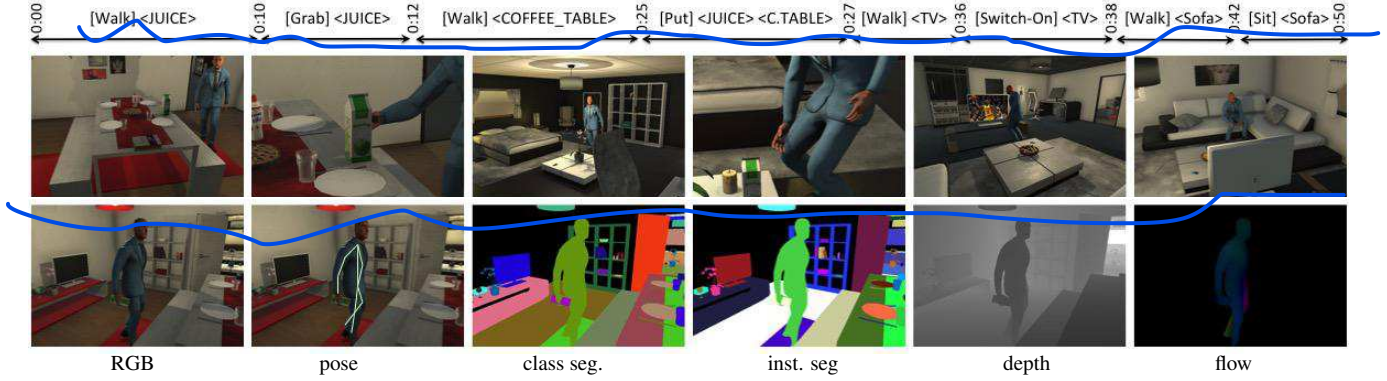| RGB | pose | class seg. | inst. seg | depth | flow |

Figure 2: **VirtualHome Activity Dataset** is a video dataset of composite activities created with our simulator. We start by generating programs using a simple probabilistic grammar. We animate each program in VirtualHome by randomizing the selection of homes, agents, cameras, as well as the placement of a subset of the objects, the initial location of the agent, the speed of the actions, and choice of objects for interactions. Each program is shown to an annotator who is asked to describe it in natural language (**top row**). Videos have ground-truth: (**second row**) time-stamp for each atomic action, (**bottom**) 2D and 3D pose, class and object instance segmentation, depth and optical flow.

pre-specified in which scene the activity should start. Scenes were selected randomly from a list of 8 scenes (*living room*, *kitchen*, *dining room*, *bedroom*, *kids bedroom*, *bathroom*, *entrance hall*, and *home office*). An example of a described activity is shown in Fig. 3. Note that these descriptions may likely omit the commonsense steps, as they were written by "naive" workers that were describing these activities as they would to a (human) friend.

In the second stage, we showed the collected descriptions to the AMT workers and asked them to translate these descriptions into programs using a graphical programing language. We told them to produce a program that will "drive" a robot to successfully accomplish the described activity. Our interface builds on top of MIT's Scratch project [1] designed to teach young children to write symbolic code. We found that workers were capable of quickly learning to produce useful programs by providing them with a carefully designed tutorial. Fig. 3 shows a snapshot of the programing interface. Finally, we asked more qualified workers hired via Upwork crowdsourcing platform to double check the collected data.

Workers had to compose a program by composing a sequence of steps. Each instruction is a Scratch block from a predefined list of 77 possible blocks compiled by analyzing the frequency of verbs in the collected descriptions. Each step in the program is defined by a block. A block defines a syntactic frame with an action and a list of arguments (e.g., the block *walk* requires one argument to specify the destination, Fig. 3.c). To simplify the search for blocks they are organized according to 9 broad action categories (Fig. 3.b).

We required that the program contains all the steps, even those not explicitly mentioned in the description, but that could be inferred from common-sense. Fig. 3.d shows an example of a program. We also allowed annotators to use a "special" block for missing actions, where the step can be written as free-form text. Programs using this special block

will not be used in the rest of the paper, but allowed us in identifying new blocks that needed to be added.

More precisely, step t in the program can be written as:

$$\text{step}_t = [action_t] \ \langle object_{t,1}\rangle(id_{t,1}) \ ... \ \langle object_{t,n}\rangle(id_{t,n})$$

Here, id is an unique identifier (counter) of an object and helps in disambiguating different instances of objects that belong to the same class. An example of a program for "watch tv" would be:

$$\text{step}_1 = [\text{Walk}] \ \langle\text{TELEVISION}\rangle(1)$$
$$\text{step}_2 = [\text{SwitchOn}] \ \langle\text{TELEVISION}\rangle(1)$$
$$\text{step}_3 = [\text{Walk}] \ \langle\text{SOFA}\rangle(1)$$
$$\text{step}_4 = [\text{Sit}] \ \langle\text{SOFA}\rangle(1)$$
$$\text{step}_5 = [\text{Watch}] \ \langle\text{TELEVISION}\rangle(1)$$

Here, the programs defines that the television in steps 1, 2 and 5 refer to the same object instance.

3.2. Dataset Analysis

In the first part we collected 1814 descriptions. From those, we were able to collect programs for 1703 descriptions. Some of the programs contained several "special blocks" for missing actions, which we remove, resulting in 1257 programs. We finally selected a set of tasks and asked workers to write programs for them, obtaining 1564 additional programs. The resulting 2821 programs form our *ActivityPrograms* dataset. On average, the collected descriptions have 3.2 sentences and 21.9 words, and the resulting programs have 11.6 steps on average. The dataset statistics are summarized in Table 1.a.

The dataset covers 75 atomic actions and 308 objects, making 2709 unique steps. Fig. 4.a shows a histogram of the 50 most common actions appearing in the dataset, and, Fig. 4.b, the 50 most common objects.

Our dataset contains activities with several examples, and we analyze their diversity by comparing their programs. Table 1.b analyzes 4 selected activities. We compute their
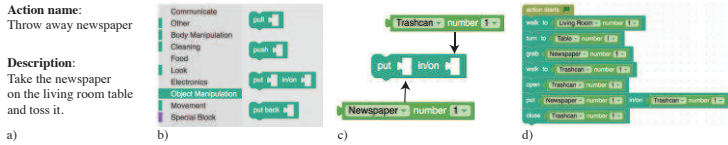
Figure 3: a) Description provided by a worker. b) User interface showing the list of block categories and 4 example blocks, c) Example of composition of a block by adding the arguments. Each block is like a Lego piece where the user can drop arguments inside and attach one block to another. d) Final program corresponding to the description from (a).

Figure 4: a) Counts of actions in our *ActivityPrograms* dataset, b) object counts (zoom to read)

similarities as the average length of the longest common subsequences computed between all pairs of programs.

We can also measure distances between activities by measuring the distance between programs. The similarity between two programs is measured as the length of their longest common subsequence of instructions divided by the length of the longest program. Table 1.c. shows the similarity matrix (sorted to better show the block diagonal structure) between different activities in our dataset.

Completeness of programs. We analyze whether the collected programs contain all the necessary steps to execute the given task. We sample 100 collected programs and ask 5 AMT workers to rate whether the program is complete, or is missing minor steps (sitting in a chair before walking towards it) or important steps (filling a glass before drinking). Results show that 64% of the programs are complete, 28% are missing minor steps and 8% are missing crucial steps.

4. *VirtualHome*: Simulator of Household Tasks

The main motivation behind using programs to represent activities is to "drive" robots to perform tasks by having them executing these programs. As a proxy, we here use programs to drive characters in a simulated 3D environment. Simulations are useful as they define a playground for "robots", an environment where artificial agents can be taught to perform tasks. Here, we focus on building the simulator, and leave learning inside the simulator to future work. In particular, we will assume the agent has access to all 3D and semantic information about the environment, as well as to manually defined animations. Our focus will be to show that programs represent a good way of instructing such agents. Furthermore, our simulator will allow us to generate a large-scale video dataset of complex activities that is rich and diverse. We can create such a dataset by simply recording the agent executing programs in the simulator. The simulator then provides us with dense ground-truth information, eg semantic segmentation, depth, pose, etc. Fig. 2 showcases this dataset.

We implemented our *VirtualHome* simulator using the Unity3D game engine which allows us to exploit its kinematic, physics and navigation models, as well as user-contributed 3D models available through Unity's Assets store. We obtained six furnished homes and 4 rigged humanoid models from the web. On average, each home con-
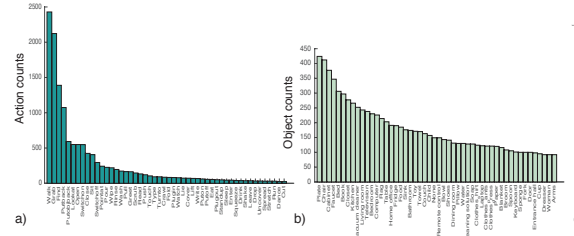
tains 357 object instances (86 per room). We collected objects from additional 30 object classes that appear in our collected programs yet are not available in the package, via the 3D warehouse [1]. To ensure visual diversity, we collected at least 3 different models per class. The apartments and agents are shown in Fig. 5 and Fig. 6.

4.1. Animating Programs in VirtualHome

Every step in the program requires us to animate the corresponding (inter)action in our virtual environment. We thus need to both, determine which object in the home (which we refer to as the *game object*) the step requires as well as properly animating the action. To get the former we need to solve an optimization problem by taking into account all steps in the program and finding a feasible path. For example, if the program requires the agent to switch on a computer and type on a keyboard, ideally the agent would type on the keyboard next to the chosen computer and not navigate to another keyboard attached to a different computer in possibly a different room. We now describe our simulator in more detail.

Animating atomic actions. There is a huge variety and number of atomic actions that appear in the collected programs, as can be seen in Fig. 4. We implemented the 12 most frequent ones: *walk/run*, *grab*, *switch-on/off*, *open/close*, *place*, *look-at*, *sit/standup*, *touch*. Note that there is a large variability in how an action is performed depending on to which object it is applied to (e.g., opening a fridge is different than opening a drawer). We use Unity's NavMesh framework for navigation (path planner to avoid obstacles). For each action we compute the agent's target pose and animate the action using RootMotion FinalIK inverse kinematics package. We further animate certain objects the agent interacts with, e.g., we shake a coffee maker, animate toast in a toaster, show a (random) photo on a computer or TV screen, light up a burner on a stove, and light up the lamps in the room, when these objects are switched on by the agent.

Preparing the Scene. While every 3D home already contains many objects, the programs may still mention objects that are not present in the scene. To deal with this, we first "set" the scene by placing all missing objects that a program refers to in the home, before we try to execute the program. To be able to prepare a scene in a plausible way, we collect a
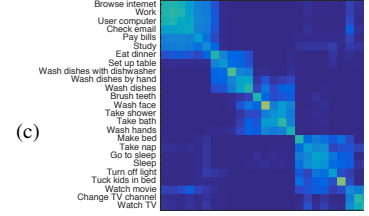
---

[1] https://3dwarehouse.sketchup.com

Dataset	# prog.	avg # steps	avg # sent.	avg # words
ActivityProg.	2821	11.6	3.2	21.9
SyntheticProg.	5193	9.6	3.4	20.5

(a)

Action	# Prog.	LCS	Norm. LCS
Make coffee	69	4.56	0.26
Fold laundry	11	1.29	0.08
Watch TV	128	3.65	0.40
Clean	42	0.76	0.04

(b)

(c)

Table 1: **(a)** *ActivityPrograms*: Analyzing diversity in the same activity, by computing similarities across all pairs of the collected programs. "LCS" denotes longest common subsequence. For "norm.LCS", we normalize by max length of the two programs. **(b)** We analyze programs and natural language descriptions for both, real activities in *ActivityPrograms* (Sec. 3), and synthesized programs (with real descr.). **(c)** shows the similarity matrix (sorted to better show the block diagonal structure) between different activities in our dataset.

Figure 5: 3D households in our VirtualHome. Notice the diversity in room and object layout and appearance. Each home has on average 357 objects. First 4 scenes are used for training, the fifth is also used in val, and all scenes are used when testing our video-to-script model.

male 1   female 1   male 2   female 2

Figure 6: Agents in VirtualHome. We use *male 1* and *female 1* in train., and all agents when testing our video-to-program model.

knowledge base of possible object locations. The annotator is shown the class name and selects a list of other objects (including *floor*, *wall*) that are likely to support it.

Executing a Program. To animate a program we need first to create a mapping between the objects in the program and the corresponding instances inside the virtual simulator. Furthermore, for each step in the program, we also need to compute the interaction position of the agent with respect to an object, and any additional information needed to animate the action (e.g., which hand to use, speed). We build a tree of all possibilities of assigning game objects to objects in the program, along with all interaction positions and attributes. To traverse the tree of possible states we use backtracking and stop as soon as a state executing the last step is found. Since the number of possible object mappings for each step is small, and we can prune the number of interaction positions to a few, our optimization runs in a few seconds, on average.

Animation. We place 6-9 static cameras in each room, 26 per home on average. During recording, we switch between cameras based on agent's visibility. In particular, we randomly select a camera which sees the agent, and keep it until the agent is visible and within allowed distance. For agent-object interaction we also try to select a camera and adjust its field of view to enhance the visibility of the interaction. We further randomize the position, angle and field of view of each camera. Randomization is important when creating a dataset to ensure diversity of the final video data.

VirtualHome Activity dataset. Since the programs in *ActivityPrograms* represent real activities that happen in households, they contain significant variability in actions and objects that appear in steps. While our ultimate aim is

to be able to animate all these actions in our simulator, our current efforts only support the top 12 most frequent actions. We thus create another dataset that contains programs containing only these actions in their steps. The creation of this dataset is explained below.

We synthesized 5,193 programs using a simple probabilistic grammar, and had each one described in natural language by a human annotator. Although these programs were not given by annotators, they produced reasonable activities, creating a much larger dataset of paired descriptions-programs at a fraction of the cost. We then animated each program in our simulator, and automatically generated ground-truth which allows us to train and evaluate our video models. As can be seen from Table 1, descriptions in *VirtualHome Activity* dataset are of comparable length. However, the vocabulary here was biased towards that used in programs.

We animate the programs as described above, by randomizing the selection of home, an agent, cameras, placement of a subset of objects, initial location of the agent, speed of the actions, and choice of objects for interactions. We build on top of [2] to automatically generate groundtruth: 1) time-stamp of each step to video, 2) agent's 2D/3D pose, 3) class and instance segmentation, 4) depth, 5) optical flow, 6) camera parameters. Example of data is shown in Fig. 2.

5. From Videos and Descriptions to Programs

We introduce a novel task using our dataset. In particular, we aim to generate a program for the activity from either a natural language description or from a video demonstration. We treat the task of transcribing an input (description or video) into a program as a translation problem. We adapt the
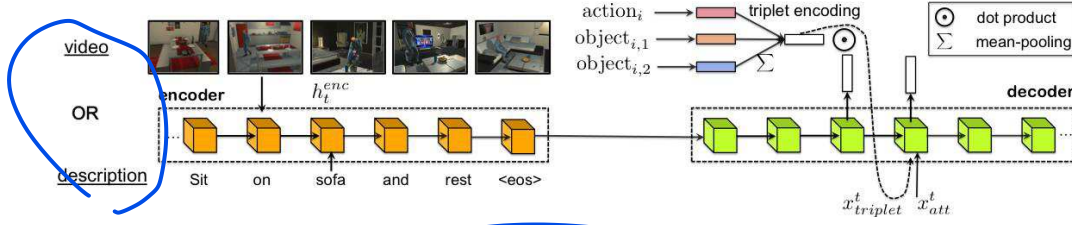
Figure 7: Our encoder-decoder LSTM for generating programs from natural language descriptions or videos.

seq2seq model [23] for our task, and train it with Reinforcement Learning that exploits the reward from the simulator.

Our model consists of an RNN encoder that encodes the input sequence into a hidden vector representation, and another RNN acting as a decoder, generating one step of the program at a time. We use LSTM with 100-dim hidden states as our encoder. At each step t, our RNN decoder decodes a step which takes the form of eq. (1). Let $\mathbf{x}_t$ denote an input vector to our RNN decoder at step t, and let h^t be the hidden state. Here, h^t is computed as in the standard LSTM using $\tanh$ as the non-linearity. Let a_i be a one-hot encoding of an action i, and o_i a one-hot encoding of an object. We compute the probability p_i^t of an instruction i at step t as:

$$\tilde{a}_i = W_a a_i, \quad \tilde{o}_{i,n} = W_o o_{i,n}, \quad v_i = \text{mean}(\tilde{a}_i, \tilde{o}_{i,1}, ..., \tilde{o}_{i,n})$$

$$p_i^t = \text{softmax}_i\left(\frac{v_i}{\|v_i\|}^T \cdot W_v(h^t \| \mathbf{x}_t^{att})\right) \quad (1)$$

where W_a and W_o and W_v are learnable matrices, and v_i denotes an embedding of an instruction. Note that here, n is either 1 or 2 (our instructions have at most two objects).

The input vector $\mathbf{x}_t$ concatenates multiple features. In particular, we use the embedding v of the step with the highest probability from the previous time instance of the decoder. Following [23], we further use the attention mechanism over the encoder's states to get another feature $\mathbf{x}_t^{att}$. In particular:

$$\alpha_j^t = \text{softmax}_j\left(v^T\left(W_{att}\left(h^t \| h_{enc}^j\right)\right)\right) \quad (2)$$

$$\mathbf{x}_t^{att} = \sum_j \alpha_j^t h_{enc}^j \quad (3)$$

where W_{att}, v are learnable parameters. Our full model is visualized in Fig. 7.

Learning and inference. Our goal is to generate programs that are both close to the ground-truth programs in terms of their LCS (longest common subsequence) and are also executable by our renderer. To that end, we train our model in two phases. Firstly, we pre-train the model using cross-entropy loss at each time step of the RNN decoder. We follow the typical training strategy where we make a prediction at each time instance but feed in the ground-truth step to the next time instance. We use the word2vec [15] embeddings for matrices W_a and W_o.

In the second stage, we treat program generation as an Reinforcement Learning problem, where the agent is learning a policy that generates steps to compose a program.

We follow [18], and use policy gradient optimization to train the model, using the greedy policy as the baseline estimator. We exploit two different kinds of reward $r(w^s, g)$ for RL training, where w^s denotes the sampled program, and g the ground-truth program. To ensure that the generated program is semantically correct (follows the description/video), we use the normalized LCS metric (length of the longest common subsequence) between the two programs as our first reward $r_{LCS}(w^s, g)$. The second reward comes from our simulator, and measures whether the generated program is executable or not. This reward, $r_{sim}(w^s)$, is a simple binary value. We carefully balance the total reward as, $r(w^s, g) = r_{LCS}(w^s, g) + 0.1 \cdot r_{sim}(w^s)$.

So far we did not specify the input to the RNN encoder. Our model accepts either a language description or a video.

Textual Description. To encode a textual description our RNN encoder gets as input the word2vec [15] embedding of the word in the description at each time instance.

Video. To generate programs from videos, we partition each video into 2-second clips and train a model to predict the step at the middle frame. We use DilatedNet to obtain the semantic segmentation of each frame and use the Temporal Relation Network [27] with 4-frame relations to predict the embedding of an instruction (action+object+subject). We use this embedding to obtain the likelihood of each instruction. The prediction at each clip is used as input to the RNN encoder for program generation.

6. Experiments

In our experiments we exploit both of our datasets: *ActivityPrograms* containing descriptions and programs for real activities, and *VirtualHome Activity dataset* that contains synthesized programs, yet natural descriptions to describe them. *VirtualHome Activity dataset* further contains videos animating the programs.

6.1. Step (Instruction) Classification from Video

We first evaluate our model for the task of video-based action and action-object-subject (step/instruction in the program) classification. Here, we partition each video in 2-sec clips, and use the clip-based TRN to perform classification. We compute performance as the mean per-class accuracy

	Action	Objects	Steps	Mean
Rand. Retrieval	8.30%	1.50%	0.51%	3.43%
Seen homes	70.32 %	42.14 %	23.81 %	45.42%
Unseen homes	31.34%	14.55%	11.48%	19.12%
All	46.85%	25.76%	18.41%	30.34%

	Action	Objects	Steps	Mean	Simulator
Rand. Retrieval	.473	.079	.071	.207	100.0%
MLE	.735	.359	.341	.478	19.4%
PG(LCS)	.761	.383	.364	.502	19.0%
PG(LCS+Sim)	.751	.377	.358	.495	22.4%
PG(LCS+Sim) Seen homes	.851	.556	.528	.645	24.6%
PG(LCS+Sim) Unseen homes	.680	.250	.236	.389	20.9%

Table 2: **Left:** Accuracy of *video-based action classification* and *action-subject-object* (step or instruction in the program) prediction in **2-sec clips** from our VirtualHome Activity dataset. **Right**: *Video-based program generation*.

Method	Action	Objects	Steps	Mean	Simulator (%)
Rand. Sampling	.226	.039	.020	.095	0.6%
Rand. Retrieval	.473	.079	.071	.207	100.0%
Skipthoughts	.642	.272	.252	.389	100.0%
MLE	.777	.723	.686	.729	38.6%
PG(LCS)	.803	.766	.732	.767	35.5%
PG(LCS+Sim)	.806	.775	.740	.774	39.8%

Method	Action	Objects	Steps	Mean
Rand. Sampling	.106	.018	.004	.043
Rand. Retrieval	.320	.037	.032	.130
Skipthoughts	.469	.297	.266	.344
MLE	.497	.392	.340	.410
PG(LCS)	.522	.433	.387	.447

Table 3: **Programs from descr.**: Accuracy on (**left**) *VirtualHome Act.*, and (**right**) *ActivityPrograms*. We compute the length of longest common subsequence between a predicted script and GT and divide by max length of the two programs, mimicking IoU for programs. Since real programs are mainly not executable in our simulator due to the lack of implemented actions, we cannot report the executability metric.

across all 2-sec clips in *test*. To better understand the generalization properties of the video-based models, we further divide the *test* set into videos recorded in *homes seen* at train time, and videos in *homes not seen* at train time. We report the results in Table 2 (left). To set the lower bound, we also report a simple *random retrieval* baseline, in which a program is randomly retrieved from the training set. We can see that our model performs significantly better. However, a large number of actions and objects of interest, makes the prediction task challenging for the model.

6.2. Program Generation

We now evaluate the task of program generation.

Metrics. We evaluate program induction using a measure similar to IOU. We compute the longest common subsequence between a GT and a predicted program, where we allow gaps between the matching steps, but require their order to be correct. We obtain accuracy as the length of the subseq. divided by the max of the two programs' lengths. We also compute accuracies for *actions* and *objects* alone. Since LCS does not measure whether the program is valid, we report another metrics that computes the percentage of generated programs that are executable in our simulator.

Language-based prediction. Since we have descriptions for all activities, we first evaluate how well our model translates natural language descriptions into programs. We report results on *ActivityPrograms* (real activities), as well as on *VirtualHome Activity* datasets (where we first only consider descriptions, not videos). We compare our models to four baselines: 1) *random sampling*, where we randomly pick both an action for each step and its arguments, 2) *random retrieval*, where we randomly pick a program from the training set, 3) *skipthoughts*, where we embed the description using [9, 28], retrieve the closest description from training set and take its program, 4) our model trained with

MLE (no RL). Table 3 provides the results. We can see that our model outperforms all baselines on both datasets. Our RL model that exploits LCS reward outperforms the MLE model on both metrics (LCS and executability). Our model that uses both rewards slightly decreases the LCS score, but significantly improves the executability metrics.

Video-based prediction. We also report results on the most challenging task of video-based program generation. The results are shown in Table 2 (right). One can observe that RL training with LCS reward improves the overall accuracy over the MLE model (the generated programs are more meaningful given the description/video), however its executability score decreases. This is expected: MLE model typically generates shorter programs, which are thus more likely to be executable (an empty program is always executable). A careful balance of both metrics is necessary. RL with both the LCS and the simulator reward improves both LCS and the executability metrics over the LCS-only model.

Executing programs in VirtualHome. In Fig. 8 we show a few examples of our agent executing programs generated from natural descriptions. To understand the quality of our simulator as well as the plausibility of our program evaluation metrics, we perform a human study. We randomly selected 10 examples per level of performance: (a) $[0.95-1]$, (b) $[0.8-0.95]$, (c) $[0.65-0.8]$, and (d) $[0.5-0.65]$. For each example we had 5 AMT workers judge the quality of the performed activity in our simulator, given its language description. Results are shown in Fig. 9. One can notice agreement between our metrics and human scores. Generally, at perfect performance the simulations got high human scores, however, there are examples where this was not the case. This may be due to imperfect animation, an indication that further improvements to our simulator are possible.

Implications. The high performance of text-based activity animation opens exciting possibilities for the future. It

Description: Get an empty glass. Take milk from refrigerator and open it. Pour milk into glass.

Description: Go watch TV on the couch. Turn the TV off and grab the coffee pot. Put the coffee pot on the table and go turn the light on.

Description: Look at the clock then get the magazine and use the toilet. When done put the magazine on the table.

Description: Take the face soap to the kitchen counter and place it there. Turn toaster on and then switch it off. Place the pot on the stove.

Figure 8: Our agent executing generated programs from descriptions, in our VirtualHome. Top description is from *ActivityPrograms*, while the rest are from *VirtualHome Activity* dataset. Notice that the top agent uses his left to open the fridge and to grab an object since he already holds an item in his right. There are also some limitations, for example, in row 3 the agent sits on the toilet fully clothed. Furthermore, sometimes the carried item slightly penetrates into the character's body due to imprecisions of the colliders.
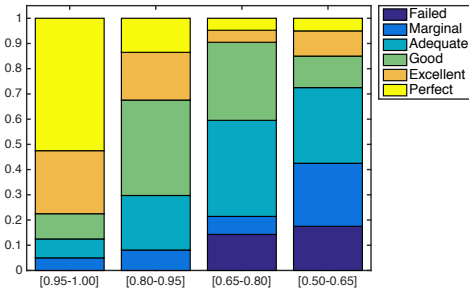

Figure 9: Human evaluation of our agent executing described activities via program generation from text: x axis shows the program prediction accuracy, y axis is the human score.

would allow us to replace the more rigid program synthesis that we used to create our dataset, by having annotators create these animations directly via natural language.

7. Conclusion

We collected a large knowledge base of how-to for household activities specifically aimed for robots. Our dataset contains natural language descriptions of activities as well as *programs*, a formal symbolic representation of activities in the form of a sequence of steps. What makes these programs unique is that they contain *all* the steps necessary to perform an activity. We further introduced *VirtualHome*, a 3D simulator of household activities, which we used to create a large video activity dataset with rich ground-truth. We proposed a simple model that infers a program from either a video or a textual description, allowing robots to be "driven" by naive users via natural language or video demonstration. We showed examples of agents performing these programs in our simulator. There are many exciting avenues going forward, for example, training agents to perform tasks from visual observation alone using RL techniques.

References

[1] https://scratch.mit.edu/.

[2] https://bitbucket.org/Unity-Technologies/ml-imagesynthesis.

[3] J.-B. Alayrac, P. Bojanowski, N. Agrawal, I. Laptev, J. Sivic, and S. Lacoste-Julien. Unsupervised learning from narrated instruction videos. In *CVPR*, 2016.

[4] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *ICML*, 2015.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. In *arXiv:1606.01540*, 2016.

[6] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *Proc. of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[7] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig. Virtual worlds as proxy for multi-object tracking analysis. In *CVPR*, 2016.

[8] J. Johnson, B. Hariharan, L. van der Maaten, J. Hoffman, L. Fei-Fei, C. L. Zitnick, and R. Girshick. Inferring and executing programs for visual reasoning. In *arXiv:1705.03633*, 2017.

[9] R. Kiros, Y. Zhu, R. Salakhutdinov, R. Zemel, A. Torralba, R. Urtasun, and S. Fidler. Skip-thought vectors. *NIPS*, 2015.

[10] S. Lauria, G. Bugmann, T. Kyriacou, J. Bos, and A. Klein. Training personal robots using natural language instruction. *Intelligent Systems*, pages 38–45, 2001.

[11] C. Li, D. Tarlow, A. L. Gaunt, M. Brockschmidt, and N. Kushman. Neural program lattices. In *ICML*, 2017.

[12] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočisky, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. *arXiv:1603.06744*, 2016.

[13] M. MacMahon, B. Stankiewicz, and B. Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *AAAI*, 2006.

[14] H. Mei, M. Bansal, and M. R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *AAAI*, 2016.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[16] D. Nyga and M. Beetz. Everything robots always wanted to know about housework (but were afraid to ask). In *IROS*, pages 243–250, 2012.

[17] C. Quirk, R. Mooney, and Y. M. Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*, 2015.

[18] S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel. Self-critical sequence training for image captioning. *CoRR*, abs/1612.00563, 2016.

[19] S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In *ECCV*, 2016.

[20] O. Sener, A. Zamir, S. Savarese, and A. Saxena. Unsupervised semantic parsing of video collections. In *arXiv:1506.08438*, 2015.

[21] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Aerial Informatics and Robotics platform. Technical Report MSR-TR-2017-9, Microsoft Research, 2017.

[22] G. A. Sigurdsson, G. Varol, X. Wang, A. Farhadi, I. Laptev, and A. Gupta. Hollywood in homes: Crowdsourcing data collection for activity understanding. In *ECCV*, 2016.

[23] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[24] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. J. Teller, and N. Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*, 2011.

[25] Y. Yang, A. Guha, C. Fermuller, and Y. Aloimonos. Manipulation action tree bank: A knowledge resource for humanoids. In *IEEE-RAS Intl. Conf. on Humanoid Robots*, 2014.

[26] Y. Yang, Y. Li, C. Fermuller, and Y. Aloimonos. Robot learning manipulation action plans by "watching" unconstrained videos from the world wide web. In *AAAI*, 2015.

[27] B. Zhou, A. Andonian, and A. Torralba. Temporal relational reasoning in videos. *CoRR*, abs/1711.08496, 2017.

[28] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler. Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books. In *ICCV*, 2015.

[29] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *arXiv:1609.05143*, 2016.