VirtualHome: Simulating Household Activities via Programs

by

Xavier Puig Fernandez

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2018

Signature redacted

Certified by .....
Antonio Torralba
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

VirtualHome: Simulating Household Activities via Programs

by

Xavier Puig Fernandez

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In order to learn to perform complex activities, autonomous agents need to know the sequences of actions needed to reach a given task. In this thesis, 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.

Thesis Supervisor: Antonio Torralba
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to start by thanking Antonio Torralba for all his advice in this work and the research I have done with him, but also for the opportunity to be part of his team and meet outstanding researchers, for all the support and for the passion he has transmitted me for Computer Vision. I would also like to thank Sanja Fidler for hosting me in the University of Toronto, for the endless help and discussions and for the opportunity to start and continue this project together. This work would not be possible without my collaborators Kevin Ra, Marko Boben, Jiaman Li and Tingwu Wang.

I would also like to thank my labmates, Yusuf Aitar, Manel Baradad, David Bau, Aditya Khosla, Àgata Lapedriza, Yunzhu Li, Javier Marin, Dim Papadopoulos, Adrià Recasens, Amaia Salvador, Dídac Surís, Carl Vondrick, Jonas Wulff, Hang Zhao, Bolei Zhou and Jun-Yan Zhu for all the research discussions and collaborations, as well as the professional and personal advice throughout my time here.

I want to thank la Fundació La Caixa and the Massachusetts Institute of Technology for all the economical and logistic support.

Thank you to all the friends I have met here, for helping in making this an incredible experience. Thank you Junkal, Livia, Lorenzo, Inés, Àngels, Hector, Toni, Carla, Tal and many others for helping me feel like at home. Thank you to my long-time friends in Spain: Gonzalo, Marc, Albert, Míriam and Víctor, for sharing this journey with me despite the physical distance.

Finally, I would like to deeply thank my family, without which I would not have made it here. Thank you Albert and Guillem, for all the encouragement, fun times and love. Lastly thank you to my parents Maite and Lluís, for always being there, for all the love, the guidance and all the unconditional support. Thank you for everything.

Contents

List of Figures

List of Tables

Chapter 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 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 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 and design a programming language for modeling activities, allowing to collect programs that formalize the activity as described in the knowledge base. This allows to collect a database of programs rep-

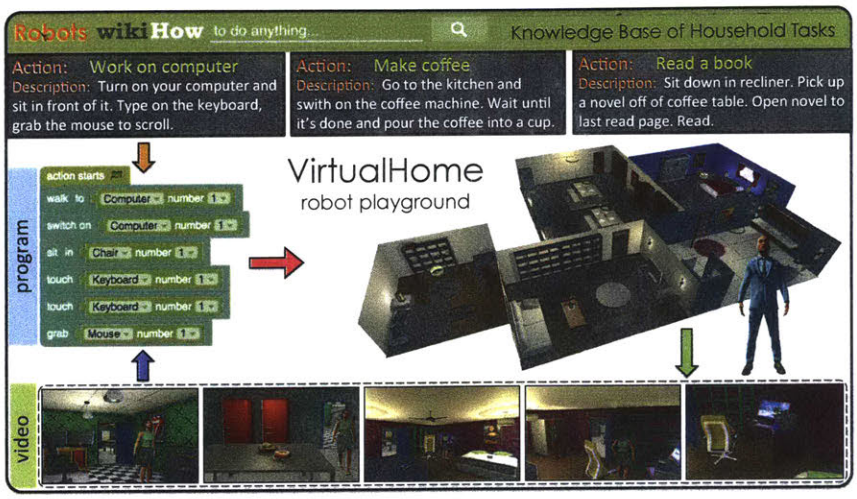


Figure 1-1: We first crowdsource a large knowledge base of household tasks, **(top)**. Each task has a high level name, and a natural language instruction. We then collect "programs" for these tasks, **(middle left)**, where the annotators "translate" the instruction into simple code. We implement the 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.

resenting activities, providing information about object affordances, typical locations of objects or co-occurrence of actions within an activity. Furthermore, the collected 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 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 Chapter 5 we show how we can learn to automatically translate natural language instructions of activities into programs. In Chapter 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.

Chapter 2

Related Work

2.1 Actions as programs

A few works have defined activities as programs. In [34], 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 [33], creating a knowledge base of actions as programs for cooking. [22] tried to translate cooking recipes into action plans using an MRF. [26, 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.

[28] 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.

2.2 Code generation

There is increased interest in generating and interpreting source code [17]. 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, [23] produces If-This-Then-That code. [18] generates a program specifying the logic of a card game given a short description of the rules. In [12], the authors inferred programs to answer visual questions about images. In recent work [9], authors adversarially train an agent to generate programs that produce target images when executed on a rendering engine. Our work differs in the domain, and works with text or video as input.

2.3 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 [30, 19, 16, 20]. 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. In [15], authors model the spatio-temporal relations of actions and objects in high-level activities, allowing robots to anticipate actions to assist humans. The work uses a dataset of 10 activities, composed of 10 atomics actions. Our proposed dataset contains a broader set of activities and they are composed of a larger vocabulary of atomic actions and objects.

2.4 Simulation

Simulations using game engines have recently been developed to facilitate training visual models for autonomous driving [8, 25, 6], quadcopter flying [27], or other robotic tasks [5]. Some works have released simulators for target-driven indoor navigation

[14, 32] and question answering [32, 10]. AI2-THOR [14] includes actionable objects, allowing to perform a set of interactions in the environment. Our proposed environment is similar in this sense, but allows to execute a wider range of actions, allowing to simulate complex 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.

2.5 Household Activity Datasets

Over the past years, there have been multiple efforts in collecting video datasets to understand every-day activities and acquire common sense knowledge [28, 11, 7]. [28] collects videos of people performing multiple actions by asking to perform activities given a script. However, the script is built by using a constrained set of actions and objects, which limits the complexity of the activities. In [7], authors collect common activities implicitly by crowd sourcing *Lifestyle Vlog videos* and annotating them afterwards. While this allows for a more natural and complex set of activities, they only provide annotations about objects and people poses and body parts, missing explicit annotations of the activity being performed. Our work focuses on generating synthetic videos of common activities, allowing to obtain a diverse set of annotations and generate complex videos with a broader set of action and object interactions.

Chapter 3

Knowledge Base of Household Activities

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 so that robots can understand and perform them. If this information was collected for humans, we could give a description of how to perform each task, but 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. 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 how to perform the 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 on how to perform the activity. 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 their implied goals.

3.1 Data Collection

In order to collect a diverse set of activities, we use crowd-sourcing to build our Knowledge Base. Describing activities as programs is not a trivial task, and using crowd-sourcing makes it more challenging, as most annotators have no programming experience. We address this by splitting the data collection into two parts.

In the first part, we asked Amazon Mechanical Turk (AMT) workers to provide verbal descriptions of daily household activities. In particular, each worker was 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 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. 3-3. Note that these descriptions are written as if the activity was being described to another human. Because of this, they may likely omit some necessary steps that are commonsense, for example, opening a fridge before grabbing something from it and closing the fridge afterwards.

In the second stage, we showed the collected descriptions to the AMT workers and asked them to translate these descriptions into programs, telling them to produce a program that would "drive" a robot to successfully accomplish the described activity, without missing any step. We designed a programming language to represent each description, together with an interface so that annotators with no programming experience could write them. Our interface builds on top of Scratch [1], a visual programming language developed by the Lifelong Kindergarten Group at MIT Media Lab aimed at introducing young children into programming. In Scratch, people write programs by stacking blocks of instructions as if they were Lego pieces, allowing to create interactive stories, games or animations. We designed our language so that each activity program could also be written by composing blocks representing simple actions or interactions, such as *sit*, *walk*, *grab* or *open*.

Each block contains the name of an action and a list of arguments to be filled with

Figure 3-1: Example block to construct activity programs.

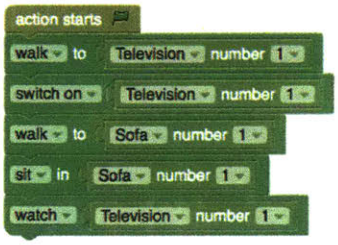


Figure 3-2: Example program for watch tv.

objects. Actions like *stand up* would not need any argument whereas *put something into something* would require two arguments to be filled with objects. Each object is linked to an instance number so that the programs are not ambiguous when they refer to multiple objects belonging the same class: if an activity consists in setting a table for two people, this allowed to disambiguate which plate should be used at every instruction. An example block can be seen in figure 3-1.

The blocks allow to select from a predefined list of 77 actions and 312 objects, compiled by analyzing the frequency of verbs and objects in the collected descriptions. We manually added affordance constraints to the objects, so that it is not possible to create invalid action/object combinations such as *grab shower*. We also allowed annotators to use a "special" block" which allowed to introduce missing actions as free-form text. These blocks were later discarded or replaced, but they allowed to identify blocks that needed to be added into the interface. Figure 3-2 shows an example of a program collected for "watch tv", where the instance number indicates that the television always refers to the same object instance.

Annotators were firstly shown a set of activities and descriptions to annotate, then they were asked to select, for each of them, the rooms and objects needed to perform the activity, and they finally created a program by using those items. Figure 3-3 shows the annotation interface.

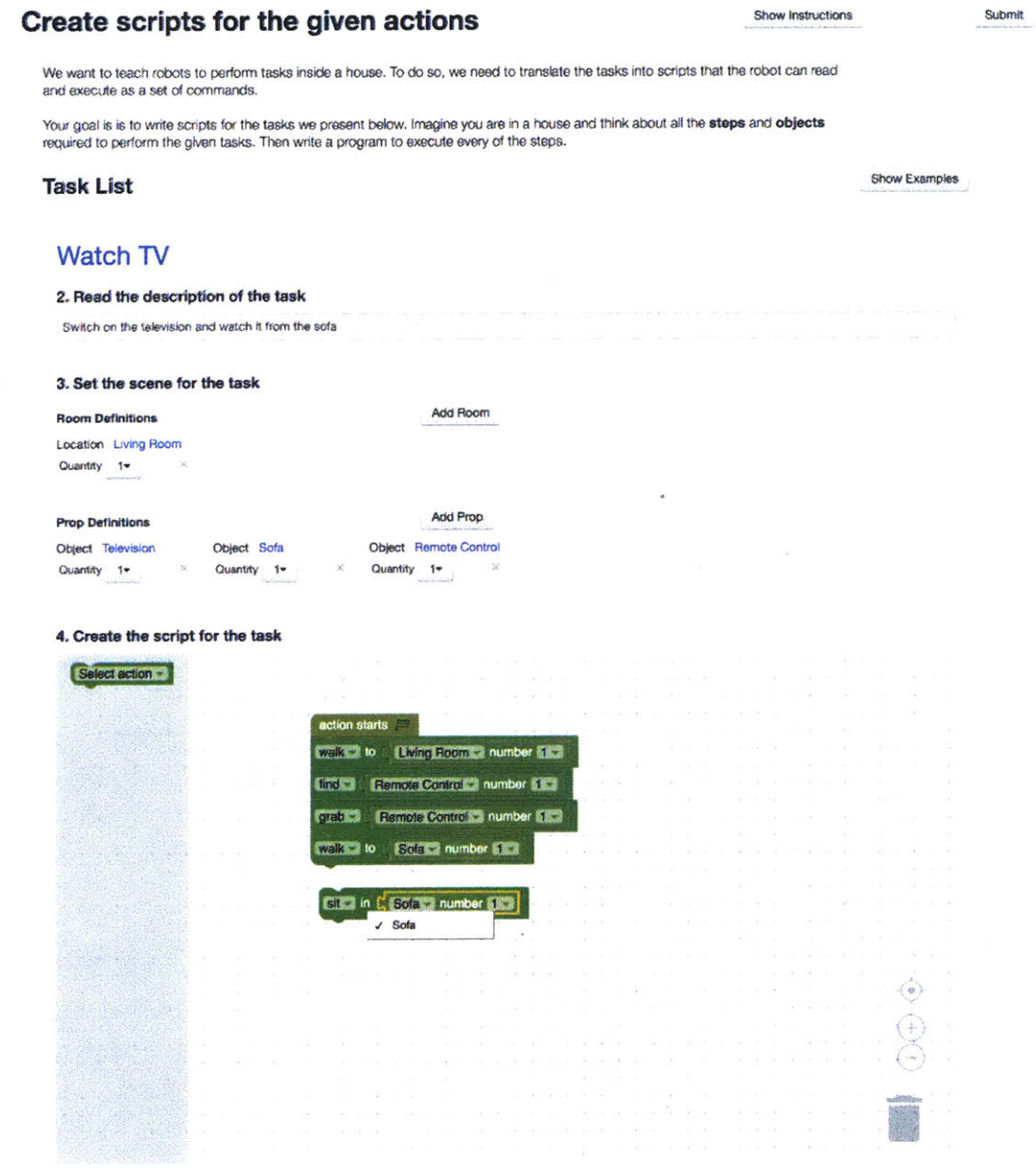


Figure 3-3: Interface for annotating programs from descriptions. Annotators would first read the description of the activity (step 2). They would set the scene (3) by adding the necessary objects and rooms and they would finally write a program by composing blocks (4).

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

Table 3.1: We analyze programs and natural language descriptions for both real activities in *ActivityPrograms* and *SyntheticPrograms* (4) with real descriptions

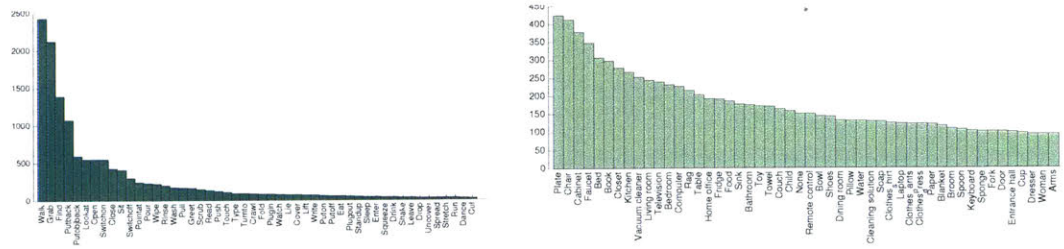
We started annotating a small set of programs using Upwork crowd-sourcing platform, which allows to hire freelancers to perform jobs. While this approach was more expensive and slow than using AMT, it allowed to obtain very high quality programs, as well as feedback on how to improve the interface for easier crowd-sourcing. We later asked annotators on AMT to write programs for our collected descriptions, which were finally validated by the annotators in Upwork. We found that annotators on AMT were capable to quickly learn to produce programs by providing a carefully designed tutorial, costing 15 cents per program annotated.

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 subset of the 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 3.1.a.

The dataset covers 75 atomic actions and 308 objects, making 2709 unique steps. Fig. 3.2.a shows a histogram of the 50 most common actions appearing in the dataset, and, Fig. 3.2, the 50 most common objects.

The programs in the dataset represent 576 activities, each with several examples, and we analyze their diversity by comparing their programs. Table 3.2 analyzes 4 selected activities. We compute their similarities as the average length of the longest



(a) (b)

Figure 3-4: Histogram of the most common actions (a) and objects (b) in *ActivityPrograms*

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

Table 3.2: 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 the LCS by the length of the longest of the two programs.

common subsequence computed between all pairs of programs within the activity. Figure 3-5 shows some example programs for the activities "Make coffee" and "Read a book".

We can also measure the similarity 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. We can measure the similarity between 2 activities by taking the average similarity across programs belonging to the 2 activities. Table 3.3. shows the similarity matrix (sorted to better show the block diagonal structure) between different activities in our dataset. As it can be seen, the similarity measure allows to cluster semantically similar activities.

The collected programs also provide a source of commonsense knowledge. By looking at the instructions, one can infer information such as the relative location of objects (e.g. the nightstand is near the bed, Fig. 3-5), or the affordance of certain obejcts (e.g. spectacles are used to read, Fig. 3-5)

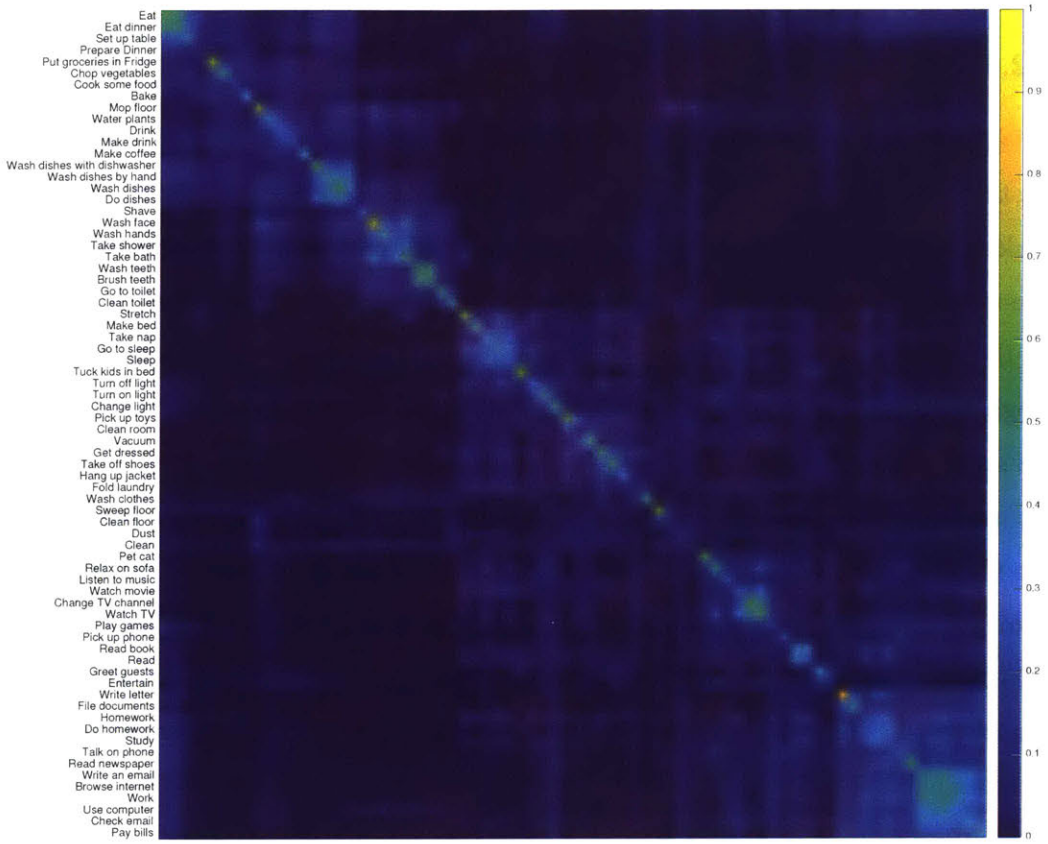


Table 3.3: *ActivityPrograms* similarity matrix (sorted to better show the block diagonal structure) between different activities in our dataset.

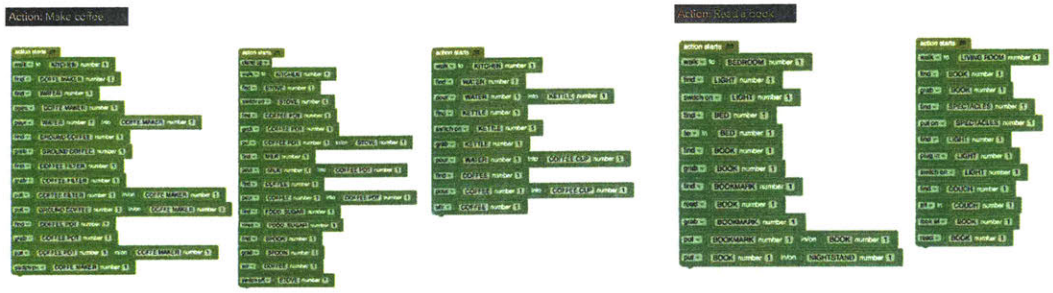


Figure 3-5: Examples of programs for the activities *Make a coffee* and *Read a book*.

3.2.1 Completeness of programs

We analyze the correctness and completeness of the collected programs to execute the task they describe. To do so, we sample 100 of the collected programs, and ask 5 workers on AMT to rate each of the programs on whether they are complete, missing minor steps or missing important steps. A program is complete when it contains all the actions so that a robot could execute them, misses minor steps when the program says "sit on the sofa" but there is no previous instruction to "walk towards the sofa" and misses crucial steps when it lacks an action that is crucial to complete the activity, for example "switch on TV" in order to watch TV, or pour water into a cup if one needs to drink water. For each of the programs, we take the median score from the 5 workers as a measure of the quality of that program. Results show that 64% of the programs are complete, 28% are missing minor steps and 8% are missing crucial steps. Note that many of the minor steps can be automatically corrected by, for example, adding walking actions before interacting with objects.

Chapter 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.

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 contains 357 object instances (86 per room). We collected objects from additional

Figure 4-1: 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 4-2: Agents in VirtualHome. We use *male 1* and *female 1* in train., and all agents when testing our video-to-program model.

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 4-1 and 4-2.

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.

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

Figure 4-3: Examples of how our hand poses are used while interacting with an object. We support both left and right for each hand pose.

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. 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 distinguish between actions that require interacting with objects and actions that do not. The later ones correspond to locomotion actions (*walk* and *run*) and *standup*. Locomotion actions are implemented using Unity's NavMesh framework for navigation (path planner to avoid obstacles) and *standup* is implemented through an animation. For the actions requiring objects, we compute the agent's target pose and animate the action using RootMotion FinalIK inverse kinematics package. To allow for more realistic animations, we annotate typical hand poses for humans when interacting with objects, shown in Figure 4-3. 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. The animation is done by keeping a state over the object, and changing it when the character's hand reaches the object.

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 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. Using this knowledge base, we place al missing objects by randomly selecting the 3D model for the object class and the target supporting surface, making sure it is placed in a free space on the selected surface by using the bounding boxes of the placed objects.

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 of the action, orientation). 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. Given that we want to generate videos depicting the executed activities, we place 6-9 static cameras in each room, 26 per home on average, allowing a third person view of the action. 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

Figure 4-4: Different groundtruth modalities generated for VirtualHome videos.

12 most frequent actions. We thus create another dataset that contains programs containing only these actions in their steps, in order to be able to execute all of the programs. The creation of this dataset is explained below.

We synthesized 5,193 programs using a simple probabilistic grammar encoding activities supported by *VirtualHome*, such as *watch TV*, *work on computer* or moving objects. For each program, we asked a human annotator to describe it in natural language. 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.

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.

Chapter 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. Note that once a script is generated, an agent could perform the task it describes in a different environment, using new 3D models and planning paths.

We aim to generate programs that are 1) consistent with the input description or video and 2) executable in VirtualHome, avoiding actions such as standing up before having sit or closing a fridge that is already closed. A script fulfills the second criteria when it is fully executable in the environment. We measure the first criteria by calculating the normalized longest common subsequence with the ground-truth script

$$\frac{|LCS(Prog_{gt}, Prog_{pred})|}{max(|Prog_{gt}|, |Prog_{pred}|)} \tag{5.1}$$

We simplify our programs and remove the instance number for the task of prediction, our scripts are now a sequence of steps of the form:

$$step_t = [action_t] < object_{t,1} > ... < object_{t,n} > \tag{5.2}$$

Where n be $0, 1$ or 2 depending on the action. We treat the task of transcribing an

input (description or video) into a program as a translation problem. We adapt the seq2seq model [29] for our task, and train it with Reinforcement Learning to optimize the two objectives.

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 the same architecture for both description and video and assume the input has been transformed to a sequence of vector embeddings. We use a LSTM with 100-dim hidden states as our encoder. At each step t, our RNN decoder decodes a step which takes the form of 5.2. 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:

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

$$p_i^t = \text{softmax}_i(\frac{v_i}{\|v_i\|}^T \cdot W_v(h^t \| \mathbf{x}_t^{att})) \tag{5.3}$$

where W_a and W_o and W_v are learnable matrices, and v_i denotes an embedding of an instruction.

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 [29], we further use the attention mechanism over the encoder's states to get another feature $\mathbf{x}_t^{att}$. In particular:

$$\alpha_j^t = \text{softmax}_j(v^T \left(W_{att} \left(h^t \| h_{enc}^j\right)\right)) \tag{5.4}$$

$$\mathbf{x}_t^{att} = \sum_j \alpha_j^t h_{enc}^j \tag{5.5}$$

where W_{att}, v are learnable parameters. An overview of the model can be seen in figure 5.

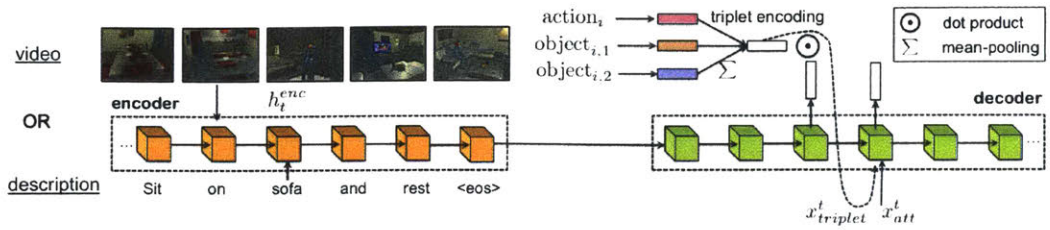


Figure 5-1: Our encoder-decoder LSTM for generating programs from natural language descriptions or videos.

5.1 Learning and inference

. Our goal is to generate programs that are both close to the ground-truth programs in terms of their LCS (eq. 5.1) 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 [21] embeddings for matrices W_a and W_o and fix their weights.

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 [24], and use policy gradient optimization to train the model, using the greedy policy as the baseline estimator. In particular, given the policy p_θ defined in eq.5.3, we define the sequence generated by the greedy policy as $\hat{w} = \{\hat{w_1}, ..., \hat{w_T}\}$ with:

$$\hat{w_t} = \arg\max_{w_t} p_\theta(w^t) \tag{5.6}$$

And the sequence w^s generated by sampling from the policy p_θ. Our reward becomes $r(w^s, g) - r(\hat{w}, g)$. Given that the baseline $r(\hat{w}, g)$ does not depend on the sampled w^s, we have that

$$\mathbb{E}_{w^s \sim p_\theta}[r(\hat{w}, g)\nabla_\theta \log p_\theta(w^s)] = r(\hat{w}, g)\sum_{w^s} \nabla_\theta p_\theta(w^s) = r(\hat{w}, g)\nabla_\theta \sum_{w^s} p_\theta(w^s) = 0 \tag{5.7}$$

And therefore

$$-\mathbb{E}_{w^s \sim p_\theta}[(r(w^s, g) - r(\hat{w}, g))\nabla_\theta \log p_\theta(w^s)] = -\mathbb{E}_{w^s \sim p_\theta}[(r(w^s, g)\nabla_\theta \log p_\theta(w^s)] \quad (5.8)$$

Which corresponds to the expected gradient of the loss using REINFORCE [31]. This means tha using the baseline will not affect the expected gradient, while allowing to reduce its variance.

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 (eq. 5.1) 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 start by training using the LCS reward alone, and fine-tune the best model using a balance of the two rewards, as $r(w^s, g) = r_{LCS}(w^s, g) + 0.1 \cdot r_{sim}(w^s)$.

So far we did not describe the input to the RNN encoder. Our model accepts either a language description or a video depicting the action, which are converted into a sequence of vector embeddings to serve as input to the encoder show in Fig. 5. We explain in the following sections how the embeddings are obtained for each modality.

5.2 Textual Description

To encode a textual description, we split the description by its words, remove punctuation symbols and use word2vec [21] embedding trained on GoogleNews to encode each word.

5.3 Video

To generate programs from videos, we partition each video into 2-second clips, corresponding to 9 frames and encode each of the clips as the embedding of the instruction

happening at the middle frame. These embeddings serve as input to our RNN encoder. Notice that, some actions (e.g. *walk*) take longer than others (e.g. *switch on*). In order to avoid having an imbalance in the number of clips for the long actions, we allow a maximum of 5 clips for every step in the video.

To obtain the embedding of each clip, we use the Temporal Relation Network [36] with 4-frame relations to predict the embedding of an instruction (action+object+object). Given, the embedding, we obtain the probability of every instruction through eq. 5.2 and train the network using cross-entropy loss.

We still have to specify what is the input of the TRN model. In order to make the model potentially generalizable to real videos, we use the semantic segmentation mask, which we obtain by training a DilatedNet [35] segmentation network using the ground-truth segmentations.

Chapter 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 Instruction Classification from Video

We first evaluate our model for the task of video-based action and action-object-subject (step in the program) classification. Here, we partition each video in 2-sec clips, and use the clip-based TRN on the predicted segmentation mask to perform classification. We compute performance as the mean per-class accuracy across all 2-sec clips in *test*. We consider accuracy at the level of *action*, *object*, or the whole *instruction*. 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 6.1. To set the lower bound, we also report a simple *random retrieval* baseline, in which a program is randomly retrieved from the training set. In figure 6.1 we show the confusion matrix for action classification. Notice that the model is biased towards the *walk* action but that is also the most common action in the programs, and we empirically found out that this setting provided better script generation results. The

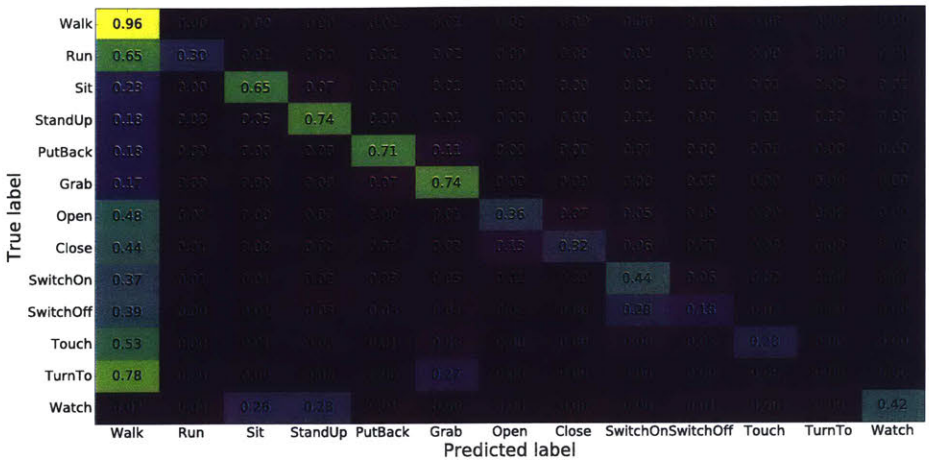


Figure 6-1: Confusion matrix for action classification in 2-sec clips.

	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%

Table 6.1: Accuracy of *video-based action classification* and *action-subject-object* (step in the program) prediction in **2-sec clips** from our VirtualHome Activity dataset.

model struggles to differentiate between *switch on* and *switch off*, which makes sense given that this information is generally not encoded in the semantic segmentation. We can also see that the model mistakes Watch with *sit/stand up*, which is due to the fact that the video is clipped without taking into account action boundaries. Given that *watch* is normally preceded by a *sitting action* (e.g. for *watch tv*) followed by *standing up*, it is likely that some shots fall in the intersection between these actions.

6.1.1 Effect of input

We also study the effect of using ground-truth segmentation or RGB images in the clip classification task. The results are shown in table 6.2. As it can be seen, there is a significant performance drop from ground-truth segmentation to the predicted one,

	Action	Objects	Steps	Mean
Seen homes	78.06 %	73.00 %	56.42 %	69.16%
Unseen homes	54.89%	52.94%	43.50%	50.44%
All	62.58%	61.18%	48.49%	57.42%

(a)

Action	Objects	Steps	Mean
77.46 %	60.21 %	44.45 %	60.71%
40.56%	26.56%	26.46%	31.19%
54.60%	40.44%	35.17%	43.40%

(b)

Table 6.2: Accuracy of *video-based action classification* and *action-subject-object* (step in the program) prediction in **2-sec clips** using ground-truth segmentation (a) and RGB images (b).

which suggests that more effort should be put in training the segmentation. Many of the interactions in *VirtualHome* are done with small objects, with which segmentation networks typically struggle with. The model using RGB images performs better than the one using the predicted segmentations, but drops significantly in performance when testing on apartments unseen during training, which suggests it would not fit for transferring to unseen environments. Given that our future goal is to have this models working on real environments, we resort to using semantic segmentation information for our video-based prediction task.

6.2 Program Generation

We now evaluate the task of program generation. We evaluate program induction using the normalized LCS, as described in eq. 5.1. We also compute accuracies for *actions* and *objects* alone. Since LCS does not measure whether the program is valid, we report another metric that computes the percentage of generated programs that are executable in our simulator.

6.2.1 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

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%

Table 6.3: Programs from description: Accuracy on *VirtualHome Act.*. We evaluate using the normalized longest common subsequence, mimicking IoU for programs, as well as the percentage of scripts executable in the simulator

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 6.4: Programs from description: Accuracy on *ActivityPrograms*. Since real programs are mainly not executable in our simulator due to the lack of implemented actions, we cannot report the executability metric.

and its arguments, 2) *random retrieval*, where we randomly pick a program from the training set, 3) *skipthoughts*, where we embed the description using [13, 37], retrieve the closest description from training set and take its program, 4) our model trained with MLE (no RL). Table 6.4 and 6.3 shows the results. Note that the retrieval baselines (*skipthoughts* and *random retrieval*) are always executable, since our training set scripts were generated to be executable. 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. Fig. 6.2.2 shows some example results for *ActivityPrograms*, trained using the LCS reward, and Fig.6-2 shows results on *SyntheticPrograms* for the trained using both rewards.

	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 6.5: Video-based program generation.

6.2.2 Video-based prediction

We also report results on the most challenging task of video-based program generation. The results are shown in Table 6.5. 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. Fig. 6.2.2 shows example results for script generation from video on *SyntheticPrograms* test set.

6.2.3 Executing programs in VirtualHome

. Given that we are optimizing our programs to be executable in *VirtualHome*, we can try running the script predictions to generate new videos. In Fig. 6.2.3 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. 6.2.3. One can notice agreement between our metrics and human scores. Generally, at perfect performance the simulations got high human scores, however, there are examples

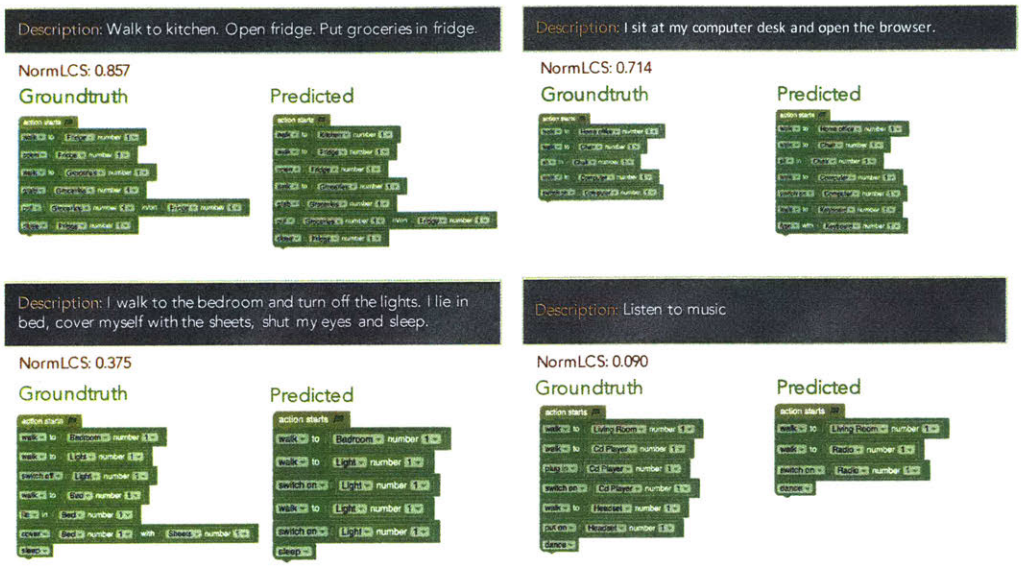


Figure 6-2: Example results for language-based prediction on *ActivityPrograms* dataset.

Figure 6-3: Example results for language-based prediction on *SyntheticPrograms* dataset.

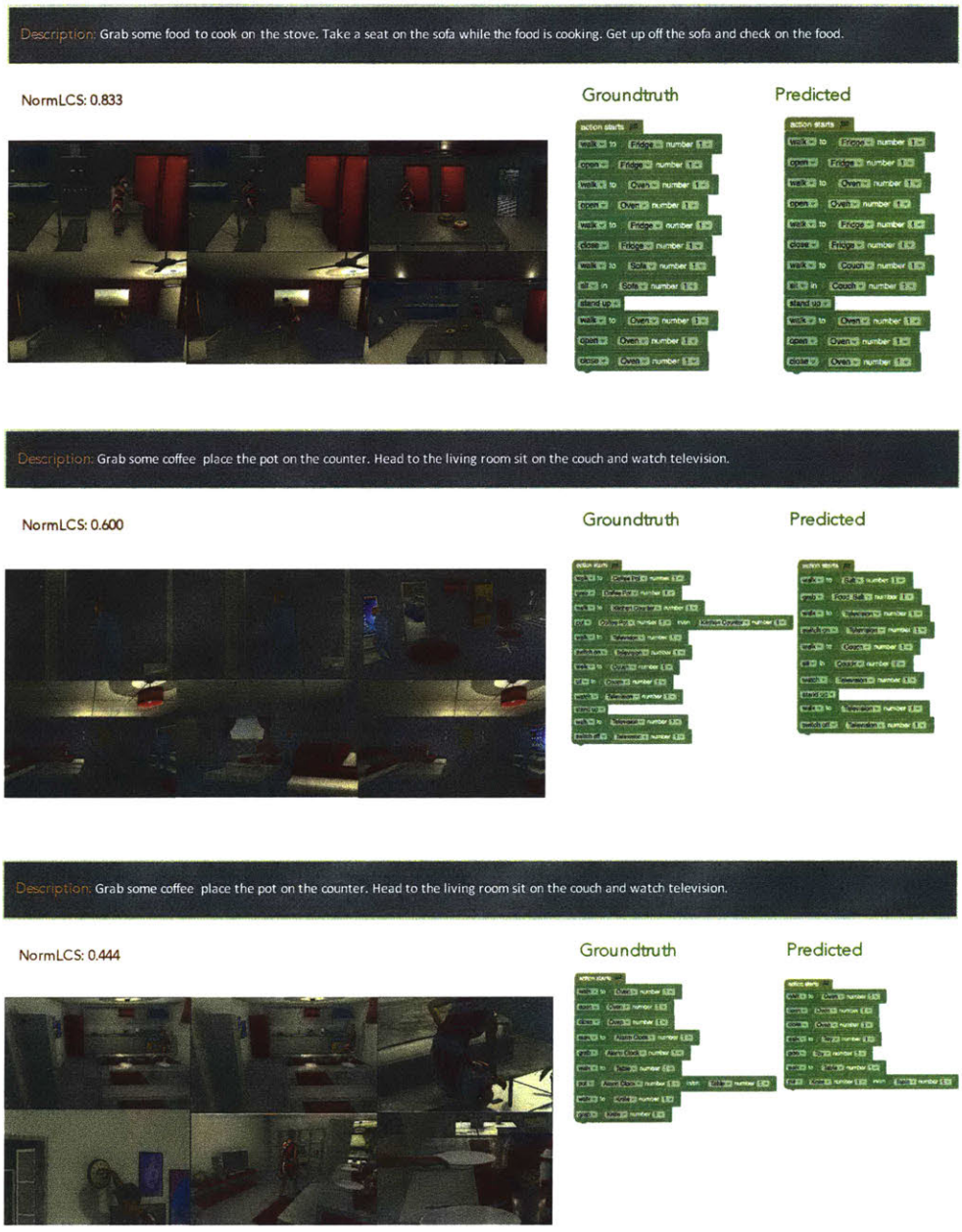


Figure 6-4: Example results for video-based prediction on *SyntheticPrograms* dataset.

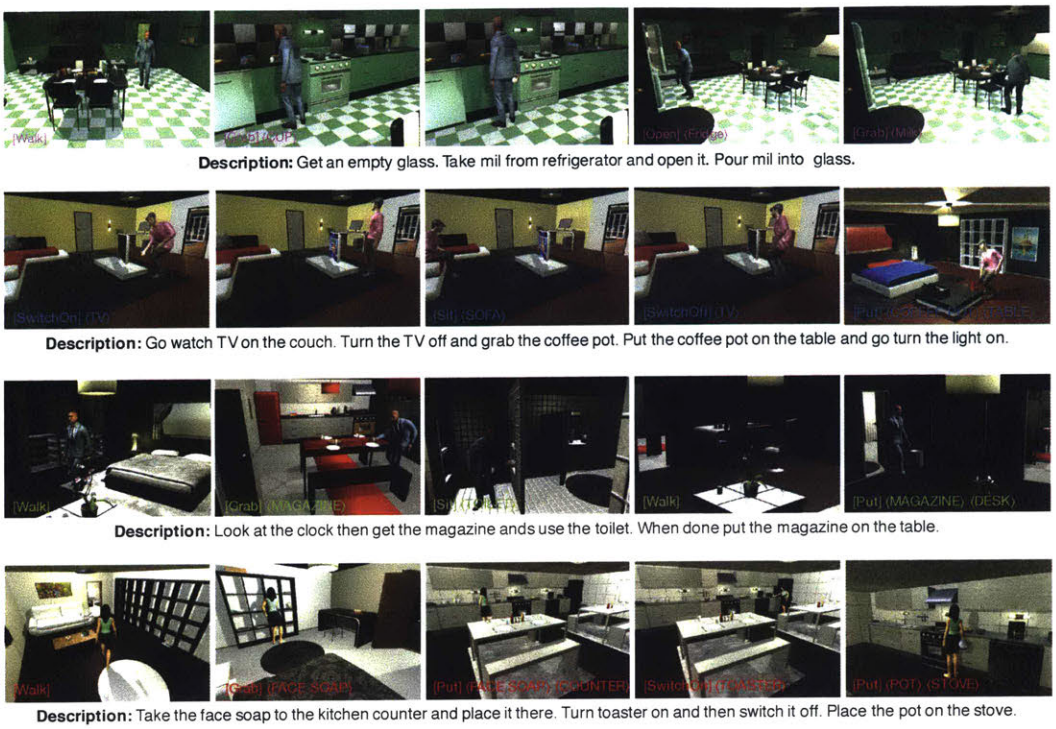


Figure 6-5: Videos generated from descriptions in *SyntheticProgram*.

where this was not the case. This may be due to imperfect animation or planning in our simulator, or the fact that the generated scripts do not have information about object instances. Future models should be able to incorporate such information in the generation of scripts.

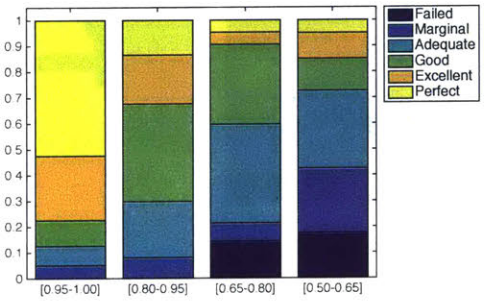


Figure 6-6: Human judgement of videos generated from text descriptions.

6.2.4 Implications

The high performance of text-based activity animation opens exciting possibilities for the future. It 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. Similarly, the text-based program generation could allow to reduce human annotation by crowd-sourcing scripts for activities from existing text corpora. We leave this as an avenue for future work.

Chapter 7

Conclusion

In order to understand or learn to perform complex activities, systems need to know what are the steps required to perform such tasks. To this end 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, as compared to human descriptions, which tens to omit common sense steps. We further introduced *VirtualHome*, a 3D simulator of household activities in which the activity programs could be executed. We used it 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 demonstrations. We finally showed examples of agents performing these programs in our simulator. We believe this work opens many exciting avenues going forward, for example, training agents to perform tasks from visual observation alone using RL techniques or anticipating actions or objects of interaction from partial observation of human activities.

Bibliography

[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] Miltos Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *ICML*, 2015.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. In *arXiv:1606.01540*, 2016.

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proc. of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[7] David F. Fouhey, Weicheng Kuo, Alexei A. Efros, and Jitendra Malik. From lifestyle vlogs to everyday interactions. In *Arxiv*, 2017.

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

[9] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018.

[10] Daniel Gordon, Aniruddha Kembhavi, Mohammad Rastegari, Joseph Redmon, Dieter Fox, and Ali Farhadi. IQA: visual question answering in interactive environments. *CoRR*, abs/1712.03316, 2017.

[11] Raghav Goyal, Samira Ebrahimi Kahou, Vincent Michalski, Joanna Materzynska, Susanne Westphal, Heuna Kim, Valentin Haenel, Ingo Fründ, Peter Yianilos, Moritz Mueller-Freitag, Florian Hoppe, Christian Thurau, Ingo Bax, and Roland Memisevic. The "something something" video database for learning and evaluating visual common sense. *CoRR*, abs/1706.04261, 2017.

[12] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *arXiv:1705.03633*, 2017.

[13] Ryan Kiros, Yukun Zhu, Russ Salakhutdinov, Richard Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Skip-thought vectors. *NIPS*, 2015.

[14] Eric Kolve, Roozbeh Mottaghi, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: an interactive 3d environment for visual AI. *CoRR*, abs/1712.05474, 2017.

[15] H. S. Koppula and A. Saxena. Anticipating human activities using object affordances for reactive robotic response. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(1):14–29, Jan 2016.

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

[17] Chengtao Li, Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, and Nate Kushman. Neural program lattices. In *ICML*, 2017.

[18] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomaš Kočisky, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv:1603.06744*, 2016.

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

[20] Hongyuan Mei, Mohit Bansal, and Matthew R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *AAAI*, 2016.

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

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

[23] Chris Quirk, Raymond Mooney, and Y. Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*, 2015.

[24] Steven J. Rennie, Etienne Marcheret, Youssef Mroueh, Jarret Ross, and Vaibhava Goel. Self-critical sequence training for image captioning. *CoRR*, abs/1612.00563, 2016.

[25] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *ECCV*, 2016.

[26] Ozan Sener, Amir Zamir, Silvio Savarese, and Ashutosh Saxena. Unsupervised semantic parsing of video collections. In *arXiv:1506.08438*, 2015.

[27] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Aerial Informatics and Robotics platform. Technical Report MSR-TR-2017-9, Microsoft Research, 2017.

[28] Gunnar A. Sigurdsson, Gul Varol, Xiaolong Wang, Ali Farhadi, Ivan Laptev, and Abhinav Gupta. Hollywood in homes: Crowdsourcing data collection for activity understanding. In *ECCV*, 2016.

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

[30] 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.

[31] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.

[32] Yi Wu, Yuxin Wu, Georgia Gkioxari, and Yuandong Tian. Building generalizable agents with a realistic and rich 3d environment. *CoRR*, abs/1801.02209, 2018.

[33] Yezhou Yang, Anupam Guha, Cornelia Fermuller, and Yiannis Aloimonos. Manipulation action tree bank: A knowledge resource for humanoids. In *IEEE-RAS Intl. Conf. on Humanoid Robots*, 2014.

[34] Yezhou Yang, Yi Li, Cornelia Fermuller, and Yiannis Aloimonos. Robot learning manipulation action plans by "watching" unconstrained videos from the world wide web. In *AAAI*, 2015.

[35] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *arXiv:1511.07122*, 2015.

[36] Bolei Zhou, Alex Andonian, and Antonio Torralba. Temporal relational reasoning in videos. *CoRR*, abs/1711.08496, 2017.

[37] Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books. In *ICCV*, 2015.