Learning to Perform Actions from Demonstrations with Sequential Modeling

A Dissertation

Brandeis University
Department of Computer Science
Dr. James Pustejovsky, Advisor

Committee Members:
Dr. Anthony G. Cohn (School of Computing, University of Leeds)
Dr. Pengyu Hong (Dept. of Computer Science, Brandeis University)
Dr. Timothy J. Hickey (Dept. of Computer Science, Brandeis University)

Submitted by Tuan Do
tuandn@brandeis.edu

May, 2018

ABSTRACT

Learning from demonstration (or imitation learning) studies how computational and robotic agents can learn to perform complex task by observing humans. This closely resembles the way humans learn. For example, children can imitate adults in a variety of domestic tasks, such as pouring milk from a carton to a cup or cleaning a table, and after one or a few observations, they can replicate the action with relative accuracy. This research would test the feasibility of a virtual agent that can process multimodal (linguistic and visual) captures of actions and learn to reenact them in a simulated environment.

In particular, this thesis would look into the problem from two perspectives. The first perspective is taken in a realistic setup, in which we will teach learning agents skills by showing real demonstrations of the skills. We will only focus on a small set of actions involving spatial primitives. The objective is for agents to learn to perform complex action skills from limited amount of training samples. We will also try to teach agents in an interactive environment, in which immediate feedback is provided to agents to correct them in due time.

The second perspective is taken from a more unrealistic viewpoint. In this setup, parallel corpus mapping natural language instructions to demonstrations is generated by soliciting human descriptions for complex action demonstrations in a two-dimensional simulator. The problem is framed in a sequence to sequence translation framework. We will discuss the difference between two perspectives, and advantages and disadvantages of corresponding learning frameworks.

The system is composed of the following components: a. an event capture annotation tool (ECAT) that captures human interaction with objects using Kinect sensor; b. an event representation learning method using recurrent neural network with QSR feature extraction; c. action reenacting algorithms using reinforcement learning and sequence to sequence methods; d. evaluation methods using 2-D and 3-D simulators (based on Unity game engine).

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The community surrounding "learning from demonstration" (LfD) studies how computational and robotic agents can learn to perform complex task by observing humans (Young and Hawes, 2015). Work in this area can be traced back to reinforcement learning studies by Smart and Kaelbling (2002) or Asada et al. (1999), and closely resembles the way humans learn. Infants can imitate adults in a variety of household tasks, such as *pouring milk from a carton to a cup* or *cleaning a table*, by replicating actions after one or a few observations. This is in part an AI movement that aims to achieve *stronger* AI, which is AI systems that can adapt to new environment, to learn new skills and can cooperate with humans to perform tasks.

So far the development toward more adaptable AI has faced with many obstacles, leading to the fact that most robots developed in the previous decades have shipped with preinstalled programs, limited to a set of predefined functionalities. Learning approaches in the robotics community seek to move toward smarter and more adaptable robots, for, among others, the following reasons:

- Consumer desire for mobile or household assistant robots that can perform multiple tasks with flexible apparatus, such as multiple grasping arms (Bogue, 2017). Robots with behavioural robustness can learn from a wider range of experiences by running in a dynamic human environments (Hawes et al., 2017).
- Advances in deep learning have afforded robotic agents a high-level understanding of embedded semantics in multiple modalities, including language, gesture,

object recognition, and navigation. This increases the circumstances and modalities available for robotic learning.

- Robots that can communicate and cooperate with humans to perform tasks are desirable, as fully automated robots are far from able to perform complex tasks in novel environments or circumstances. Human can only communicate using qualitative semantics, such as ”put A closer to B“, while robots function in a lower quantitative level of semantics, such as ”put(A, coordination_XYZ)“. Understanding the nuances in human languages requires communicative robotic agents to learn various human concepts, such as the predicates ”put” or ”closer to”, in a form of action programs that are compositional with arguments.

For robots to achieve this level of autonomy and understanding, a scaffold of robot learning can be summarized as follows:

- Robots learn object forms (shape, color, size etc) and their mappings to human languages (round, square; red, green; big, small) (Alomari et al., 2017). Moreover, robots also need to learn relative relations among objects (such as whether one object could contain another object (Liang et al., 2015)) and between agents and objects (what is generally called object affordances, a.k.a what a human or robotic agent can do with the object (grasping, pouring) (Koppula et al., 2013)).
- Robots learn to perform simple actions by learning compositional programs, such as to learn the programmatic form of action verbs (put, slide, push, roll etc), in addition to programmatic form of motion adjuncts specifying path of motion (on, in, next to etc.).
- Robots learn to perform more complex actions (actions that are generally represented by structured programs, such as procedural or repetitive programs), such as “make a row from given objects”.
- Robots learn to communicate with humans in a feed-back loop to make necessary correction, or to improve learned models. While human experts can give instructions or demonstrations, robots might not be able to recognize immediately very abstract concepts that are intended by experts. For example, with the same learned action as before, a smart robotic learner can recognize that the intended action is a repetitive programs of “move A next to B”, but it would be hard to recognize “direction of extension need to follow the longest axis”.

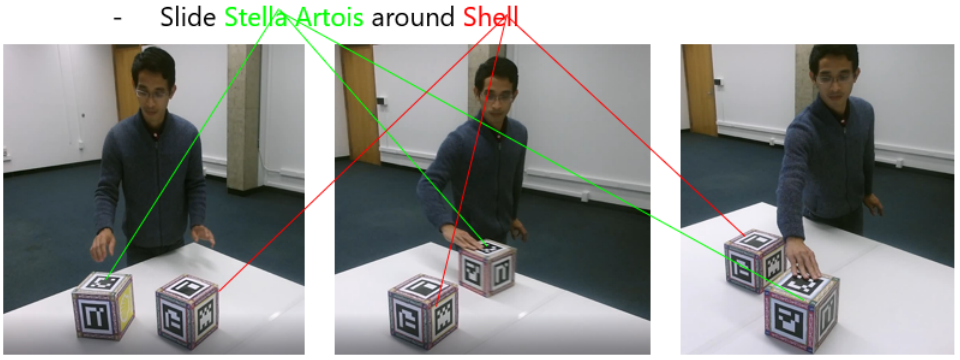


FIGURE 1.1: Learning from demonstrations with visual and linguistic inputs

While the discussion so far has been about robots, it applies to virtual agents as well. Moreover, as most of the learning process could be simulated, this dissertation work will focus on the learning capacity of a virtual agent that is co-situated with humans and participate in some collaborative tasks. The virtual agent can observe what the humans have done, perform action in a novel situation, and receive feedback from human partners.

Because of this dissertation scope, the focus of my work will be on the capacity of virtual agents to learn to perform action in regard to a set of spatial primitives that functioning as adjuncts specifying path of motions. The result of this dissertation work would be a feasibility study of a multi-step framework combining various components, including a module to learn action representation from video captures, one to generate different simulations of the virtual agents performing the action, and one to visualize those simulations for evaluation.

1.1 Background

In this section, I will recapitulate a number of ideas that have inspired my work. The arching theme of this study is the ability of AI systems to learn in an adapting environment, to communicate with humans on different levels of abstraction and by different modalities and to learn novel actions by mimicking human companions.

The most important contributing idea is a collaborative environment for communication between humans and computers, presented at Subsection 1.1.1. For humans to teach robots new actions, we can employ LfD approach, summarized at Subsection

1.1.2. While classical LfD is typically built upon robot sensorimotor modality, to extend it for linguistic modality, we need a linguistic theory for the mapping between action represented in language and in reality, discussed in Subsection 1.1.3. Two machine learning frameworks to recognize and perform actions are sketched at Subsections 1.1.4 (Sequential Modeling) and 1.1.5 (Reinforcement Learning). System evaluation is facilitated by human feedback on visualization of performed actions, presented at Subsection 1.1.6.

Other inspirations to my dissertation work come from: interpretation of spatial cognitive reasoning and its application in spatial control for robotic systems, as well as qualitative reasoning as a common knowledge interface between humans and AI systems, discussed at Subsection 1.1.8; XAI as an emerging AI sub-field that aims for machine learning models that could be interpreted by domain experts (1.1.7); usage of video captures to infer or recognize event/action structure (Subsection 1.1.9), and especially usage of RGB-D datasets (Subsection 1.1.10); cognitive linguistic work regarding human ability to learn new concepts or to perform novel actions (Subsection 1.1.11).

1.1.1 Communicating with Computers

As a general term, communicating with computers can be understood as human-computer interface (HCI), in which there is an exchange of information between human and computers. Communicating with Computers (CwC) program, a DARPA project in specific, advance HCI to take into account multiple modalities, namely vision, language and gestures. Moreover, it would enable humans to share complex concepts with computers, building upon more elementary concepts. Quoting the program's statement, it *focuses on developing technology for assembling complex ideas from elementary ones given language and context*.

More importantly, for computers to be able to understand complex concepts from elementary concepts, it is necessary that we include **learning from demonstration** as one modality of communication. Not only that this kind of learning allows an agent to learn new concepts through interaction, incremental learning allows humans to provide immediate feedback to the computers to change its execution programs.

I will also explore some potential methods to incorporate feedback from human interlocutor to improve our AI's learned action model. Because in real world collaborative

environment, communication is carried out over multimodal channels, we will discuss the form and source that feedback could be taken from. In Section 5.3.5, I will investigate use of two simple kinds of feedback (positive-negative acknowledgement and deixis/pointing feedback), leaving more complex ones, such as linguistic instruction/-correction for possible future extension.

1.1.2 Learning from demonstration

Learning from demonstration (LfD) can be traced back to the 1980s, in the form of automatic robot programming. The earliest form might be called *teaching by showing* or *guiding*, in which a robot's effectors could be moved to desired positions, and robot's controlling system would record their coordinations and rotations (controlling signals) so that when the robot is asked to execute the program, the controller would move its effectors according to the recorded controlling signals. This mechanism, though rudimentary, has proved successful with little memory or computational power in the eras before the popularity of general-purpose computers. It is, however, simply *record and replay* method, which is agnostic to the environment (Lozano-Perez, 1983).

This simple *guiding* framework has developed into robot Programming from demonstration (PbD) framework, which has several development in recent time. In basic, it still applies the same principles of using sensorimotor information of robot's joints to learn models of actions. However, it has been able to generalize from just *record and play* method, by allowing interpolation for novel configurations or new environments. Calinon et al. (2007) is an exemplary research for this framework, in which a teacher guides a humanoid robot's hands to move a chess piece forward through kinesthetics. Recently, the field starts to pick up newer machine learning tools such as Artificial Neural Networks (ANNs), Radial-Basis Function Networks (RBFs) etc. Moreover, the field progressively expands with other modalities of teaching-learning interface, such as vision, haptics etc.. A representative work that shows this development is the thesis of He (2017), in which robots observe multiple object manipulation (in-hand rotation of objects) and process them into visual and haptic features, and learning action policy by applying RBFs network over these feature vectors.

Another branch of the original line of research is its interdisciplinary variance, named **imitation learning**. Evidence from biology, neuro-science, cognitive sciences and human-computer interaction (HCI), has been used to guide its researching methods.

For instance, neuroscientific research showed that primate imitation learning requires a conceptual model of object spatial and relative locations, resolved from visual sensory information (Maunsell and Van Essen, 1983). HCI contributes a framework for incremental learning, which makes interaction between human and computers an important component in the learning process. This effectively speeds up the learning rate, corrects learning errors, or guides learning agent to focus on certain part of learning. In particular, deixis cues (pointing or gazing) could be used to limit the start and end of a demonstration, or to constraint the objects in an action (Calinon and Billard, 2006).

On the terminology used in this proposal, I will use the general term Learning from demonstration (LfD), following discussions in Argall et al. (2009). Programming by Demonstration will not be used because this term normally applies to robot PbD, and in this proposal, all experiments would be carried out in simulated environment (with embodied agent for visualization). Also following Argall et al's survey, this research proposal could be categorized as an *imitation with external observation* approach, which means the **teacher execution** (motion of human body) could not make an Identity mapping to the **recorded execution** (input from 3-D sensors), and **recorded execution** could not make an Identity mapping to the **learner execution** (the agent's planning policy). However, this subcategorization system is not widely adopted by other authors, such as Billard et al. (2008). In fact, Billard et. al refers to the *imitation* as the reenacting part of the whole system, in contrast to *demonstration*. Without adding more confusion, I would resort to use the most accepted term that could be found in the literature.

In subsections 1.1.4 and 1.1.5, I will describe two machine learning techniques that I will use: temporal-sequential modeling learns model of the skills from demonstrations, and searching/reinforcement learning methods to reproduce the learned skill in a new context.

1.1.3 Dynamic Event Structure

This work is a culmination of an effort to cross-pollinate different lines of research in our Brandeis's Computational Linguistic lab. Pustejovsky's rigorous framework of dynamic events and event participants (Pustejovsky (2013)) led to different lines of research: a top-down semantic framework called *Multimodal Semantic Simulations (MSS)*, which can be used to encode events as programs in a dynamic logic with an operational semantics, and in turn is used to create an conversational and grounding interface between

humans and computational agents; a bottom-up approach that learning event representation through machine learning methods, using data from 3-dimensional video captures. The first approach requires programming of various predicative types, such as verbal-action types (**slide**, **roll**, **put**), and spatial adverbials (**on**, **in**, **next to**, **around**). The second approach, so far, can only make distinction between different verbal-action types and spatial adverbial types (Do and Pustejovsky (2017a)). This work is trying to bridge the gap between two approaches, which is to learn the programmatic form of action directly from the data.

This work is also related to a traditional treatment in computational linguistics, in particular, in lexical semantic analysis, in which, motion verbs can be divided into two classes *manner*- and *path*- oriented predicates, and adjuncts can be used to complement meaning on the other aspect of motion (Jackendoff, 1983). These classification correspond to answers *how* the movement is happening and *where* it is happening. Taking, for example, these two following sentences (Krishnaswamy, 2017):

(c) The ball rolled$_m$ across the room.

(d) The ball crossed$_p$ the room rolling.

In the first sentence, the predicative verb describes the *manner* of motion, while the adjunct refers to the motion's path and vice versa for the second sentence. While this work will focus on machine learning ability in regard to motion's path, the methodologies in this work can be applied to manner of motions as well, with a consideration that *manner* of motions generally refers to movement gradient of the object over time. For example, *slowly* refers to small change of movement, while *roll* refers to intrinsic change of object orientation over time. The difficulty of learning these *manners* of motion lies in the fact that these intrinsic movements sometimes depend on object affordances. For instance, a ball can be rolled, but not a cube.

1.1.4 Temporal-sequential modeling

Essentially, temporal-sequential models are any algorithmic, machine learning or logical models that allow us to represent changing of system states over time. There are many different types of models belonging to this category, with a wide variety of application, including classification (Do and Pustejovsky, 2017a), text/dialog generation

(Serban et al., 2017) etc. Some popular sequential modeling systems are: finite-state machine or finite automaton (FSM) in either deterministic or non-deterministic forms; its development with more explanatory power, Hidden Markov Model (HMM); interval temporal logic etc. More recently, the development of neural methods has brought about several sequential neural network models, such as Recurrent neural networks (RNNs), with its flavors such as Long short term memory (LSTMs) or Gated recurrent unit (GRU).

Arguably, predictive sequential learning model is the most appropriate model for learning of trajectory level skills. What the learners observe in each action demonstration is a frame-by-frame sequence of feature vectors, and a predictive model could be used to produce different label of the sequence: which category of action it is, whether it has been terminated or not etc. A feed-forward network such as RNN can be aptly trained to fit high level of complexity patterns in the feature vectors.

In Chapter 4, I will use RNN models to learn an action recognizer that can distinguish between fine-grained actions. The RNN models we use are instances of sequence-to-one classifiers that predict a distribution of action classes for each input.

In Chapter 5, I will use RNN models to learn functions called *progress functions*, each helps to guide the agents to complete an action. A progress function is an instance of sequence-to-one regressor that produces a value from 0 to 1, that will be used to as a reward function for reinforcement learning (briefly described in Section 1.1.5).

In Chapter 6, I will use a more advanced form of RNN models (called Attention RNN) to learn mapping from natural language instructions to action sequences. The model is an instance of Seq2Seq models that have been successfully used in a wide spectrum of applications, such as neural machine translation (Wu et al., 2016) and image caption generation (Xu et al., 2015).

1.1.5 Reinforcement learning

Reinforcement learning (RL) is a set of methodologies used for robotic or virtual agents to learn by interacting with environment, trying to find an action policy to optimize a target value function while exploring the action search space. In this work, I will explore some combination of searching methods and reinforcement learning methods, especially ones belonging to Monte Carlo family.

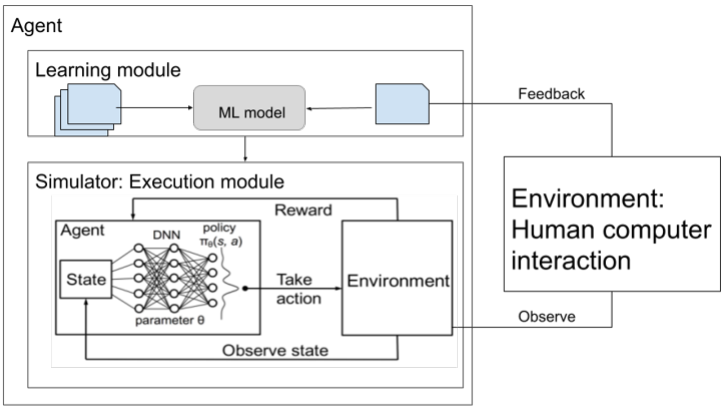


FIGURE 1.2: Reinforcement learning framework

A general formalization of reinforcement learning includes the following components: a *policy* dictates actions of agents given current state of the environment (*state* or *observation*), a *reward signal* dictates the goal of RL problem in a form of a single reward value returned from the environment after each action is performed, and a *value function* is the accumulated rewards over long run. In this problem, *state* refers to a feature vector that describes the current configuration of simulated environment, which I would build upon qualitative spatial features between objects, *action* refers to movement of blocks (and which block to move) and *value function* refers to the aforementioned progress function.

A particularly appealing framework that can be applied in learning policy in continuous space is one of policy gradient methods. In a nutshell, these methods learn a *parameterized policy* by optimizing a target value using *stochastic gradient descent* methods. In this dissertation work, I will try different Monte Carlo methods, including REINFORCE (Williams (1992)) and some variances.

In this problem, I constraint my set of actions so that all of them could be simulated in a 2-dimensional environment. This is not really a requirement for the idea to work, but it allows me to quickly develop a 2-dimensional simulator to try different RL algorithms. An extension to 3-dimensional space by employing a more sophisticated physical engine simulator that can generate, possibly in parallel, multiple simulations, without visualization expense (as seen in game engines like Unity).

1.1.6 Evaluation interface

To evaluate, I will visualize the actions being performed in 2D and 3D simulation environment, which allows evaluators to judge whether the system has successfully learned the novel concepts. The 2-D simulator could be used to quickly evaluate the learned model. The 3-D simulator aligns better with human cognitive ability, allowing more precise and intuitive judgment, but is generally slower to setup.

The 2-D simulator is written with Python's *matplotlib* library, supporting simple human-machine interactions. Users can reset new starting configuration of objects in a 2-D environment, then observe learning agents to plan actions step-by-step. Beside using it for evaluation, it can also be used as a playground for interactive learning, as discussed in Subsection 2.4.1.

The 3-D simulator is based on Krisnaswamy's Voxicon Simulator (VoxSim Pustejovsky and Krishnaswamy (2014)) used for generating visual scenes from textual sentences. VoxSim is, in turned, developed using the popular Unity 3-D game engine. VoxSim setup has been configured specifically to test action simulations in a **Block World** environment, whereas a virtual humanoid agent named Diana can be commanded to move objects around on a flat table.

VoxSim, with the connection to a real-time gesture recognition using deep convolutional neural networks (Krishnaswamy et al., 2017) allows human-computer communication with generation and interpretations of multiple modalities, including: text, gesture and visualization. In fact, that creates a powerful framework for interactive and incremental machine learning. Though not discussed in this dissertation, VoxSim can also be used for interactive learning, i.e. users can gesture to Diana sequential locations of a block as a demonstrate of an action.

1.1.7 Explainable AI

As we will see, in Chapter 4, I use a deep learning model to learn an action recognizer. As typical to deep learning models, the model has a "black box" nature, i.e. it can hardly be explainable. That creates an issue of knowing whether we have actually learned action representations, or we have learned something else that helps on classification task. For example, deep learning methods on activity recognition has a tendency to

employ all channels from RGB-D input to predict action classes (Rahmani and Mian, 2016). While this method leads to a higher performance score, the classifier does not really capture action representation. For example, when there is an action like ”open a door”, can we tell that the learned model really capture the semantics of that action? Or it might just recognize the door observed in the scene? Or it might actually capture just the shiny door knob? We do not know. As long as the only target of a machine learning model is to optimize objectives on validate data, it is hard to understand and visualize the landscape of predictions that the model produces.

In recent years, there have been several efforts to make sense of machine learning models, or to demystify its black box nature. For example, uses of adversarial samples to “fool” deep learning model (Papernot et al., 2016) is one way explore the “landscape” of deep network models, to understand when they fail, how they fail and how to combat against this bad behavior of DNN.

Another, more explicit effort, is to push forward development of more interpretable machine learning models. For example, DARPA’s Explainable AI (XAI) initiative (Gunning, 2017) calls for creation of new machine learning techniques that piggyback on current high-performing but opaque models such as deep learning or random forests, but are trained to also optimize a meta-objective of being interpretable. Models of this type include models generating high-level and interpretable features automatically from deep network (Si and Zhu, 2013), models generating visual explanation for classification of images (Hendricks et al., 2016) etc.

Although this work is not directly related to the XAI initiative, XAI has been a recurring topic discussed in our lab. The development of our VoxSim and EpiSim (an in-house service for visualization of VoxSim epistemic model) is an effort to allow humans to peek into the machine’s brain. In that spirit, I propose that for action recognition, the most interpretatable model is the one that allows re-enaction of the action on a novel setup. The learned action representation is analogous to a *disembodiment* of action from the training observations, just to be incarnated in a novel situation.

1.1.8 Qualitative reasoning

Qualitative reasoning is first and foremost human’s logical reasoning strategy to cope with infinite amount of data. Our brain, though a very complex system with billions of

neurons, can still only process and memorize a finite amount of data. We, therefore, always have to make decision on impartial knowledge. Moreover, We do not usually need to remember exact values, simple binary or categorical distinction often work as well for us in our decision making process (probably excepts for financial matter). For example, we stop if the traffic light is red, we go if it is green. That tendency allows us to only need to memorize and reason on *qualitative* descriptions, reduced from full quantitative descriptions. We, therefore, only focus on some *landmark values* ((Bobrow, 2012)) - subset of infinite possible values, and disregard others. For instance, "freezing" and "boiling" are too natural landmark values of water temperature.

Qualitative reasoning, as applied in computer science, is a framework that applies the same qualitative principle in designing decision making machine. Think of a smart home system that allows you to set a rule "Turn off the heater when it gets too hot". You might have predefined what room temperature is considered "too hot" for you a while ago, but with that *landmark value* already fixed, you can communicate on a more meaningful level with this system, while disregard the mundane details of what exact temperature you have set.

Qualitative reasoning is strongly associated with natural languages. Before the invention of measurement devices, we can still talk about temperature. It could be freezing, cold, warm or hot. These *landmark values* are more subjective, fuzzy and personal (for example, a hot day for a Siberian person might be a freezing day for me), and language-dependent, but as long as there is a some conversational convention, it allows humans to effectively communicate and collaborate.

Spatial qualitative reasoning is a sub-field of qualitative reasoning, directly applied to embodied AI systems, such as robots. As embodied systems, they process inputs, plan actions and navigate spaces through reasoning on spatial information (Bhatt et al., 2011; Cohn and Renz, 2008). While robots can work on raw data, communication between robots and humans requires an intermediate level of representation, which spatial qualitative principles could be aptly used.

In the experiment part of this dissertation, we will see that qualitative features play an important role in facilitating learning of action model as well as to reduce planning space when we re-enact actions in a new environment.

Admittedly, qualitative reasoning applied in this work is more of a feature extraction method than a way to represent learned model in a way readable to human examiners. This representation is, however, better than a fully symbolic and human-readable representation but hard to work on for machine learning methods, and also better than a raw feature-based representation that generally require more training data to learn efficiently.

1.1.9 Action learning from videos

Human activities such as *running*, *sitting*, *eating*, and *playing sport* have been investigated in previous research, such as (Shahroudy et al., 2016). These human activities have significantly different motion signatures, therefore, classifying them is pretty simple. Recently, some studies have begun to introduce datasets with more complex activities, especially involving human-object interactions, such as cooking activities (Rohrbach et al., 2012), taking medicine (Koppula and Saxena, 2016), or human-human interactions, such as hand shaking (Ryoo and Aggarwal, 2010). More recently, (Li and Fritz, 2016) investigates the possibility of predicting partial activities using a hierarchy label space. These studies have gradually led to a more fine-grained treatment of event classification.

Actions can be learned atomically, i.e., entire actions are predicted in a classification manner (Shahroudy et al., 2016), or as combinations of more primitive actions (Veeraraghavan et al., 2007), i.e., complex action types are learned based on recognition of combined primitive actions. For the former type of event representation, there are quantitative approaches based on low-level pixel features such as in (Le et al., 2011) and qualitative approaches such as induction from relational states among event participants (Dubba et al., 2015). For the latter approach, systems such as (Hoogs and Perera, 2008), use state transition graphical models such as Dynamic Bayesian Networks (DBN).

Action classification using qualitative spatial methods has been discussed in a fair amount of work. (Suchan et al., 2013) use the Regional Connected Calculus (RCC5) adjusted for depth field with data also recorded by Kinect®Sensor. This work classifies events related to people moving, sitting, or passing each other. (Dubba et al., 2015) provide an interesting framework which gives a summary explanation for a sequence of observations by alternating between inductive and abductive commonsense reasoning. They

apply their method for two activity types from RGB capture, aircraft and truck movements at an airport, and human-object interaction.

As a precursor for this research, I have done some preliminary work on learning action representation from a movie dataset (Movie description dataset (Rohrbach et al., 2015)). My idea is simple: can we learn the representation of an action, such as "slide" from a dataset that has mapping between a short video snippet and an action description? We also want to compare that to the same verb "slide" represented in our semantic framework of VoxML (Figure 1.3). It is attractive if we could use some readily available and large dataset like that to facilitate learning representation of actions. However, text descriptions from this dataset are highly stylistic, and scenes described with the same verb "slide" are highly varied that they are hardly comparable to our prototypical semantic definition. Following are some examples of text descriptions from the movie corpus. Without the visual scenes, it is still quite obvious that the nature of actions described in these sentences are very different:

- He wipes out and his bike slides underneath the vehicle.
- Someone slides across a white limo's hood.
- As the car slides into a turn, he loses control and spins out completely.

$$\left[\begin{array}{l} \textbf{slide} \\ \text{LEX} = \left[\begin{array}{l} \text{PRED} = \textbf{slide} \\ \text{TYPE} = \textbf{process} \end{array}\right] \\ \text{TYPE} = \left[\begin{array}{l} \text{HEAD} = \textbf{process} \\ \text{ARGS} = \left[\begin{array}{l} \text{A}_1 = \textbf{x:agent} \\ \text{A}_2 = \textbf{y:physobj} \\ \text{A}_3 = \textbf{z:physobj} \end{array}\right] \\ \text{BODY} = \left[\begin{array}{l} \text{E}_1 = grasp(x,y) \\ \text{E}_2 = [while(hold(x,y), \\ \quad while(EC(y,z), \\ \quad move(x,y)))] \end{array}\right] \end{array}\right] \end{array}\right]$$

FIGURE 1.3: Sample voxeme: [[SLIDE]]

1.1.10 RGB-D dataset

Because the main part of this work is on learning action from RGB-D dataset, it would be a deficiency if we do not cover some previous work on RGB-D dataset and learning

with depth-field data. A detailed review of RGB-D dataset is provided by (Firman, 2016). In this subsection I will only give brief introduction to a few relevant datasets.

Cornell 3D activity datasets: This is a very interesting dataset with an intention of capturing not only human activities but also to recognize object affordances for robots to interact with objects (Koppula et al., 2013). This is I believe the first effort to using human activity dataset with the intention of teaching robots to figure out how to interact with objects and plan actions. This dataset has 120 activity sequences of ten different high-level activities, with activity-subactivity annotation, as well as affordance labels on objects.

Robotic grasp dataset: This is a static RGB-D dataset that has bounding-box annotation at grasping positions of objects. This dataset and the learning methods from (Lenz et al., 2015) allows real robots to operate on novel objects. Learning how to work with objects in the real world is a very complex task. Even for a primitive kind of action as "grasp", the problem is not yet solved. Even if robots can pick up objects, the grasping point might not be the correct grasping position to allow further interaction with objects (e.g. where to grasp a water jug to pour to a cup; where to grasp a pen to write).

Watch-n-Patch ((Wu et al., 2015)): captures human activities, each containing a sequence of sub-actions in which they are dependent in a causal-chain manner. For example, a complex activity such as *warming milk* is a sequence of subactions such as *fetch-milk-from-fridge*, *microwaving*, *fetch-bowl-from-oven*, *put-milk-back-to-fridge*. The authors of this dataset also proposed an unsupervised method to auto-segment long activities into sequence of sub-actions.

Manipulation Action Dataset ((Aksoy et al., 2015)): a dataset of 8 activities *push, hide, put, stir, cut, chop, take, uncover*. The authors of the dataset also use a sequential model called semantic event chain (SECs), that learns a probabilistic graphical model with states being simple RCC relations (*touching*, *not touching* and *absence*).

Together with this dissertation, I will also provide two datasets based on RGB-D inputs: one is used for fine-grained activity recognition task in Chapter 4, and one for action reenactment task in Chapter 5. Notice that the datasets published online are tracking coordinates of objects and humans from RGB-D datasets, while original and raw RGB-D data are not available for privacy reasons.

1.1.11 Humans' action learning capability

To develop AI systems that can learn from humans, we could look into the way babies learn simple skills from adults for inspirations. Baby humans are the most powerful learning agents in the world, with the ability to learn to grasp multiple types of objects, then moves on to use different kinds of tools in just the first few years of their lives.

Action and language are connected through language of actions, mostly reflected by verbs and spatial adjuncts. There is a strong association between verbs and actions in infantile learning. For example, it has been shown that new verbs are learned more quickly when the action is named just before the infant performs it (Tomasello and Kruger, 1992). Also new verbs are learned more effectively if they also perform the action instead of just watching it (Gampe et al., 2016). Toddlers also show sensorimotor activity (reflected by electroencephalography (EEG)) during auditory processing of action verbs (Antognini and Daum, 2017). These clues, combining together, suggest associative learning happens between visual perception, sensorimotor and linguistic faculties.

To understand how infants learn action skills, first let see how they learn simple reach and grasp actions. These are among the first object-interaction actions of infants, but quickly they would become tool manipulation skills. We know that infants start to explore the world in a random manner. While exploring the world, a child sometimes randomly makes contact with objects (under 4 months). A primitive infantile reflex called *palmar reflex* causes his finger to reflex that creates an involuntary grasp (Kuipers, 2010). Grasping causes further actions, such as "picking it up", "sliding it away" etc. Between 7 and 12 months, after mastering picking up objects, the child could start to explore them in details, by passing them between his hands, rotating them to check from different angles, or selecting the best grasp. As early as 12 months, he starts learning complex object compositional relation, such as *container-containee* relation (Garner and Bergen, 2006). After that, more oriented activities such as playing toys allows children to explore more diverse and complex compositional characteristics of objects, such as learning of the distinction between functional parts of objects (e.g. parts corresponding to objects' *Gibsonian* versus *telic* affordances (Pustejovsky et al., 2017)).

Mapping that to our learning discussion, we can see the analogy between infantile learning and reinforcement learning, or in a broader sense, an *exploration-exploitation* learning mechanism. Exploration allows learning of new knowledge without direct teaching

by incorporating perceptive inputs while exploring searching space. Exploitation allows applying learned models to efficiently perform actions.

Infantile action learning is normally carried out by pretending plays, for example, using a brush on their hair or drinking from a cup. These are the effect of both mirroring from adults (whether or not adults intentionally teach children to use tools), and mastering of tool functionality. As pointed out by (Gergely et al., 2002), pre-verbal toddlers aged 14 months interact with tools by imitating adults performing actions, but with complex inferential process, rather than just emulation. They found that toddlers imitate adults switching a light bulb by using forehead if the adult performers have their hands free, but they would not imitate the forehead action if the adult hands are tied. It suggests that children at this age start to have a distinction between the goal and the mean of an action.

In Chapter 5, we can see a reflection of that distinction in the form of actions we teach AI agents. Actions, as well as events, extend a duration of time, and without further instructional input, AI agents might not be able to reason whether the whole action is important to learn (the mean), or only the final result (the goal) is needed. We will see that in the experimental setup, we will include both types of actions to find out if AI agents can learn them using the same framework.

1.2 Outline

This dissertation is structured as follows: Chapter 2 describes the scaffolding learning framework, with a brief introduction to my capture and annotation guidelines, feature extraction methods, and machine learning models that are common for the remaining chapters; Chapter 3 is dedicated to introduce my annotation tool named ECAT, used for annotation events captured from RGB-D inputs; Chapter 4 describes an action recognizer, that I have developed to study appropriate representation for action learning; Chapter 5 is the main focus of this dissertation, describing a methodology for learning agents to learn to perform a set of primitive actions from observing real human demonstrations, using sequential modeling and reinforcement learning; Chapter 6 flips the problem setup in Chapter 5 on its head, teaching learning agents to perform chain of actions directly by "neural" translating from textual instructions, using Sequence to

Sequence models. We will close this thesis with a short conclusion in Chapter 7 that include a road map of my dissertation work and future directions.

Chapter 2

Learning Framework

The learning framework for this research rests on the mapping among linguistic, visual and programmatic representations of actions. Inputs that robots could take in include text or spoken instructions from humans, visual features (including RGB, depth-field and infra-red etc.). Mapping between linguistic and visual representation allows robots to recognize and describe actions, whereas an addition step of mapping from linguistic and visual representation to programmatic representation (or program form of action) allows robots to perform the action given human linguistic commands.

Linguistic event representation in my framework is modeled as a verbal subcategorization in a frame theory, a la Framenet (Baker et al., 1998), with thematic role arguments. In addition, I also account for *extra-verbal factors*, i.e. the aforementioned adjuncts describing path of motions. Therefore, I consider *A moves B toward C* and *A moves B around C* as different event types and aim to learn each event type as a separate *atomic* action. A future work that include learning *manner* of action would allow further distinction such as between *A rolls B toward C* vs *A slides B toward C* and combination of path and manner aspects of actions.

Our visual event representation comprises visual features extracted from tracked objects in captured videos or virtual object positions saved from a simulation environment. Both types of feature represent information visible to humans and observable by a machine in object state information. Using these data points and sequences, machines can observe humans performing actions through processing captured and annotated videos, while humans can observe machines performing actions through watching simulated scenes.

Programmatic event representation can be based on formal event semantics or on features that can direct simulated or robotic agents to perform an action with an object of known properties. From a human perspective, the distinction between learning to recognize and learning to perform an action might be obvious. However from a machine's perspective, these two tasks might require different learning methods. My work aims to demonstrate that given an appropriate framework, it is feasible to map between them, in a manner similar to the way humans actually learn: by matching actions to observations.

This chapter could be considered an extension of Chapter 1, but we will focus on various machine learning techniques and environmental setups used in this work. In particular, we will first discuss environment for capturing of learning actions in Section 2.1. We will then move on to discuss feature extraction framework used to process the captured data in Section 2.2. We will briefly discuss a general framework of sequential modeling using Long Short Term Memory method in Subsection 2.3.1, discussing its architecture, practices and advantages over other sequential models. We will also cover two simulation/visualization modules, a Python 2-D simulator implemented to quickly run multiple reinforcement learning episodes (Subsection 2.4.1) and a 3-D visualizer extended from VoxSim (Krishnaswamy and Pustejovsky, 2016) for more realistic demonstrations (Section 2.5).

2.1 Capture environment and annotation guidelines

In this work, I use my Event capture and annotation tool (ECAT) for capturing and annotating actions. We will separately discuss ECAT in chapter 3. In this section, we will only focus on setup for capturing actions:

1. ECAT is installed on DARPA CwC Apparatus (Tamrakar, 2015). It includes the following components: a square table about 5' x 5', one flat screen mounted on one side of the table, and two Microsoft's Kinect® devices mounted on two sides next to the side mounted the screen (Figure 2.1).

2. Blocks are 6" cube blocks marked with ARUCO markers. The blocks belong to the original CwC setup, each printed with a brand name, such as *Pepsi*, *Stella Artois*). The ARUCO markers are stuck on different sides of the blocks (Figure 2.2).

3. The apparatus is so located that performers can move freely around the performing table, and can observe the captured scenes on the flat screen, to ensure that they do not move out of the field of view (FoV).

FIGURE 2.1: DARPA CwC Apparatus

Data capture guidelines are as follows:

1. Create a separate project for an action type. Under each project, for each performer, capture a long session of the performer doing multiple instances of an action.

2. The performers are given the linguistic description of an action, such as *slide A around B* or *make a row from three blocks*, and are asked to perform according to their understanding of the action.

3. After each demonstration, the performers move the blocks to some initial configuration for the next demonstration. The performers are asked to vary the initial locations of blocks in each demonstration as much as they can, to vary their position with regard to the table, and change their performing hands. These strategies are mainly to add variation into training data.

4. Data capture is carried out with one person being the action performer, and one person helping with starting/stopping capturing sessions. The helper also watches the live capture to make sure that the performer is still tracked and objects are not pushed outside of the capturing FoV.

FIGURE 2.2: Block with ARUCO markers

Annotation guidelines are as follows:

1. Annotators first mark the surface of the table throughout multiple frames of a session, by drawing a polygon boundary. Then the annotators click on *Generate 3D* to generate the 3-D planar equation for the table.

2. Annotators create or load ARUCO markers prototype file. This file maps a block name to its ARUCO markers. In turn, each ARUCO marker is represented by a 5x5 bitmap.

3. Annotators click on *Detect objects* for the ARUCO marker detection algorithm to run. The algorithm first maps the marker it sees on a face of a block to an Id (e.g. ARUCO 129). The ARUCO marker is, in turn, mapped to a block name.

4. Annotators click on *Map to 3D* to map objects into 3-D coordinates using the captured depth-field.

5. Annotators annotate actions by marking action spans (beginning and end frames) as well as adding a description of the action (e.g. The performer slides Pepsi around Stella Artois) while watching the replay of a session. Annotators are asked to mark the beginning of an event when the object starts to move, and the end when the performers stop moving object and release their grasp.

2.2 Feature extraction module

To facilitate event classification, it is necessary to present events in a learnable format. This introduces the question of how to represent events, namely the difficulty in defining their temporal and spatial extensions, as well as the difficulty in selecting an observational perspective.

Moreover, the feature representation for actions needs to be shared between the real captured-data used for training and the artificial data generated from the simulator. The feature space in these two types of data are quite different: the real captured data is fuzzy and sometimes includes noisy outliers (shown as flickers when visualizing the captured session); the simulated data is smooth and interpolated frame-by-frame from planned actions.

This leads to the difficulty in finding an appropriate feature set as common abstraction to bridge these two kinds of data. I enumerate some different feature types I have experimented in this work. They can be generally divided into two sets: quantitative and qualitative, whereas qualitative features are calculated based on quantitative features.

Both quantitative and qualitative features of the captured data are calculated based on projecting the blocks' coordinates on 2-d surface of the apparatus's table. The details

of the projecting algorithm is presented at Algorithm 1.

Input: 3-D coordinates of markers for each block (one block might have multiple visible markers)
Size of the block (different from size of a marker) **Purpose**: Estimate a 2-d square for each block on projecting surface
Result: Centroid and angle of projected 2D square
Estimate plane equation P of the table;
for *Each block* **do**
    Select the marker that has the most number of corners, and the largest size (most clearly observed marker) ;
    Estimate plane equation Q of the marker face;
    if *the cosine product of P and Q norm vertors* < 0.5 **then**
        Calculate the square T that perpendicular to Q, have the correct size, and have the edge on the middle line of marker P;
        Project T on P (T_P);
    else
        Rescale and project the marker on P (T_P);
    end
    Use a single point on P as the (0,0) origin, estimating the transform of the 2-d square from (T_P);
end

Algorithm 1: Algorithm to project blocks on 2-D simulator

2.2.1 Quantitative features

Firstly, in each demonstration of an action, I rank the objects by their salience, i.e. the object that moves the most is the most salient. Quantitative features include the followings:

1. Location of the centroid of a block, and the block's angle made with regard to the Ox axis (the transform as estimated from Algorithm 1).

2. The difference between transforms of the most salient object to the second salient object.

3. The gradient of the transforms of the objects, i.e. the difference between two frames.

While quantitative features are not the focus in my discussion, we need to include them to provide a more comprehensive picture, and to give comparison to the qualitative methods.

2.2.2 Qualitative features

Qualitative spatial reasoning (QSR), a sub-field of qualitative reasoning, is considered to be akin to the way humans understand geometry and space, due to the cognitive advantages of conceptual neighborhood relations and its ability to draw coarse inferences under uncertainty, and also analogous to the way humans map from continuous space of spatial perception to discrete space of linguistic description (Freksa, 1992, 1991).

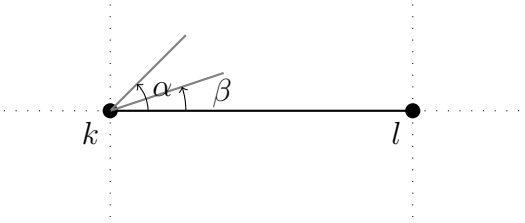
It is also considered a promising framework in robotic planning (Cohn and Renz, 2008). QSR allows formalization of many qualitative concepts, such as *near*, *toward*, *in*, *around*, and facilitates learning distinction between them (Do and Pustejovsky, 2017b). The use of qualitative predicates ensure that scenes which are semantically close have very similar feature descriptions.

Moreover, from a machine learning perspective, as pointed out by Yang and Webb (2009) and Jiang et al. (2017), qualitative representation is a method of discretization, which makes data sparser, therefore easier to learn. Especially when taking the difference between features of two adjacent frames, as a qualitative feature strongly distinguishes between 0 and 1, the effect of sequential change is more pronounced.

There is a extensive literature supporting the use of discretization for feature embedding. Yang and Webb (2009) shows that discretization is equivalent to using the true probability density function. More recently, Jiang et al. (2017) has used this method for classification of GPS trajectories. They studied three different approaches for discretization, including *equal-width binning*, Recursive Minimal Entropy Partitioning (RMEP) (Dougherty et al., 1995) and fuzzy discretization (Roy and Pal, 2003). Their finding is that the *equal-width binning* approach is both simple and effective, so I used this approach for quantization of both distance and orientation.

QSRLib (Gatsoulis et al., 2016), a library that allows computation of Qualitative Spatial Relations and Calculi is employed to generate qualitative features. In particular, I use the following feature types from QSRLib:

- CARDINAL DIRECTION (Andrew et al., 1991) a.k.a QSRLIb *cardir*, transforms compass relations between two objects into canonical directions such as North, North East etc. In total, this qualitative relation gives 9 different values, including one where two locations are identical.
- MOVING or STATIC (QSRLIb *mos*) measures whether a point is moving or not.
- QUALITATIVE DISTANCE CALCULUS (QSRLib *argd*) discretizes the distance between two moving points. Here I discretizes the distance between two centers of two squares.
- QUALITATIVE TRAJECTORY CALCULUS (Double Cross) a.k.a QSRLib *qtccs*: QTC_C is a representation of motions between two objects by considering them as two moving point objects (MPOs) (Delafontaine et al., 2011). The type C21 of QTC_C (implemented in QSRLib) considers whether two points are moving toward each other or whether they are moving to the left or to the right of each other. The following diagram explains this:



QTC_C produces a tuple of 4 slots (A, B, C, D), where each could be given either -, + or 0, depending on the angle α. In particular, A and C explains the movement of k toward l, and vice versa for B and D. A is $+$ if $\alpha > -90 \wedge \alpha < 90$, $-$ if $\alpha > 90 \wedge \alpha < 270$, and 0 otherwise, i.e. whether k moves toward l or away from l. C is $+$ if $\alpha > 0 \wedge \alpha < 180$, $-$ if $\alpha > 180 \wedge \alpha < 360$, and 0 otherwise, i.e. whether k moves to the left or right of l. QSRLib also allows specification of a *quantisation factor* θ, which dictates whether the movement of a point is significant in comparison to the distance between k and l.

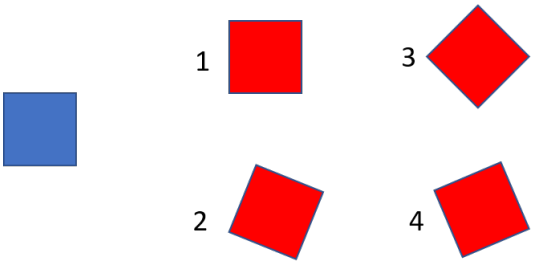


FIGURE 2.3: Quantizing object rotations

The followings are the qualitative features that I would use in this research:

- F1: cardinal direction between two centers
- F2: qualitative distance between two centers
- F3, F4: $\Delta(F1), \Delta(F2)$
- F5, F6, F7, F8: 4 QTC_C slots
- F9, F10: quantized orientations of first and second object, as described in Figure 2.3.
- F11: F9 - F10
- F12, F13: $\Delta(F9), \Delta(F10)$

There is some correlation between these features. In addition to the features that are derived from other (like the Δ features), some features have similar meaning but different implementation. For example, F4, F5 and F6 are very similar. The difference is that F4 is only sensitive to the change of quantized distance, while F5 and F6 is generally much more sensitive. F5 and F6 distinguish which object is moving, while F4 does not.

2.3 Supervised machine learning module

The main supervised learning method used in this dissertation is Recurrent Neural Network, with a specific version of Long-short term memory (LSTM) (Hochreiter and

Schmidhuber, 1997) as the processing cell. LSTM has found utility in a range of problems involving sequential learning, such as speech and gesture recognition, human activity recognition and movie description generation. While I have given a short introduction to sequential modelling methods in Section 1.1.4, in this section I will address the architecture of LSTM I used in this research and common practices needed to learn a successful LSTM model.

Even though LSTM has existed for a long time (more than 20 years!) and has somewhat become an industrial standard for sequential modeling, its working principle is arcane and its model is hard to interpret, as common to most, if not all, deep learning models. While it is hard to explain why LSTM, as a specific flavor of Recurrent Neural Network, has been so successful over the years, there are a few important points worth recapitulating in respect of the motivation for the specific form of LSTM architecture (Section 2.3.1.1). Over the time, use of LSTM models has created some common practices, which I will summarize in Section 2.3.1.2.

Moreover, I will also provide a brief reviews of a few other common sequential models, including their advantages and disadvantages (in comparison with neural network based model like LSTM), as well as their appropriateness for the problem at hand. Two popular kinds of models would be addressed in Section 2.3.1.3, discrete hidden state models such as Hidden Markov Model (HMM) and Chained Conditional Random Field (Chained CRF), and continuous state models such as Linear Dynamical System (LDS). Note that another sub-type of CRF (Tree-CRF) will be discussed later for constraining classification outputs, but not for sequential modeling.

2.3.1 LSTM

2.3.1.1 Architecture

LSTM, first and foremost, is a flavor of Recurrent Neural Network. Recurrent Neural Network is simply just a neural network that persist some **state** or **memory** during learning, while simple Feed Forward Neural Network (FFNN) does not keep any memory, just a set of **parameters**. RNN is, therefore, more powerful and generative than FFNN, which can be considered as RNN in one step. Following is an example to illustrate this difference. Let's say you want to learn a Part-of-speech tagger using neural network methods. The FFNN method would use some features of the current context

window (word forms, suffix of the current words and some words before it) as feature vector to feed into a neuron network that predicts the output POS. In contrast, RNN would keep another memory vector, accumulating information from the beginning of the sentence (and also from the end if you use Bidirectional RNN); as a result, you can capture long-distance dependency, such as if you have a left quote somewhere in a sentence, there should be a right quote as well. For POS tagging task, it might not be straightforward what is the benefit of using sequential modeling, but the Bi-LSTM model is among state-of-the-art models, outperform FFNN method by a small margin (Huang et al., 2015). A state update of classical RNN is represented by Equation 2.1. State from the previous time step is concatenated with input X_t and then passed through a neuron with $sigmoid$ activation function. W is the weight and B the bias matrices.

$$H_t = \sigma(W * (H_{t-1} : X_t) + B) \tag{2.1}$$

LSTM is a RNN that has a special calculating unit called LSTM cell, depicted at Figure 2.4. In a nutshell, an LSTM cell has two states, one is the *memory state*, denoted by C_t, and one is the actual output vector denoted as H_t. Outputs from 4 neurons are called gates: I is input gate, G is new input gate, F is forget gate and O is output gate. Updating the memory state C and the output cell H is given by the following equations

(From (Zaremba et al., 2014)):

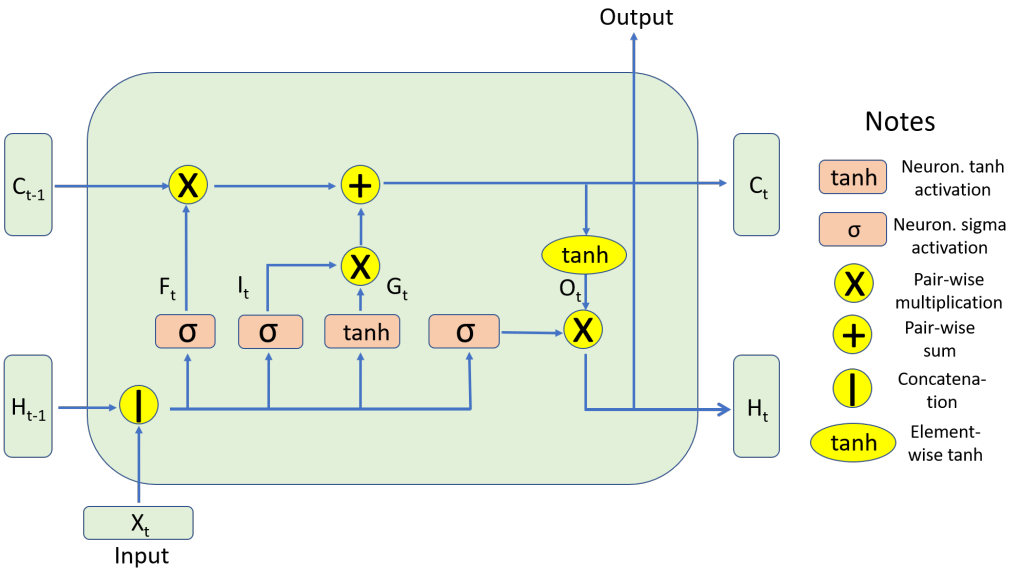


FIGURE 2.4: Architecture of an LSTM node. Note the color code for simple (pairwise, or elementwise) operations is yellow, color code for neural operation is light orange.

$$C_t = \underbrace{F_t \odot C_{t-1}}_{\text{Long}} + \underbrace{I_t \odot G_t}_{\text{Short}} \tag{2.2}$$

$$H_t = O_t \odot tanh(C_t) \tag{2.3}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.4}$$

The intuition for the equation 2.2 is that at each time step, we want to keep a part of the memory at the previous step, scaled by a scaling factor at **F gate**, plus a new value for the memory, calculated at **G gate**, also scaled by another scaling factor at **I gate**. Because F and I are two scaling factors, their component values are between 0 and 1, therefore they are given by two neurons with $sigmoid$ activation function. **G gate** calculates a new candidate value for the *memory state*, therefore it is given by a neuron with $tanh$ activation function (value ranged from -1 to 1). The advantage of LSTM model over vanilla RNN is the use of the *memory state* C. Without it, the model outputs directly values from **O** (from 2.3), and become vanilla RNN model.

You might ask why we need a product of two gates I and G as candidate value for new *memory state*, or why we multiply output gate value O with $tanh$ activation of *memory state*. Fortunately, the arcane architecture of the LSTM cell has been demystified by the work of (Jozefowicz et al., 2015). They proved that not all the gates in LSTM is strictly

necessary. Except for forget gate, you can drop any other single gate without sacrificing too much of performance.

In short, technically, the most important characteristic of LSTM model is the use of *memory state* separated from the output as two states of LSTM cells. Intuitively, cell output is *predictive*, i.e. trained to predict some labels, whereas *memory state* is *explanatory*, i.e. explain why you predict such label. Also very important is its breakdown into an update to *long-term* memory and a new capture of *short-term* memory (Equa. 2.2).

2.3.1.2 Common practices

Over the years, development of LSTM has picked up a few common practices that I used in this research. The followings are some good practices for various purposes, including practice to process inputs, to combat over-fitting, to tune hyper-parameters etc.:

1. Dense layer at the end of output: Output straight from LSTM cell H_T has the same dimension as the *memory state* dimension, so a common practice is to pass it through a dense linear layer. For classifier, the output is a logarit probability vector, its dimension is the number of classes. For regressor, it could be just one value.

2. Chunking of input sequence: A common practice when generating inputs for LSTM network is to pad input sequences so that all feeding sequences have the same length. For task such as categorize of text inputs, each sentence would be truncated or padded to a fixed length (e.g. 50 words) before being fed as an sequence input. For the problem at hand, because we have a very long sequence of continuous input, the most appropriate method is to chunk the long input sequence into multiple shorter sequences, each sequence would be considered as one input sample. Notice that we could not achieve independent and identically distributed (i.i.d) property for this dataset because all input sequences are captured in a small set of long videos. However, by randomly permuting them before training, the problem of temporal correlation among samples is minimized.

3. Multi-layer LSTM: As typical in deep learning methods, when you go deeper, the model performs better, with a trade-off of longer training time, and larger model

to keep. For LSTM, go deeper means stacking multiple layers of LSTM cells on top of each other. The output from H_t is fed as input to the LSTM cell on the next layer, while we are keeping multiple *memory states*, each for a layer of LSTM.

4. Retaining of *memory state* between training (or testing) mini-batch: While we call the state vector C_t as long-term *memory state*, whether it can capture long term dependency between input sequences or it just capture context from beginning of each sentence depends on another property of the model, called *statefulness*. A stateful model persists the memory states between consecutive mini-batch update, whereas a stateless model resets the memory states after each mini-batch. In practice, the matter of using stateful LSTM model is coupled with the matter of shuffling training samples. As I shuffle samples before training, I also use a stateless model.

5. Dropout: Dropout is a common method in deep neural network training to address the problem of overfitting(Srivastava et al., 2014). In training, in each mini-batch, a percentage $1 - p$ of network nodes are chosen randomly to be skipped, so that only the parameters of p remaining nodes are updated. In testing, all nodes are present, but the activation output is scaled with p. For deep feed-forward neural network, one can choose to thin out different layers with different dropout rates. For LSTM model, as pointed out by (Jozefowicz et al., 2015), only two vertical edges in Figure 2.4 can be applied dropout: Input edge from X_t to concatenation operator, and output edge from H_t. In total, information flow from the input to the output of the LSTM can be corrupted $L + 1$ times where L is the number of stacked LSTM layers, independent of the number of steps in the sequence. Also, as theoretically proven by (Gal and Ghahramani, 2016), we need to use the same dropout operator (exact same set of dropout nodes) for all learning steps.

6. Training method: LSTM, similar to other deep learning models, typically use an online-learning method, such as Stochastic gradient descent or Adam (Kingma and Ba, 2014), to minimize an objective by continuously calculating the objective and updating the parameters on a portion of the training data.

7. Hyper-parameter grid searching: In training a neural network model, there are a number of hyper-parameters that are difficult to set right from the beginning. By grid searching over some different values for these hyper-parameters, we can select the best combination. They include $keep\ prob$ (the percentage of nodes remained in dropout layer), $hidden\ size$ (the size of feature vector in the *memory*

state C_t), $num\ layers$ (the depth of the LSTM stack), $learning\ rate$ (speed of changing model parameters), $training\ algorithm$.

2.3.1.3 LSTM versus other sequential models

In this section, I will pit LSTM against two classical sequential models: discrete State Markov Models and Linear Gaussian State Space Models (LG-SSM) (RNN and LSTM are, by no mean, recent, as we have discussed previously, but enter mainstream focus only after the advance of deep learning methods).

Firstly, discrete state Markov Models, such as Hidden Markov Model (HMM), its discriminative sibling Maximum Entropy Markov Model (MEMM), and their cousin Chained CRF, are very similar to RNN in their nature. MEMM is analogous to a classical RNN which has a hidden state represented by an one-hot encoding vector (vector that is zero, except for the position corresponding to an active state), and the neuron has a **hardmax** activation function (hardmax function turn a probability distribution to an one-hot vector by setting the value at max category to 1, and other to 0). There are two disadvantages with this set of methods. The first problem is that while they can be aptly used for label tagging (sequence to sequence), it is not trivial to turn them into a classifier or a regressor (sequence to one model). Using HMM method as a classifier for multiple categories requires training K different HMM models, one for each category, then selects one that maximize the posterior. It also seems quite hard to create a regressor (like my progress function) from HMM. To the best of my knowledge, there is no popular way to implement a regressor using HMM method. The second problem lies in the fact that we typically want to choose HMM because of its explanatory power. It is always nice to be able to name the hidden states, and has the HMM model keeping track of which state you are in. However, when we need more complex models, which means we have to increase the number of hidden states v, the number of parameters is also dominated by a squared term of v (transition probability table), which is the same as RNN models (we will see their number of parameteres soon). With the same number of parameters, the use of *hardmax* function in these discrete state models limits their learning ability, in exchange for an obscure interpretability.

Another family of popular sequential models is linear-Gaussian state space models (LG-SSM). Equations from 2.5 to 2.8 represent an LG-SSM. In this model, we have z_t is the hidden state, u_t is an input or control signal and y_t is the observation. Notice the

similarity between equations of LG-SSM and the classical form of RNN (Equa. 2.1). The difference is the explicit reification of error terms in LG-SSM, in particular, as two Gaussian distributions.

$$z_t = A_t z_{t-1} + B_t u_t + \epsilon_t \tag{2.5}$$

$$y_t = C_t z_t + D_t u_t + \delta_t \tag{2.6}$$

$$\epsilon_t \sim \mathcal{N}(0, Q_t) \tag{2.7}$$

$$\delta_t \sim \mathcal{N}(0, R_t) \tag{2.8}$$

The method to filter on LG-SSM (i.e. to predict the hidden state z_t given history of $u_{1:t}$ and $y_{1:t}$) is to use the famous Kalman filter (Bishop et al., 2001). To train the parameters of A, B, C, D, Q, R of LG-SSM for the problem at hand, where we have training samples of the input signals u_t and observed outputs y_t, without knowing the hidden states z_t, the well-known method is to use Expectation Maximization (analogous to the Baum-Welch algorithm for HMM). The limitation of linear-Gaussian SSM is that it can only capture linear relationships between u_t, y_t, z_t. For example, in its application on object tracking problem (e.g. tracking a controlled drone), u_t is the signal sent to drone (left, right, up, down), y_t is observed location and velocity of the drone (with some noise of recorded device) and z_t is the real location and velocity of the drone. y_t is, therefore, just $z_t + \delta_t$ whereas $z_t = z_{t-1} + u_t + \epsilon_t$, When we could not make this assumption, LG-SSM would not perform well.

The number of parameters for LSTM model: A single-layer LSTM model with input dimension i, cell size (which is also memory size and output size) is o (notice that dimensions of C_t and H_t are the same), has $4 * (i + o + 1) * o$ of parameters. A two-layer model with the same hidden size has $4 * (i + o + 1) * o + 4 * (2 * o + 1) * o$ parameters (outputs of the first layer become inputs for the second layer). For example, my final configuration used for all action classes has $i = 8, o = 200$, and 2 layers, so the number of parameters is 488000, dominated by a squared term of cell size. That even does not include a dense layer connecting the output of LSTM cell to the target output, such as the progress value (1 output) or action class (5 outputs). Superficially it might sound like a disadvantage of LSTM model, as the number of parameters might even be larger than the number of training samples, but it has become a standard in deep learning community that you can use an over-complete model, that can easily overfit your data and regularize it, e.g. by minimizing validation error rates. The reason for

this phenomenon is not well understood, but it has been observed in other classical machine learning algorithms as well (Belkin et al., 2018).

2.3.2 Progress learner

"Learning from demonstration" framework is one of the *frontier* reinforcement learning topics, as discussed in (Sutton and Barto, 2018, Chapter 17), which means that there is no known reliable methods to learn them. This problem, in particular, has actually goes beyond the scope of Markov decision process (MDP) formalization. A default assumption in classical MDP is that at each time step, the subsequent action is defined by a *fixed* probabilities dependent only on the preceding state and action. In the problem at hand, there is no defined way to summarize the "shape" of the progressing action.

There are typically two approaches to learn behaviors from experts' demonstrations. One is supervised learning (for example, this work on learning to draw Chinese character (Zhang et al., 2018)), another is extracting a reward signal from expert's behavior for reinforcement learning. In this research, we will use a reward signal produced by training recognition model (cf. "inverse reinforcement learning" (Abbeel and Ng, 2004)).

Inputs are sequence of feature vectors taken from capture-data or from the simplified simulator and outputs are a function that correspond to the progress of an event. In particular, it is a function that takes a sequence S of feature vectors, current frame i and action e: $f(S, i, e) = 0 \leq q_i \leq 1$

The training set of sequential captured data is passed through an LSTM network, which is fitted to predict a linear progressing function. At the start or outside of an event span, the network produces 0, whereas at the end, it produces 1.

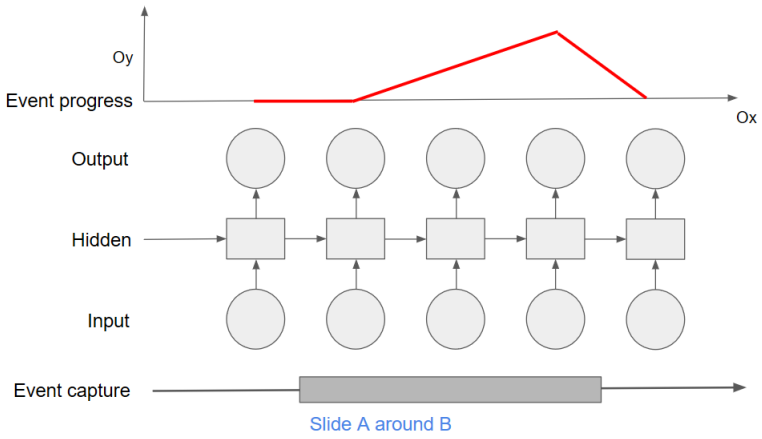


FIGURE 2.5: LSTM network producing event progress function

2.4 Simulation module

This module encompasses both a 2-D simulator that can quickly generate multiple demonstrations of actions, as well as multiple algorithms to learn the best policy to generating an action. In implementation, it is also a wrapper around the predictors of the aforementioned supervised machine learning module (Section 2.3), and also around the feature extractors (Section 2.2).

Therefore, in my implementation, this module simulates the whole mechanism of a virtual agent's faculty of action learning. Section 2.4.1 gives a summary of the 2-D simulator, including its components and functionalities. Section 2.4.2.1 describes searching algorithms that search over the space of trajectory in a heuristic manner, however these algorithms will have to search over again for any new configuration. Section 2.4.2 sketches out some reinforcement learning algorithms that can generate a strategy for new situation without searching over the entire space.

2.4.1 2-D Simulator

For the updating loops in my reinforcement learning algorithm, I want to simulate observational data faster than real-time simulation for effective computation. As a real-time, graphic-heavy simulator, Unity engine is not feasible for this task. While being aware

of a few other physical simulation environments such as Gazebo[1], but as I do not focus on physical constraints in this study, I implemented my own simplified simulator in Python.

My set of learnable actions is limited to ones that can be easily approximated in 2D space. 3D captured data is transformed into simplified simulator space by projecting it onto a 2D plane defined by the surface of the table used for performing the captured interaction. My 2D simulator has the following features:

- It is equipped with aforementioned feature extractors. These extractors are used to generate features on a frame-by-frame basis from the simulator.
- It includes the aforementioned progress learner (Section 2.3.2) for each action type. This allows us to calculate an accumulated reward function after a move has been carried out.
- Each object is represented by a polygon (or square), and attached with a $transform$ object that stores its position, rotation, and scale.
- The space is so constrained that objects do not overlap, and objects could not be located outside of a square rectangle (table boundary).
- Object could be move to a new location (change of the attached *transform* object), by sending action command to the simulator. If the action is an illegal move (moving to outside of the table, or to an overlapped location), action is recorded, but not carried out.
- Speed can be so specified that object movement can be recorded as a sequence of feature vectors interpolated frame to frame.
- It supports step-by-step visualization of a sequence of moves, or locations of objects can be interpolated and the movement can be recorded into video. Therefore it provides a simple visualization and debugging environment.

In details, the 2-D simulator is implemented with the following components:

[1] http://gazebosim.org/

- An environment simulator class that supports adding of objects at specified locations. A movement command can be sent to the environment to change location of one object, as far as: a. destination location is not blocked, b. assuming that the object is moved with the original orientation, and only changes its orientation at the end of the movement, the corridor of space needs to be clear for movement to happen.

- An action environment class that inherits *gym.Env*, a class from OpenAI Gym (Brockman et al., 2016) Python package. It is basically an adapter for reinforcement learning methods to the aforementioned simulator, providing functionality to initiate environment with random configurations, and support recording and traversing through history of each episode.

- 2-D Visualizer/debugger that can show an episode to an interactive python notebook, or save it down as a video. It can also be used to visualize the captured data as they are projected onto 2-D space.

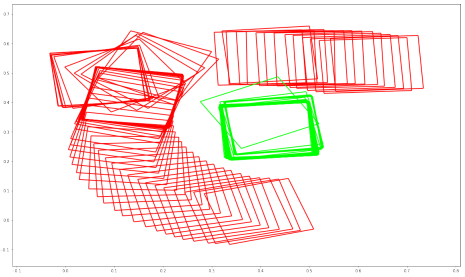


FIGURE 2.6: An recorded demonstration of "Move A around B" projected onto the 2-D simulator. A is projected as a red square, B as a green square. In this image, the simulator is in offline mode, and only used to show the trajectory of a recorded demonstration.

Interactive mode: The visualizer could be switched to an *interactive* mode, in which the visualizer shows the demonstration step by step, and users can click on a Next button to move to the next state, or Prev to back to the previous state. At each time step, the simulator uses the greedy algorithm presented at 2.4.2.1 to plan the next action.

At any point, users can decide that the next action is a really bad decision, and choose another location on the interactive interface as the location of next action. The interactive mode will be used for incorporating feedback into learned model.

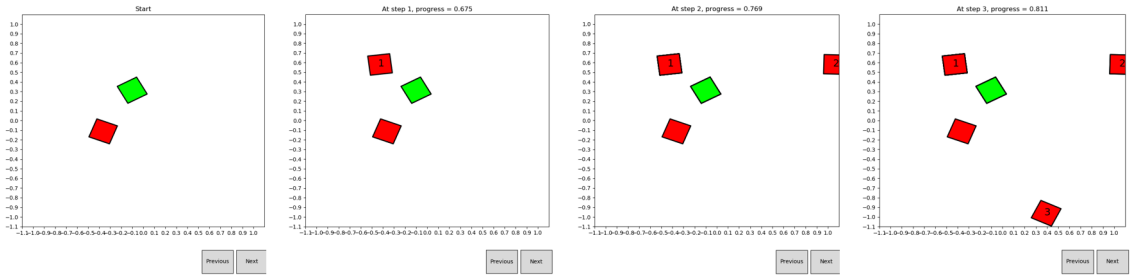


FIGURE 2.7: A demonstration in interactive mode using continuous learned progress function. At each step, the simulator looks for the best action that can increase the progress value. The progress values at the top improve from 0 to 0.675 to 0.769 to 0.811

2.4.2 Reinforcement learning algorithms

In the following sections, I will introduce two algorithm families used in Reinforcement learning framework. The first one is heuristic search algorithms; the second one is policy gradient algorithms. We use search algorithms as a quick way to evaluate the correctness of the progress learner (Section 2.3.2) in producing correct demonstrations. Algorithms of this type are, however, agnostic to past demonstrations. Therefore, for each novel setup they would have to search through the entire space to find an good solution.

True reinforcement learning algorithms usually have overhead, as we train them through generating a number of trial episodes. They have an advantage that after training time, you can just use the learned policy to generate higher valued episodes (corresponding to better demonstrations of actions in this discussion). The learned policy, however, depends on the progress learner, and therefore has to be retrained if the progress function changes.

2.4.2.1 Search algorithms

In this section, I will address two heuristic search algorithms. One algorithm is a *greedy* algorithm, in which from any current state, we just pick the action that leads to the next state that has highest possible reward. This algorithm can be used in *interactive mode* of the 2-D visualizer.

The second search algorithm is one-step *beam search* algorithm. The general idea of *beam search* algorithms is that at each step, we keep a list of "good" explorations. For each exploration, we randomly generate a number of sequences of actions, to create some candidate explorations. For example, at each time step, we have b explorations $L_l, l \in \overline{1..b}$. From each of them, with a two-step *beam search* algorithm, we generate 3 sequences of actions S_{li}, each sequence has two actions a_{lij}. We than apply each sequence of action on L_l, to generate exploration L_{li}, each with a different progress p_{li}. By sorting all resulted explorations according to their progress values, we select the next batch of explorations of size b. The one-step backup algorithm is shown in Algo. 3. This algorithm could not be used in *interactive mode*, because at any time step, we keep a list of candidate explorations, and the best exploration of the next step is usually not the continuation of the best exploration from the previous step.

input : state of the 2-D simulator (locations of blocks, which block would be moved), desired progress threshold (e.g. 0.80), search breadth n
output: A chain of actions
init : Randomize initial state in simulator space X_0, progress value = 0
for *Step* $k \leftarrow 1$ **to** max_step **do**
  Random n action u_{ki} on the searching space;
  for *action* u_{ki} **do**
    Do action u_{ki} ;
    Calculate progress p_{ki} and reward r_{ki};
    Back to previous state;
  end
  Select $u_{k\ best}$ according to best progress value $r_{k\ best}$;
  if $r_{k\ best} > 0$ **then**
    Do action $u_{k\ best}$;
  end
end

Algorithm 2: Greedy search for trajectory of action

input : Same as Algo. 2 + A beam search width b (e.g. $b = 20$)
output: Same as Algo. 2
init : A list of explorations L and their progress values P (stored in a map). At the beginning it has only one exploration $[(X_0)]$ which is the blocks are at their original positions, no movement has made yet (progress value = 0)

for step $k \leftarrow 1$ **to** max_step **do**
  $L_{candidate} = L.clone()$;
  for $l \leftarrow 1$ **to** b **do**
    Exploration $e = L_l = [X_{l0}, u_{l1}, X_{l1}..]$, progress $P[e]$;
    Random n action u_{lki} on the searching space;
    for *action* u_{lki} **do**
      Do action u_{lki}, lead to state X_{lki}, reward r_{lki} ;
      Create candidate exploration $e' = e + [u_{lki}, X_{lki}]$, progress $P[e'] = P[e] + r_{lki}$;
      $L_{candidate}.add(e')$;
      Back to previous state;
    end
  end
  Sort $L_{candidate}$ so that the highest progressed explorations are at the top;
  Set $L = L_{candidate}[: b]$;
end
Return the best exploration L_0

Algorithm 3: One-step beam search for trajectory of action

2.4.2.2 Policy gradient algorithms

In this work, I will experiment with both running RL on continuous space and discretized space and we can evaluate the effectiveness of both methods in solving this problem. The selected algorithms for both cases are policy gradient algorithms, including REINFORCE (Williams, 1992) and ACTOR-CRITIC. These algorithms could be used for both continuous and discretized RL problems.

While digging deep into several RL concepts is not the focus of this dissertation, it is sensible for me to include a very brief introduction of policy gradient algorithms, and the reason of selecting them in this research. Concerned readers can find more details in other seminal RL work, such as (Sutton and Barto, 2018, Chapter 13).

The first important point to note about policy gradient methods is that they typically have two learning functions (or *estimators*). One is called *policy estimator*, i.e. a function that takes in a state, and outputs a probability distribution over a set of actions that could be taken from this state. Another estimator is called *state estimator*, technically an optional baseline used to reduce learning variance, formalized as a function from a state to the expected reward that one can achieve when starting from that state. For each estimator, we have a set of parameters that we would like to learn.

Now the reason these methods are called *policy gradient* is that we will apply online gradient learning methods, such as Stochastic gradient descent, to learn the optimal policy estimator. Optimality is defined by the accumulated reward (or episode *value*) that one can get for every possible starting conditions.

input : Initial policy parameters θ, learning rate α^{θ};
Initial state parameters ω, learning rate α^{ω}
Learned progress function f, Termination condition q, No. of episodes M
output: Learned policy parameters θ_{final}, ω_{final}

Algorithm 4: Inputs and outputs of REINFORCE and ACTOR-CRITIC algorithms

In particular, REINFORCE algorithm depends on the intuition that if one does not know which policy leads to the best reward, one should try exploring the environment by following some random policy until termination, each exploration we will call an *episode*. After each episode, we observe the reward collected from the episode and improve upon the random strategy, *rewarding* actions that lead to better accumulated reward, and *punishing* ones that does not. For example, if we have seen a common state s in two previous *episode* e_1 and e_2, at time step t_1 and t_2, but the strategy chose two different actions a_1 and a_2. Starting from this common state s, at the end of each episode, we accumulated two different values $v_1 > v_2$. Intuitively, we should reward a_1 and punish a_2, and update our action policy at state s. In practice, we add an additional term called *baseline*, which estimates the expected value of state s, and instead of comparing between two values v_1 and v_2, you can compare them with the baseline, rewarding the action corresponding to value higher than the baseline and punishing one that is not.

Details of REINFORCE algorithm is presented in Algorithm 5.

for *Episode* $j \leftarrow 0$ **to** M **do**
  init: policy estimator $\pi_\theta \leftarrow \theta$, state estimator $\hat{v}_\omega \leftarrow \omega$, randomize initial state in simulator space X_0, progress r = 0
  for *Step* $k \leftarrow 1$ **to** ∞ **do**
    Draw a set of n actions from policy distribution $u_{ki} \leftarrow \pi_\theta(X_{k-1}), i = \overline{1..n}$;
    for *action* u_{ki} **do**
      Do candidate action u_{ki} in simulator, get candidate next state X_{ki}, record frame-to-frame sequential features, feed them into LSTM network to get progress output;
      Calculate immediate reward as $r = \delta_f = f(X_{ki}) - f(X_{k-1})$;
      Back to previous state;
    end
    Select action u_k that leads to highest reward r_{max};
    Do action u_k in simulator;
    Get next state X_k;
    if $q(X_k) = True$ **then**
      break ;
    end
  end
  After episode terminates, obtain $X_{0:H}$, $u_{1:H}$, $r_{1:H}$;
  for *step k from H to* 1 **do**
    Calculate R_k, state value of X_{k-1} as accumulated reward from step k to the end of episode;
    Estimate baseline by state estimator $baseline = \hat{v}_\omega(X_{k-1})$;
    Calculate $advantage = R_k - baseline$;
    Estimate gradient and update parameters for value estimator
    $g_\omega = \bigtriangledown_\omega \hat{v}_\omega(X_{k-1}) * advantage$;
    $\omega = \omega + \alpha^\omega * g_\omega$;
    Estimate gradient and update parameters for policy estimator
    $g_\theta = \bigtriangledown_\theta log\ \pi_\theta(u_k|X_{k-1}) * advantage$;
    $\theta = \theta + \alpha^\theta * g_\theta$;
  end
end

Algorithm 5: Hybrid REINFORCE + Heuristic search algorithm in simulator. Notice that if the number $n = 1$, the algorithm becomes true REINFORCE with baseline

ACTOR-CRITIC algorithm is different from REINFORCE in an important aspect. While in REINFORCE, we have to run the episode until termination to decide the episode

value of each state, and only update the model at the end of each episode (in what is called *Monte Carlo* approach), ACTOR-CRITIC models are updated much more eagerly, i.e. after each action step. Because we do not have real state value R_k when we have not terminated the episode, we estimate this value by a term called *temporal difference target* or td_target, basically the sum of the current reward and the *expected* accumulated reward from the next state. ACTOR-CRITIC algorithm is presented in Algorithm 6. Because these two algorithms are similar in formalization and in inputs/outputs, and only different in details of parameter update, I will show ACTOR-CRITIC in an abridged form.

for *Episode* $j \leftarrow 0$ **to** M **do**
  init : Same as REINFORCE
  for *Step* $k \leftarrow 1$ **to** ∞ **do**
    Select the best action u_k from a set of n actions drawn from policy distribution $u_{ki} \leftarrow \pi_\theta(X_{k-1}), i = \overline{1..n}$, the corresponding reward is r_k, next state X_k ;
    $td\_target = r_k + \hat{v}_\omega(X_k)$;
    Using td_target instead of the real state value R_k
    Estimate gradient and update parameters for value estimator
    Estimate gradient and update parameters for policy estimator
    if $q(X_k) = True$ **then**
      break ;
    end
  end
end

Algorithm 6: Hybrid ACTOR-CRITIC + Heuristic search algorithm in simulator. Abridged form, same inputs and outputs as REINFORCE, same gradient estimation and update steps as REINFORCE

Because REINFORCE[2] and ACTOR-CRITIC are similar in their setup, they are generally considered to be variances of the same algorithm in RL implementation.

Some common practices that we should note in training policy gradient algorithms are listed as follows:

[2]REINFORCE actually stands for *REward INcrement = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility*

- Optimizer: Because of the form of gradient update in policy estimator, only the simple Stochastic Gradient Descent optimizer can be used for policy update, other optimizers (such as Adam) would be not appropriate. For the value estimator, because the network model is very simple, with just one neuron, I also use SGD for simplicity.

- Termination condition q: Because in this problem, we do not have an inherent termination condition (cf. Gridworld problem (Tizhoosh, 2005), where we terminate when the agent hit a target, or fall out of the grid boundary), we have to craft a termination condition. Two termination conditions are considered: one is limitation of the number of steps, and the other being *early-stopping* when progress surpasses a certain a threshold.

- Updating order: In two algorithms, I show the classical order of predicting (get $\hat{v}_\omega(X_k)$) and updating ($\omega = \omega + \alpha^\omega * g_\omega$) value estimator in which the predicting step is taken before the updating step. However, it is observed that if the value estimator is updated *before* being used for prediction, the algorithm has less variance and generally converges toward better result (the reason is intuitive, if we know that the *updated* model is better, than use it for prediction would lead to better result). This detail is considered a preferable implementation variance, rather than an algorithmic difference.

- Use of learned algorithms: a learned policy estimator has a stochastic nature, because you choose an action based on the produced probability. However, when we use the learned policy for downstream applications, we should use a greedy method to pick the most probable action for each state.

- Use of hyperparameters: Similar to when we train sequential models, we usually make the learning rate decay over time, i.e. we keep a high learning rate at the beginning so that learners can pick up some learning pattern quickly, but lower the learning rate later to avoid learners jumping out of the good parameter region.

- Performance of reinforcement learning algorithms is shown by the episode value, in this case the progress value, averaged over a slicing window of certain size. For an effective RL method, this value need to increase over time. We can also analytically evaluate the learned policy by printing out the predicted action probabilities for each state.

2.5 Visualization module

The visualizer is modified from our lab's internally developed environment for generating animated scenes in real time using real-world semantics of objects and events (Krishnaswamy and Pustejovsky, 2016). The whole framework is a robust system that can work with a large set of actions and objects, and backed by a solid theoretical foundation (Pustejovsky and Krishnaswamy, 2016). In this dissertation work, I will only describe the particular scene (5.11) used to generate visualization data for testing my learning methodology:

1. In the original setup, the scene has an embodied agent, named Diana, playing the role of a communicative agent; she can move blocks around using her hands. In production of visual scenes, however, her movement is quite shaky, especially in the transition phase between two actions in a row. Therefore I removed her out of the scene.

2. The scene has a Loading button that can be used to load an existing demonstration. After loading a demonstration, the users can click play and record the visualized scene.

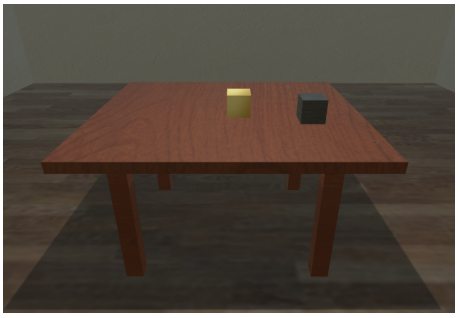


FIGURE 2.8: Visualizer is implemented as a Unity scene

Chapter 3

Event capture and annotation tool (ECAT)

3.1 Motivations

ECAT is an open-source interface tool for annotating events and their participants in video, capable of extracting the 3D positions and orientations of objects in video captured by Microsoft's Kinect® hardware. ECAT is created to address the lack of an annotation tool that can handle multimodal motion captures and allow multiple layer of semantic annotation on top of captures. ECAT can also be used as a general purpose toolkit for development of multimodal resource for machine learning tasks.

Particularly, ECAT addresses the deficiency of two other Kinect's capturing softwares. The first one is the official tool provided by Microsoft's Kinect SDK that writes the captured data into proprietary format. It is also not open source, and it is impossible to add semantic annotation extension on top of it. The second tool is the CwC apparatus API that can coordinate between two Kinect sensors, but the output depth stream data are also raw, and again, it is proprietary. In order to make ECAT independent of those tools, and to remedy their shortcomings, ECAT provides its own capturing, tracking and annotating functionalities. ECAT is originally designed to be interoperable with Communication with Computer (CwC) capturing and tracking apparatus API. However, following the progress of CwC project, the API was deprecated, and ECAT became

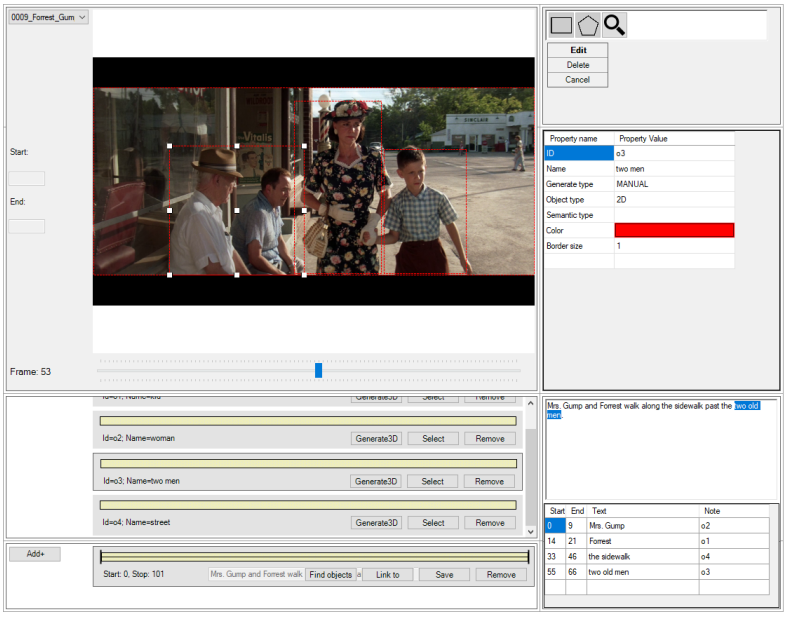


FIGURE 3.1: An example of ECAT-based annotation of the Movie dataset

a standalone and general-purpose application. The toolkit has been presented at the International Workshop on Interoperable Semantic Annotation ((Do et al., 2016)).

We also look for other possible applications of ECAT, besides its original purpose. ECAT has been used to capture and annotate data for fine-grained event classification of human-object interaction (Do and Pustejovsky, 2017a) and for action annotation in movie snippets (Do, 2016). In the next section, the details of these applications are presented.

3.2 Applications

In this section, I will list some applications of ECAT that we have explored, and some others that might be of interest for future consideration:

- To generate corpora for events and human activities captures: together with a RGB-D motion captures, ECAT supports tracking and marking of positions and orientations of objects and human body-rigs. These objects can be annotated as participants in the recorded motion event and these labeled data can then be used to build a corpus of *multimodal semantic simulations* of these events that can

model object-object, object-agent, and agent-agent interactions. A library of simulated motion events can serve as a novel resource of direct linkages from natural language to event visualization. For example, Do and Pustejovsky (2017a) have used ECAT to capture actions for event classification. This project has shown that ECAT-based capture and annotation are convenient for multimodal captured data.

- To create annotated resource for video understanding. ECAT has been used to collect dynamic event semantic annotation for movie dataset (Do, 2016). By annotating complex human activities and human-human as well as human-object interactions, we can extract adequate features to detect salient events in movie snippets. For example, we can use ECAT to annotate binary predicates (over humans and objects), such as HOLD, LOOK AT, POINT AT etc. These can be used to mark location of salient event from movie snippets (See Figure 3.1 for an example annotation).

- To create annotated resource for learning of object Gibsonian affordances (Pustejovsky et al., 2017). For example, ECAT can be used to capture video of an object from multiple viewpoints, and one can annotate the part of the object that correspond to its Gibsonian affordance region (i.e. the part of a tool that supports grasping for manipulation). As we have discussed in the Introduction, object affordances are important concept in action learning. While there is some annotated static image corpus for locations of grasping (Lenz et al., 2015), a similar corpus but with captured videos could be of interest for robotic community.

3.3 Graphical user interface

Figure 3.2 shows the ECAT GUI. Its components are listed below:

1. Project management panel. Each project can hold multiple captured sessions.

2. Video display. For displaying either the color video or grayscale depth field video, and locating objects of interest in the scene—e.g., the table outlined in green in Figure 3.2.

3. Object annotation controller. Yellow time scrub bars show when each tracked object appears in the video. Black ticks mark frames where an annotator has drawn

a bounding polygon around the object using the object toolbox (item 5). *Link to* button links the selected object to another using a specified spatial configuration. *Generate3D* button generates the selected object's tracking data using the depth field.

4. Event annotation controller. Time scrub bars here show the duration of a marked event. Users provide a text description for the event, or use *Link to* button to link the selected event to another captured event as a subevent. ECAT supports marking events that comprise multiple non-contiguous segments. Due to space constraints not all annotated subevents are visible in this screenshot.

5. Object toolbox. Annotators can manually mark an in-video object with a bounding rectangle or arbitrary polygon. Marked bounds can be moved across frames as the object moves.

6. Object property panel. Data about a selected object shows here, such as ID and name.

7. Event property panel. The selected event's properties, including its type and participants, show here, and the event can also be linked to a VoxML event type.

3.4 Functionality

1. ECAT provides a tracking mechanism based on OpenCV implementation of detection algorithm for ARUCO markers (Garrido-Jurado et al., 2014). Users can put different ARUCO markers on different sides of a block so that ECAT can calculate its pose/orientation change over time. In a nutshell, the algorithm looks for parallelograms (demarcated by strong contrast pattern of black and white border) by searching through a gray image. Then it looks for the checker pattern after transforming the parallelograms to squares. Searching over the whole image is carried out every 10 frames, whereas for remaining frames, the algorithm looks for space in the neighborhood of the previous frame.

2. ECAT provides mapping back and forth between 2-D representation and 3-D representation of objects. It can infer corner coordinates of cubes in 3-D from detected ARUCO markers. It can also infer the equation of the geometrical plane

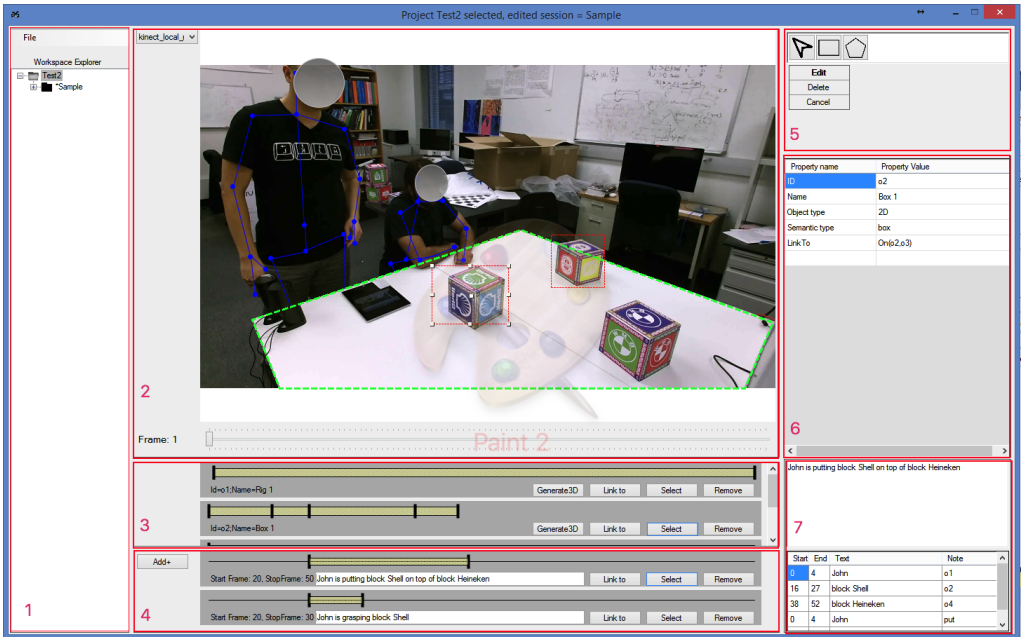


FIGURE 3.2: ECAT GUI. The left panel allows annotators to manage their captured and annotated sessions. Recognized human rigs are displayed as blue skeletons. Objects of interest are marked in color in the scene view. Here shows an example of collecting data for human activity corpus

of a flat surface (such as the supporting table). Inference is based on object's `semanticType`, that can be linked to VoxML (Pustejovsky and Krishnaswamy, 2016). for example, if it is a $marked\_cube$, ECAT can infer the cube's 3-D coordinates, whereas if it is a $flat\_surface$, it can infer the equation of the surface's geometrical plane.

3. Users can mark the presence of an object using the object toolbox. Marking the boundary of an object in 2-D can be used to infer its convex hull in 3-dimensional space by using depth-field data.

4. Annotators may also mark spatial relations between objects. For example, at 3.2 two blocks are on top of the table. Users can create a link between a block object and the table object, using the *Link_to* button on the object tracking panel, and specify the relation between the objects as "on," resulting in the creation of a predicate $ON(Block\_1, Table)$ that is interpretable in a modeling language such as VoxML.

5. ECAT allows annotators to mark subevent relations between events. Thus, as in the example, the overarching event may be annotated as *put*, but it contains the

subevents *grasp*, *hold*, *move*, and *ungrasp*, which may overlap partially with each other.

3.5 Future extensions

The following are a list of desirable functionalities that a next version of ECAT will support:

- Recording of audio inputs: Currently ECAT does not support recording and playback of audio inputs. We can imagine a scenario of using audio inputs to automatically annotate actions, i.e. action spans and action descriptions. Action spans can be demarcated by saying key words such as START and END, while action descriptions can be automatically recorded by a speech recognizer.
- Multiple camera support: in robotics, there are usually multiple cameras operating at the same time, capturing a scene from multiple viewpoints. The aim is three-fold: to extend the robot's field of view (FOV), to compensate for object occlusion, and to augment object tracking. This addition would generally require users to calibrate different cameras for their coordination before capturing.
- Integration with object detection (and tracking) library: deep learning object detection library, such as (Ren et al., 2015) could be integrated with ECAT to provide an easy-to-use toolkit for researchers that are not familiar with working principle of deep learning methods. This feature would also expedite the annotation process by automatically selecting annotation regions for objects.
- Support for generic cameras: one of the current deficiency of ECAT is that it only supports Microsoft's Kinect®V2 hardware. It is desirable to include a module for capturing from generic cameras (or other types of RGB-D sensors, as Microsoft Kinect®V2 project has been killed off for lack of interest from the gaming community).

Chapter 4

Action recognizer

To learn action representation, the most intuitive approach is to learn an action recognizer that can classify different action types. Apparently, this does not directly lead us to a representation that allows machines to perform actions, but we would come close to a good feature representation for the reenactment task.

This chapter describes a machine learning framework used to make distinction between different fine-grained action types. Particularly, it could be used to classify different action types that combine *manner*- and *path*- aspects of motions.

Technically, the framework uses the LSTM models we have addressed in Section 2.3.1 as the backbone to classify frame sequences, and Conditional Random Field (CRF) as a constraint layer for output labels. In this infrastructure, *manner*- and *path*- motion aspects could be jointly classified.

4.1 Motivations

1. To learn action models that allow AI agents to reenact actions, current coarse-grained human activities learner are not sufficient. For example, assuming that we have a learner that can distinguish generic coarse-grained activities such as *running*, *sitting*, *eating*, and *playing tennis*, it is unclear how to turn it into an actionable model. Therefore, we have to zoom into the details of each action. For example, action model of *playing tennis* would be a loop of *hit the ball to the other*

side. This unit action, in turn, requires a model of the tennis ball's trajectory, of robot's locomotion, of racket swing and of the opponent etc. We focus on much simpler type of actions, but the principles hold, we need to peruse the details of actions, i.e. we need to treat actions in a fine-grained level.

We can distinguish two different dimensions of fine-grained treatment, i.e. *temporal* granularity and *semantic* granularity.

***temporal* granularity**: Take the event *The performer rolls A past B* as an example. Zooming into different parts of the event, we see different things. At the beginning, the performer reaches his hand to A, A starts to roll closer to B till some point of time at which we can claim that A moves past B. A might continue rolling, with or without force from the performer's hand. If we slice different windows throughout the capture of this action, the action description would change from time to time. At the beginning, it is "The performer reaches A", then "The performer rolls A", then "The performer rolls A toward B", then "The performer rolls A past B" etc.

***semantic* granularity**: *manner*- and *path*- aspects of motions are reflected in the verb and adjunct slot in a frame-based representation of action description. For example, a description *The performer pushes A toward B* could be mapped to the slot representation (Subject, Object, Locative, Verb, Adjunct) as (The performer, A, B, Push, Toward). An ensemble learner could learn all the slots at the same time by combing multiple sequential learners. In particular, multiple LSTM learners, one for each slot, could be constrained to produce appropriate outputs.

2. Event classifier could be used as an auxiliary component for agents to distinguish their learned actions. We can try to make learning agents learn a novel action in isolation, but there are two reasons making distinction to other (already learned) actions might be helpful. The first reason is that we could make reference to the way humans learning a new concept. We exhibit a strong *mutual exclusivity bias*, which entails Principle of Contrast (Merriman et al., 1989), i.e. different concepts should have different names and vice versa, different words should be associated with different things. The practical reason is that a naive learner might erroneously learn both actions "move A past B" and "move A around B" as "A move closer to B, then move further away from B" (Figure 4.1). The learner might not be able to make distinction between these two actions without referring to a classifier.

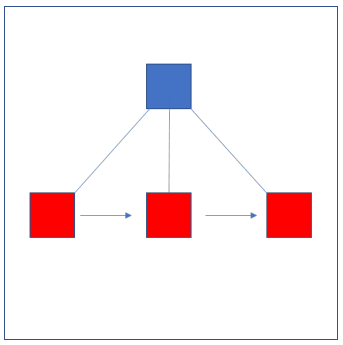


FIGURE 4.1: Red block moves past blue block

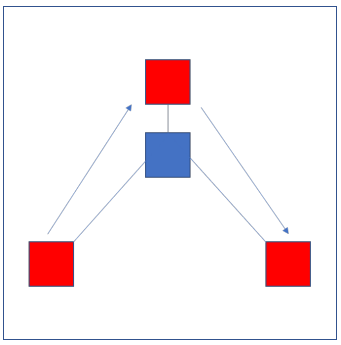


FIGURE 4.2: Red block moves around blue block

4.2 Models

4.2.1 Conditional Random Field (CRF)

Conditional Random Field (CRF) is a common undirected discriminative graphical model for machine learning of structural data. The most generic form of CRF is shown by Equation 4.1 or its log form (Equation 4.2):

$$p(y|z,w) = \frac{1}{Z(z,w)} \prod_c \psi_c(Y_c|z,w) \tag{4.1}$$

$$log(p(y|z,w)) = \sum_c \phi_c(Y_c|z,w) - log(Z(z,w)) \tag{4.2}$$

In these equations, z is input, y is output, w is model parameters (weights), Z is the **partition function** that sums over all joint configurations to normalize p into true probability. c is short for clique (a set of nodes that you want to model their co-dependency), and ψ_c (and its log version ϕ_c) is called a **clique potential**, which is a function over the values of all nodes in the clique. Notice that this function is not a true probability function, therefore, the partition function Z is always required to normalize the value on the right hand side into true probability. In other words, if we represent our graphical model as a *hypergraph*, each clique would correspond to a *hyper-edge*, and we constrain values of nodes on each *hyper-edge* by the clique's potential function. In its simpler form (called pairwise CRF), where we model on normal graph with normal edge, clique potentials are reduced to functions on nodes (node potential) and edges (edge potential).

In the following equations of pairwise CRF, I do not show the dependency on w for brevity:

$$log(p(y|z)) = \overbrace{\underbrace{\sum_{s \in \mathcal{V}} \phi_s(y_s|z)}_{\text{node potential}} + \underbrace{\sum_{(s,t) \in \mathcal{E}} \phi_{s,t}(y_s, y_t|z)}_{\text{edge potential}}}^{\text{Un-normalized log prob}} - log(Z(z)) \tag{4.3}$$

$$Z(z) = \sum_{y \in \mathcal{Y}} exp(\sum_{s \in \mathcal{V}} \phi_s(y_s|z) + \sum_{(s,t) \in \mathcal{E}} \phi_{s,t}(y_s, y_t|z)) \tag{4.4}$$

where $\mathcal{V}$ is the set of nodes, $\mathcal{E}$ is the set of edges, $\mathcal{Y}$ is all combinations of labels y.

4.2.2 LSTM-CRF

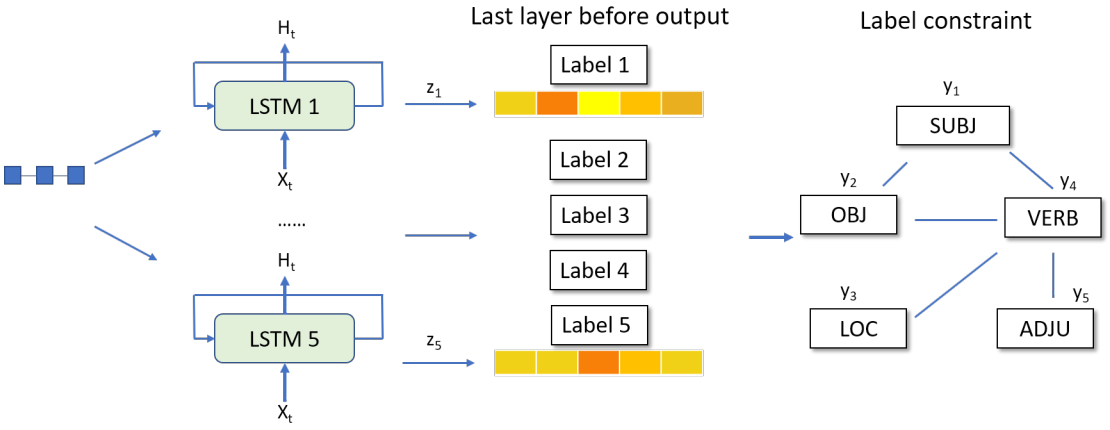


FIGURE 4.3: An array of LSTM models produce an array of output values, each corresponds to a sentence slot classifier. The values at the last layer of each LSTM is a probability distribution (in practices, their logarit values). These values would be fed into CRF as values of x in Equation 4.3

Now let say we want to use CRF to constrain outputs from LSTM models, we can use the log probabilities calculated from the last layer of each LSTM as input z for the CRF layer (remember, for each LSTM, we produce a log probability vector, see the first common practice at Section 2.3.1.2, so $z = (z_1, z_2, z_3, z_4, z_5)$ where z_i is the lob-probability vector for i-th LSTM). The value of y is a tuple of thematic slots, corresponding to our predictive label of (Subject, Object, Locative, Verb, Adjunct). For node potential, we can just use directly the log probability from LSTM+Dense output, selected for value of

y_s (Equation 4.5). For edge potential, we can totally ignore the input z, and just model the function on the labels of the edge's vertices. The edge potential for an edge (s, t) turns into a lookup into a 2-dimensional array that keeps a correlation value for each possible pairs of (y_s, y_t) (Equation 4.6).

$$\phi_s(y_s|z) = z_s[y_s] \tag{4.5}$$

$$\phi_{s,t}(y_s, y_t|z) = \phi_{s,t}(y_s, y_t) = L_{s,t}[y_s, y_t] \tag{4.6}$$

It is needless to say that fancier model could be used, such as to set $\phi_{s,t}(y_s, y_t|z) = L_{s,t}[y_s, y_t] * z_s[y_s] * z_t[y_t]$. The formalization of edge potential as in Equation 4.6 is a simpler *hardmax* approach.

4.2.3 CRF to Tree-CRF

The parameters of CRF are the lookup tables along the edges on the graph ($L_{s,t}$ in Equation 4.6), together with parameters of the individual LSTM models. Training the model with a set of training samples $(x_i, y_i), i \in \overline{1..S}$ would require us to calculate the value of z_i by passing x_i through LSTM models, then calculate $log(p(y_i|z_i))$. The complexity lies in calculation of $Z(z_i)$, as we have to sum over all possible combinations of y_i. Using the model to predict for a novel sample x_i would be equally expensive, as we still have to calculate the probabilities for all possible combinations of y_i, before finding the best combination by $y_{i\ best} = argmax_{y \in \mathcal{Y}} \quad log(p(y|z_i))$.

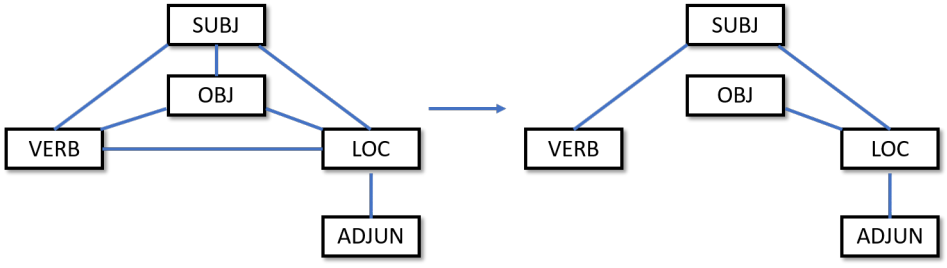


FIGURE 4.4: An example of reducing full CRF into tree CRF.

Updating and predicting a generic CRF model would be difficult, especially when implemented with a neural network architecture. I propose that we could modify the CRF layer into a tree-CRF structure (right side of Figure 4.4) to make the model learnable using dynamic programming. My algorithms for updating and predicting Tree-CRF

model, using a node-collapsing method, are presented in the following subsections. They are *exact inference* algorithms, that allow us to calculate the partition function Z (for updating) as well as to find the best y_i (for predicting) without having to iterate over all possible combinations of y. Implementation extending Tensorflow is provided at my repository.[1]

4.2.3.1 Update

Updating the model requires us to calculate the partition function Z. The graph collapsing algorithm on LSTMs is as following:

input : Input sequence x_i, output label tuple $y_i = (y_{i1}..y_{iQ})$
Q LSTM models and look up table values for all CRF edges
output: Updated model parameters, including parameters of Q LSTM models, and the look up table L
begin
  Calculate log-prob for each node on the graph (for example, $[z_{A_1}, z_{A_2}]$ on Figure 4.5), using a forward pass on Q LSTM models;
  Calculate un-normalized log prob for these specific values of y_i (See Equation 4.3);
  Set $collapsed\ graph = original\ graph$;
  while $collapsed\ graph$ *has edge* **do**
    Select an edge AB, where B is a leaf node. We index values of A by u, and B by v;
    Follow the update formula at Equation 4.7 to update stored values at A;

$$z_{A_u - collapse - update - B} = z_{A_u} + \sum_v log(exp(L_{A_u B_v} + z_{B_v})) \tag{4.7}$$

    Delete B and edge AB from $collapsed\ graph$
  end;
  Set Z = Sum all the values collected from the last node;
  Calculate log loss and run back-propagation on all networks (done automatically by Tensorflow mechanism);
end

Algorithm 7: Update step for LSTM-CRF

[1] https://github.com/tuandnvn/ecat_learning/blob/master/crf_tree.py

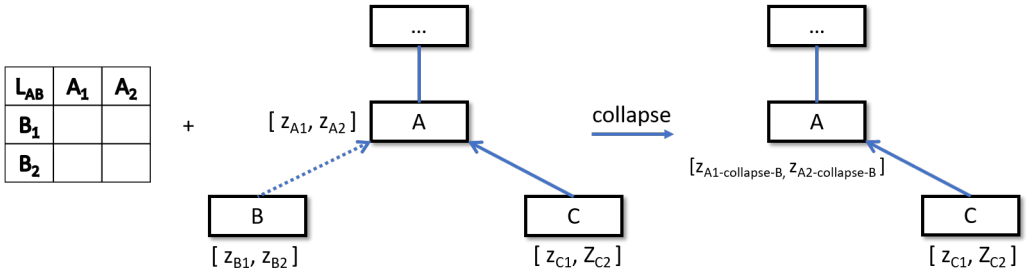


FIGURE 4.5: Collapsing an edge between nodes A and B into one node when calculating Z for updating. Note that B needs to be a leaf node

4.2.3.2 Predict

The algorithm for predicting a novel input is very similar to the algorithm for updating the model. The difference lies in another equation for updating the values stored in A (Equation 4.8). We just need to replace the sum operator in updating equation with a max operator.

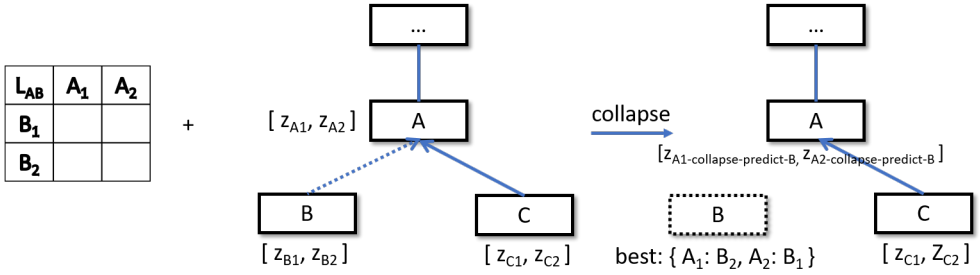


FIGURE 4.6: Collapsing an edge between nodes A and B into one node when predicting output for a novel input.

We also store a stack of edges that has been collapsed in their collapsing order. We also need to keep a mapping to help tracing the best combination. In Figure 4.6, imagine we keep a *best* map at the collapsed node B that maps from any value of A to the value of B that maximize our log probability for that value of A (Equation 4.9).

When the graph is collapsed to just one node, we can pick its label by selecting one that maximize its stored z values. Now it is straightforward that by popping from our collapsing edge stack, we can recover the best combination of labels. For example, in Figure 4.6, by picking out A_1 as the best label for node A, when we need to recover for the collapsed edge of AB, we will pick B_2 as the best label for B.

$$z_{A_u-collapse-predict-B} = z_{A_u} + \max_v log(exp(L_{A_uB_v} + z_{B_v})) \tag{4.8}$$

$$best_B[u] = \arg\max_v log(exp(L_{A_uB_v} + z_{B_v})) \tag{4.9}$$

Complexity: The time complexity of the algorithm is reduced from $\prod_{i \in \overline{1..Q}} |V_i|$ to $\sum_{(s,t)\in\mathcal{E}} |V_s| * |V_t|$ where $|V_i|$ is the number of classes for i-th label.

4.3 Experiment setup

To demonstrate the capability of this model to learn the *spatio-temporal* dynamics of object interactions in actions, I used a data collection of four action types: *push*, *pull*, *slide*, and *roll*, along with three different *spatial* adjuncts used for space configurations between objects, namely *toward* (when the trajectory of a moving object is straightly lined up with a destination static object and makes it closer to that target), *away from* (makes it further from that object) and *past* (moving object getting closer to static object then further again).

For each recorded session, I sliced the events into short segments of 20 frames. Two annotators were assigned to watch and annotate them (segments can be played back). To speed up annotation, only the event types related to the original captured types are shown for selection. For instance, if the event type of the captured session is "The performer pushes A toward B", other available event types are "The performer pushes A", "A slides toward B" or "None".

The set of constraints for output labels are the followings (described on the left side in Figure 4.4).

1. One object (Performer or the other objects) is not allowed to fill two different syntactic slots. Notice that a person can be either a Subject (in *The performer rolls A toward B*) or a Locative (*A rolls toward the performer*). Because an object can take Subject, Object or Locative slots, there are pairwise edges between these slots.

2. When there is no verb, all the other slots should be None. That is why we have the edge from Verb to Subject, Object and Locative.

3. Locative and Adjunct are dependent, because if Locative is None, Adjunct must also be None and vice versa.

Even though my set of constraints is quite simple, one can easily extend it to more generic case, such as to model selection constraints of predicate to its arguments. The reason I do not have that constraints is because the set of objects I have for tracking is simple and artificial (a cube and a cylinder).

4.4 Evaluation

Captured sessions are split for 5-fold cross validation, i.e., 24 sessions for training and 6 for testing on each fold. We use the LSTM-CRF model with raw input data as the baseline. A prediction is correct if all slots are correct. Performance is reported in the following tables:

	Model	Precision
Frame level	Quant-LSTM-CRF	48%
	Qual-LSTM-CRF	**60%**
Event level	Event-Qual-LSTM-CRF	23%

TABLE 4.1: Evaluation

Label	Precision
Subject	93%
Object	90%
Locative	80%
Verb	83%
Preposition	82%

TABLE 4.2: Label precision breakdown for Frame-Qual-LSTM-CRF

We can observe a significant improvement of classification using 2-dimensional frame-level qualitative features. Frame-level quantitative features did improve over our baseline, but the improvement is not as impressive. Moreover, summarizing frame-level features to create event-level features creates a lossy representation that is not be able to learned efficiently.

Given these results, it is worth considering possible explanations for our findings. Firstly, as pointed out in Yang and Webb (2009) and Jiang et al. (2017), qualitative representation is a method of discretization, which makes data sparser, therefore easier to learn. Especially when taking the difference between features of two adjacent frames, as a qualitative feature strongly distinguishes between 0 and 1, the effect of sequential change is more pronounced.

The best performance for **Qual-LSTM-CRF** is achieved by configuring two layers of LSTM with 400 nodes on the hidden layers, while for other models, the number of layers does not affect the performance significantly. Different from a feed-forward neural network such as Convolutional Neural Network (CNN), which can learn more abstract and useful features when it get deeper, LSTM needs some help from the representation of features to reap a benefit from going deeper.

We also learned that a simple summary representation for events is not effective. If we also takes into account features in intermediate frames, the representation comes back to a sequential one.

4.5 Conclusion

The overall architecture of this recognizer was presented at (Do and Pustejovsky, 2017a), but the content in the paper was shortened because of length limitation. I believe the approach as well as the method used in this recognizer would be of high interest for work in action recognition, and a detailed discussion of the motivation and algorithms used in this recognizer deserve a separate chapter in this dissertation.

What we found in this chapter is that the use of qualitative features would benefit classification results. We will see in the next chapter that the same principle would also apply when we learn models to reenact actions.

Chapter 5

Action reenactment by imitating human demonstration

To facilitate the process of teaching robots new actions and concept, we need an interface to teach robots the mapping between linguistic and visual representation of an action, in composition with participants (agents and objects). Whereas the ultimate target is an online mechanism that allows robots to learn new actions and concepts while continuously interacting with environment, to simplify the learning process, this dissertation work will still stick to the learning loop of:

$$Capture \rightarrow Process\ Data \rightarrow Machine\ Learning \rightarrow Evaluation$$

This learning process could be mapped to the following pipeline of:

1. **Capturing data using Event Capture and Annotation tool (ECAT)** — ECAT functionality and its usage is already summarized in Chapter 3. Data capture and annotation guidelines is already addressed at 2.1.

2. **Data processing and feature extraction modules** — This actually encompasses various components in ECAT and some other components implemented in Python. Components in ECAT include an algorithm to synchronize data from various capturing streams, such as depth-field and RGB, and one to generate 3-D data from 2-D tracked frame-by-frame object data, an algorithm. Components implemented in Python include an algorithm to recover missing data (when there are 2-D coordinates that could not be mapped to 3-D coordinates), to project data to simulator

space, to interpolate coordinates in frames that do not have corresponding tracked data etc. Feature extraction framework will be addressed at 2.2.

3. (a) **Supervised machine learning algorithms** — Algorithms that are used to predict progress of a learned action, detailed at 2.3

   (b) **Running simulation with 2-D simulators** — A 2-D simulator implemented in Python with various searching and reinforcement learning algorithms. Functionality and implementation of this simulator will be described in 2.4.1.

4. **Evaluation with visualizers** — Human evaluation based on watching the demonstration of action on 2-D and 3-D visualizers. 2-D visualizer is a component of 2-D simulator, whereas 3-D visualizer is a scene modified from Voxsim (Krishnaswamy and Pustejovsky, 2016). 2.5 will give a brief description of the 3-D visualizer, while **??** will elaborate on various experiments carried out on the visualizers. Finally, a *Feedback* step could be included to provide improvement to the learning loop (5.3.5).

5.1 Models

5.1.1 Searching algorithms

Similar to the RL algorithms, searching algorithms also have a continuous and a discretized versions.

For the continuous case, the action space is a Gaussian distribution that has $sigma = diagonal([2.0, 2.0, 0.5])$ and mean located at the center of a playground of size $(2, 2)$. Over the constrained playground, this distribution is close enough to a uniform distribution (The reason I do not use an exact uniform distribution is of implementation details: I want to have the policy produced in continuous case always follow a Gaussian distribution, so that its API could be unified with RL algorithms)

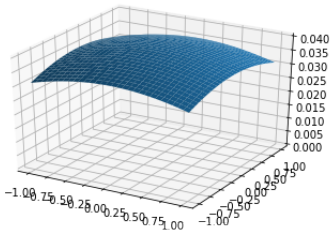


FIGURE 5.1: Randomize of action is based on a Gaussian distribution centered at $(0, 0)$

For the discretized case, it would be simple enough to derive the algorithm based on my previous discussion on discretization. One noteworthy point is that when we generate n random actions, we can just generate one action for each discretized space slot. That reduces the computational expense, while keeping the coverage over the searching space. That also allows keeping deeper layer of backups (so we can plan two or three actions ahead, before deciding which one produces the best progress).

5.1.2 Reinforcement learning algorithms

Using Williams Episodic REINFORCE algorithm (drafted in 2.4.2), I devise two different versions, corresponding to continuous feature space and discretized feature space as follows:

5.1.2.1 Continuous space

For the case when the action space is continuous, planning is carried out by selecting the action (coordinates in continuous space) at step k (u_k) based on the current state of the system ($X_{k-1} \in R^n$) (also continuous values). A stochastic planning step is parameterized by policy parameters θ : $u_k \sim \pi_\theta(u_k|X_{k-1})$

Problem formulation is as follows:

1. **State**: *transforms* (x- and y- coordinates and rotation) of two blocks in continuous space.
2. **Action**: *transform* of the target location of the moving object.
3. **Reward**: gain of *progress function* over one move.

4. **Policy**: My *Policy estimator* is made by an artificial neural network (ANN), used to produce values μ (mean of the action's transform) and σ^2 (variance of the action's transform). The ANN will be parameterized by a set of weights θ. For simplicity, the dimensions of μ and σ are the same as the degrees of freedom in the simplified simulator (2 dimensions for position and 1 dimension for rotation). Action is generated by a Gaussian distribution $\pi_\theta(u|X) = Gaussian(\mu, \sigma)$.

5. **Baseline function**: A function that estimate value of a state (expected accumulated reward from a state to the end of an episode). Here I use a simplest kind of ANN, i.e. a single neuron with *tanh* activation to produce a value from -1 to 1, to predict this value.

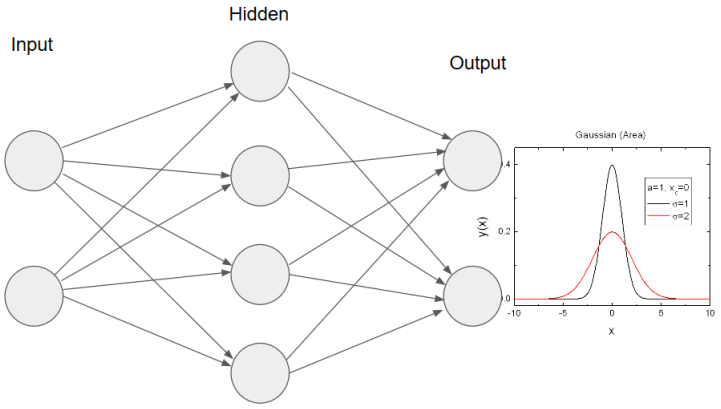


FIGURE 5.2: An ANN architecture producing Gaussian policy

5.1.2.2 Discretized space

I discretize the searching space using qualitative reasoning method. In specific, the searching space for the *transform* of the target location could be separated into space for (X, Y) coordinates and rotation r. The searching space for (X, Y) could be discretized as in Figure 5.3, the searching space for r could be discretized as in Figure 5.4. Notice the searching space for (X, Y) is so divided that the granularity of discretization is coarser when the moving object get further away from the static object.
Problem formulation is as follows:

1. **State**: Discretized state of the moving object with regard to the static object (6 values) + discretized previous action (or 0 if the object hasn't moved) (5 values)

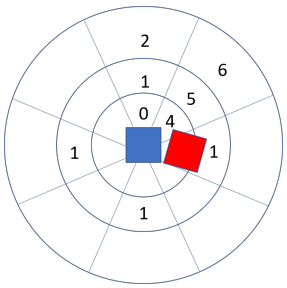


FIGURE 5.3: Discretizing 2-D searching space around the static object

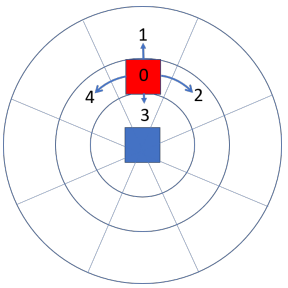


FIGURE 5.4: Discretizing action of the moving object

+ discretized progress of action (5 values). This combines into an *one-hot* vector of size 150.

2. **Action**: Discretized action of the moving object can take one of 5 values. Value 0 corresponds to an *early-stopping* of episode, while values from 1 to 4 correspond to going North, East, South and West (Figure 5.4).

3. **Reward**: gain of *progress function* over one move (same as for continuous case)

4. **Action policy**: An artificial neural network (ANN) will be used to produce probability distribution of discretized action for each state. My simplest implementation will use a fully connected layer for this ANN, with no activation and no bias, i.e. a linear matrix multiplication. A *softmax* layer is put at the end to create a distribution. A (discretized) action is generated from this distribution.

5. **Baseline function**: For baseline function, I use a linear function over the state vector. Because the input vector is *one-hot*, the corresponding linear coefficient would be the same as the state value.

The reason for the 6-way discretizing of relative position is that I want to utilize the four-way symmetry of the static block. While internally, I still make distinction among all the partitions in Figure 5.3, I would use the same action policy for same numbered slots, such as the slots numbered 1. I include the previous action into the state, because the previous action strongly affects the current action. For example, if the action is *Slide Around* and if the previous action is 4, the next action should also be 4. If the action is *Slide Closer* and if the previous action is 3, the next action should also be 3. The reason I include discretized value of the progress function is that I want to automatically learn a model that can stop when the progress is high enough. So for example,

when the discretized value of progress is 4 (corresponding to when $progress > 0.8$), the appropriate action would be 0. An alternative method would be to exclude discretized progress (which reduces state dimension to only 30), and to apply stopping when $progress > 0.8$. I would show results for both approaches in the following sections.

5.2 Experiment setup

In this experiment, I will use the learning framework drafted above for learning agent to perform the set of following *actions*:

1. An agent moves {object A} **closer to** {object B}
2. An agent moves {object A} **away from** {object B}
3. An agent moves {object A} **past** {object B}
4. An agent moves {object A} **next to** {object B}
5. An agent moves {object A} **around** {object B}

This set of actions, on linguistic description, differ only in their adjuncts. These adjuncts are prepositions describing different trajectories of motions. For this set, the learning problem is reduced to learning trajectories, or path, of motions.

It is noted that the action types in this set are not homogeneous. Firstly, we would discuss about temporal extent of each action type, whether they are open-ended or close-ended, and whether they have a hard boundary or soft boundary. Secondly, we would discuss the way humans would recognize these action types, whether we base on the trajectory of movement, or we just base on the start and ending points of movement. Followings are the differences:

1. **closer to** differ from **next to** that A only gets **closer to** B but does not touch B. **closer to** is a close-ended action, as A can only move closer to B until certain point. **closer to** has a soft ending, which means that the action is satisfied for a time interval.

2. **away from** is an open-ended action, it also has a soft ending.

3. **past** is an open-ended action and has a soft ending. It combines the effect of **closer to** and **away from**.

4. **next to** has a hard ending and is close-ended.

5. **around** has a soft ending and is open-ended.

From cognitive point of view, recognition of these action types, excepts possibly for **move next to**, requires consideration of the trajectory as well as the start and ending points of movements. For example, **closer to** is conceptually just involve change of distance between the start and the ending position of the moving object in relative to the static object, but a complex moving trajectory would lead to misinterpretation of the action. **Closer to**, therefore, strongly indicates a trajectory of the moving object toward the static object.

By putting these event types to be learned altogether, I want to examine the capability of a single learning framework that can learn multiple event types. The reason is rather obvious: we, as humans, can learn all of these actions without prior knowledge of different action types, or without further input from the teachers.

Data preparation

Currently for each action type, I have data recorded with two performers, and each person performs the action 20 times, for a total of 40 demonstrations for each action. Even though this is just a small amount of demonstrations, increasing this number is rather simple, as all data so far was captured in a single evening. Annotation took a little bit longer, about 10 hours. I also hope that the feedback loop could compensate for the small number of demonstrations, as increasing the number of demonstrations might still not be able to give good negative samples to the learning module.

5.3 Evaluation

In this section, I would carry out multiple evaluations. Firstly I would evaluate the performance of the learner progress, comparing between qualitative and quantitative

feature sets. Secondly, I would compare between the running time of searching algorithm and RL algorithms. We would like to know if the searching algorithms are fast enough. If they are fast, and perform equally well to RL algorithms (when evaluated by humans), it might be more reasonable to use searching algorithm to avoid overhead of running RL algorithms.

Thirdly, the performance of RL algorithms against our reward function could be measured by examining the average return for a span of consecutive runs. This figure would be increased overtime for a successful RL algorithm. Therefore we can use this method to avoid running expensive human-driven evaluation for any RL algorithm that could not converge to a good average return.

I will conduct one human evaluation based on watching the 2-D simulator, and one human evaluation based on watching simulations from the Unity 3-D visualizer. The human evaluation on 2-D simulator would be carried out on a smaller scale. The purpose of evaluation on 2-D simulator is two fold: firstly to reduce the number of control algorithms we would finally use for more expensive evaluation on the 3-D visualizer; secondly, that would help to answer if there is any disparity between the way humans judge the same action demonstration on 2-D simulator and 3-D visualizer.

Evaluation with 2-D visualizer will be performed by lab members or school students.

Evaluation with 3-D visualizer will be conducted using the Amazon Mechanical Turk platform (AMT). This platform has been proven to be a good source for cheap and reliable evaluation (Buhrmester et al., 2011), especially for cognitive ability tasks, such as testing adult ability to recognize words from uncertain auditory and visual inputs (Fourtassi and Frank, 2017), or human mental simulation in judging physical support (Gerstenberg et al., 2017).

We will answer the following questions:

1. Which algorithm works the best, and what is its performance? Addressed at 5.3.3
2. Can the learner makes distinction between learned action types? Addressed at 5.3.4
3. Can we use the feedback from human evaluation to improve the learned model? Addressed at 5.3.5

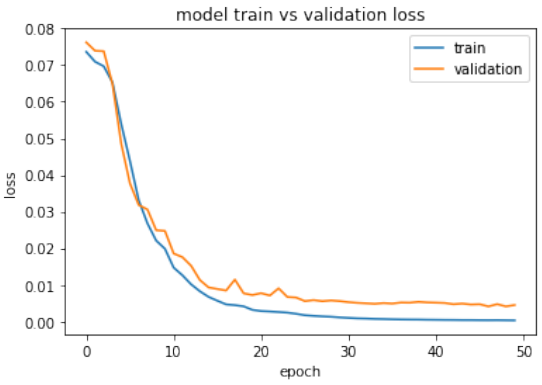


FIGURE 5.5: MSE with all data and **quantitative** features.

FIGURE 5.6: MSE with all data and **qualitative** features.

5.3.1 Evaluation of progress learner

Evaluation on validate data

The progress learner is a simple LSTM model that inputs the feature sequences from captures (or generated demonstrations) and outputs a single value from 0 to 1 corresponding to the progress of an action. Objective of this LSTM model is to minimize mean squared error (MSE) rate w.r.t annotated target progress values. Followings are the configuration that achieve the best performance:

Hyper-parameter	Definition	Value
keep_prob	Remaining percentage in dropout (See 2.3.1.2, 5)	0.6
hidden_size	memory state size	200
num_layers	Number of LSTM layers (See 2.3.1.2, 3)	2
max_epoch	Number of epoch runs, each epoch run through the whole training data	50
optimizer	Type of optimizer	Adam
learning_rate	Initial learning rate	0.005
lr_decay	Learning rate decay after each epoch	0.03

TABLE 5.1: Hyper-parameters of LSTM model for progress learner

In the following charts, I show the performance of the progress learner for different amount of training data and quantitative versus qualitative features:

We can observe that with all training data, performance with quantitative features is comparable with qualitative features. With less data (half the data equals 20 demonstrations, a quarter equals 10 demonstrations), the model with quantitative features is *underfitting* while model with qualitative features stills presents good fit quality, though the MSE line becomes quite bumpy.

FIGURE 5.7: MSE with half the data and **quantitative** features.

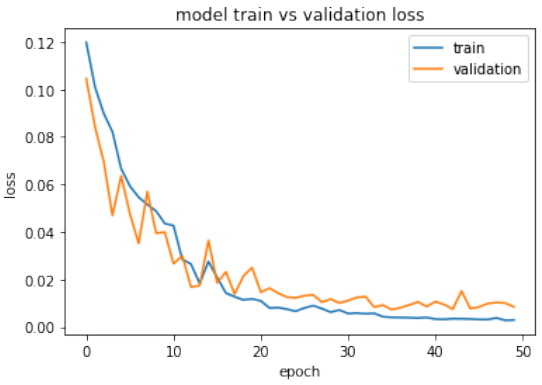


FIGURE 5.8: MSE with half the data and **qualitative** features.

FIGURE 5.9: MSE with 1/4 the data and **quantitative** features.

FIGURE 5.10: MSE with 1/4 the data and **qualitative** features.

5.3.2 Machine-based evaluation of actions

To complement human-driven evaluation, I also created a set of machine-based algorithms to evaluate the set of target actions. It is important to note that interpretation of these actions will be varied from person to person, and any effort to encode them in a programmatic way would be just an approximation. This evaluator could also be used as an oracle to provide feedback to our learning method.

The output of these evaluation algorithms (excepts for *Slide Around*) is binary, which means that we either accept a demonstration as correct or incorrect. Followings are the details implementation of these evaluation methods.

1. **Slide Close**: Slide closer means that for all the steps taken in the environment, the moving object needs to get closer and closer to the target object.

The algorithm is as follows:

(a) For each action step of the moving object, check the distance toward the static object, it should always getting smaller.

(b) The distance at the end of the action needs to be small enough (smaller than a threshold hyper-parameter).

(c) If one action step does not satisfy this condition, the whole action sequence is considered wrong (output 0).

2. **Slide Past**: My interpretation for Slide Past is that the distance between two blocks needs to be widest at the beginning and end states. If the total movement involves multiples action steps, that distance at a intermediate step ¡ max of distance at beginning and distance at the end. Moreover, we also want that the total path that the moving block has traversed is long enough, so from certain viewpoint, we can observe some occlusion happens during a period of time. Here I will just constrain that the largest edge of the triangle made from the beginning position, the end position and the static object position is the one between beginning and end positions. Also the angles next to this edge need to be smaller than a threshold angle (a hyper-parameter).

3. **Slide Away**: Slide away means that for all the steps taken in the environment, the moving object needs to get further and further to the target object.

   The algorithm is as follows:

   (a) For each action step of the moving object, check the distance toward the static object, we should always have each move getting the two objects further.

   (b) The ratio of the distance at the end and at the beginning needs to be higher than a threshold.

   (c) If one action step does not satisfy this condition, the whole action sequence is considered wrong.

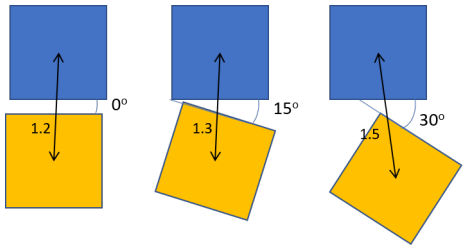


FIGURE 5.11: Examples of different values for hyperparameters: angle between two blocks, and distance between two blocks

4. **Slide Next To**: This action is a special action in this list of actions, because we only care about the final position of the moving block. A perfect position for "slide next to" is when the two blocks are aligned, and just exactly next to each other.

   The algorithm is as follows:

   (a) We would set two hyper-parameters for this method. One is the threshold for angle difference between two blocks (two squares). For example, difference of 0 to 15 degree seems quite aligned, but anything larger than that is quite bad. The distance between two square centers is normalized with block size. A threshold of 1.3 seems a reasonable value for this hyper-parameter. However, we could tune these hyper-parameter accordingly based on human evaluation.

   (b) If the values at the end of action does not satisfy the threshold condition, the evaluator outputs 0, otherwise 1.

5. **Slide Around**: My interpretation for "Slide Around" is quite simple: just calculate the covering angle of the moving object around the static object from the start point to the end point. The angle for each moving step will be positive if the angle is counter-clockwise, negative if the angle is clockwise. The final result is the absolute of the sum for all steps. Note that we do not take into account the change of the distance between two objects over time, because my method of generating action does not have smooth trajectory.

   I add two hyperparameters α_1 and α_2. To soften the definition, the output of the evaluator will be either 0, 1 or 0.5. If the calculated covering angle $\alpha < \alpha_1$, output 0, $\alpha_1 <= \alpha <= \alpha_2$, output 0.5, otherwise output 1.

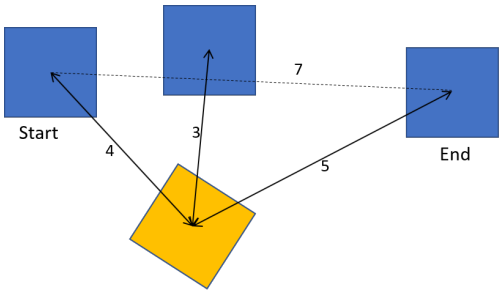


FIGURE 5.12: A demonstration of evaluation algorithm for Slide Past. We could check the distance between two objects at the beginning, at the end, and one intervening frame.

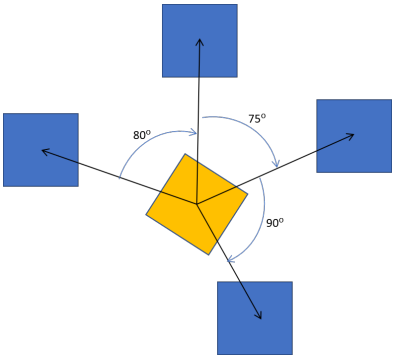


FIGURE 5.13: A demonstration of evaluation algorithm for Slide Around. We could check the total non-overlapping covering angle of the moving object around the static object. In figure, it is 245^o

5.3.2.1 Evaluation of automatic algorithms against human evaluation

Using the human judgment score, it is straightforward to evaluate the automatic evaluators. While the scores given by our machine methods are binary whereas the scores given by annotators range from 0 to 10, these scores are correlated, because we could imagine we can squash the annotators' score to the range from 0 to 1, then apply a binarizer function according to a threshold. Therefore, we can calculate a Pearson correlation coefficient for each hyper-parameter value, and select one that maximizes this correlation.

Followings are details selections of hyperparameters for each action type:

- **Slide Close**: Based on the values from Table 5.2, it immediately follows that we should choose the value of 3.5, i.e. the distance at the end of action between two blocks needs to be smaller than 3.5 times the block size.

Threshold	1.5	1.75	2.0	2.25	2.5	2.75	3.0	3.25	3.5	3.75	4.0	4.25	4.5
Correlation	-0.09	0.17	0.29	0.11	0.18	0.22	0.22	0.34	**0.40**	0.39	0.39	0.39	0.34

TABLE 5.2: Pearson Correlation coefficient (PCC) between automatic and human evaluations of Slide Close for different values of *threshold*

- **Slide Past** The best value combination is when $side\_ratio = 1.1$ and $0.4 * \pi <= angle\_threshold <= 0.5 * \pi$ or $72^o <= angle\_threshold <= 90^o$. Therefore

we can choose $side\_ratio = 1.1$, i.e. the straight path from the beginning to the end of moving object should be comparable to distance from the static object to the moving object, and $angle\_threshold = 90^o$, i.e. the angles adjacent to the moving objects are acute angles. Maximum value on the heat map is 0.72.

- **Slide Away** Based on the values from Table 5.3, it immediately follows that we should choose the value of the ratio threshold to be 2.3, i.e. the evaluators highly agree with the automatic method when we push one object 2.3 times further to the other object.

Ratio threshold	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
Correlation	0.04	0.01	0.01	0.15	0.24	0.27	0.11	0.03	0.18	**0.44**	0.41	0.31	0.28	0.28	0.26

TABLE 5.3: PCC between automatic and human evaluations of Slide Away for for different values of *ratio threshold*

- **Slide Next To** Based on the values from Figure 5.16, it immediately follows that we should choose the value for the angle between two blocks to be $0.05 * \pi = 9^o$, and the threshold for distance between two block centers to be $1.7 * block\_size$. Maximum value on the heat map is 0.66.

- **Slide Around** Based on the values from Figure 5.15, it immediately follows that we should choose the value for two thresholds to be 1.1 and 1.7. These values are quite close to our original guess of 1 (half a cycle) and 1.5 (three quarters of a cycle). Maximum value on the heat map is 0.8.

5.3.2.2 Mini conclusion

In this section, we have used scores given by human experts to create 5 automatic scorers, which we have tuned parameters to achieve highest possible agreement with human evaluation. In following experiments, we will use these automatic scorers because it will save some manual evaluation labor cost. In particular for when we want to improve the learned progress function with feedback (Section 5.3.5), we need quick oracle functions to direct our improvement.

Again I have to emphasize that there are probably other ways to parameterize the automatic evaluators that might equally fit with human judgment intuition. For example, one can argue that for good demonstration of **Slide Past**, we should parameterize the shortest distance between two objects (when one past another). However, increasing the

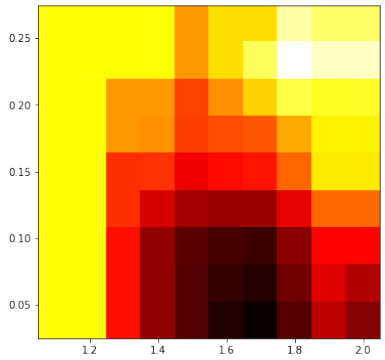


FIGURE 5.14: Heatmap showing the PCC values for different combination of $angle\_diff$ and $threshold$ for Slide Next To. The best value combination is when $angle\_diff = 0.05$ and $threshold = 1.7$.

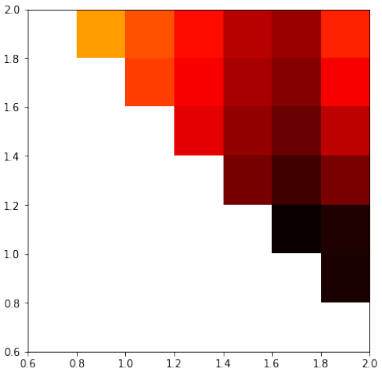


FIGURE 5.15: Heatmap showing the PCC values for different combination of $alpha_1$ and $alpha_2$ for Slide Around. The best value combination is when $alpha_1 = 1.1$ and $alpha_1 = 1.7$

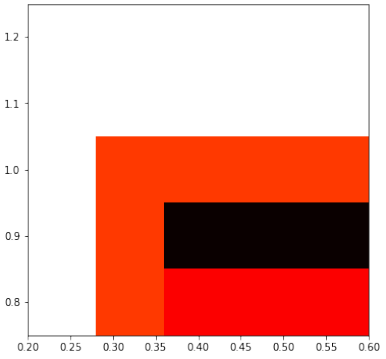


FIGURE 5.16: Heatmap showing the PCC values for different combination of $side\_ratio$ and $angle\_threshold$ for Slide Past.

number of parameters makes encoding the model more difficult and our target is just to create a simple oracle model that is agreeable to the human judgment scores.

5.3.3 Which algorithm works the best, and what is its performance?

For each evaluation round, I generate a random configuration of 2 blocks on the surface of the table. For each action type, the initializing algorithm will generate a text description of the action and generate a simulation corresponding to each algorithm. The

annotator will be asked to answer the question: *"On the scale of 0 to 10, how do you grade the video shown with regard to the description?"*.

In this experiment, by collecting grades for each action type, we get average performance of an algorithm in generating simulations. Hereby, we can evaluate if one algorithm would work for all action types in our collection, or there are disparity between action types that leads to advantages or disadvantages of a certain algorithm.

As the ultimate purpose is to pick out a good algorithm that is indifferent to action types to be learned, we would average over the performance of an algorithm over all action types. As for later experiments, they would get increasingly expensive if we keep multiple algorithms, so I will only keep the best algorithm for other downstream experiments.

5.3.3.1 Results

Searching algorithm

To measure performance of searching algorithms, we will generate 30 starting configurations. For each configuration, we will run searching algorithms (4 variances, *continuous* vs *discrete*, and *greedy* vs *one-step backup*)) for 5 different actions. We will evaluate with a few different measurements. The first one is average progress values of all demonstrations. The second one is the average episode length (number of actions before an episode terminates). The third statistics is average score calculated with the automatic oracles. The fourth figure is average time spent to plan the episode.

In the following experiments, hyperparameters are set as follows: maximum number of steps in one episode $max_step = 6$, number of explorations kept at each step in one-step backup $l = 36$, number of random actions tried at each step $n = 9$, threshold for early-stopping of an episode $progress\_threshold = 0.85$ (See Algorithm 3).

Results:

We can also see that on average, one-step back up algorithms will lead to higher progress value, and among two variances, the continuous version always performs better than the discrete one (Table 5.5). The search space of a one-step backup algorithm inherently subsume the search space of a greedy algorithm at the same depth depth, therefore we usually can find a demonstration that got higher target value with one-step backup

Action type	Search algorithm			
	Greedy		One-step back up	
	Continuous	Discrete	Continuous	Discrete
Slide Closer	1.77	1.27	1.57	2.03
Slide Away	1.60	1.27	1.03	1.8
Slide Next To	1.53	1.1	1.6	1.97
Slide Past	1.67	1.8	1.23	1.6
Slide Around	2.23	1.77	2.33	2.33

TABLE 5.4: Averaged number of steps for different action types and search algorithms

(There is an exception, which is when the greedy algorithm go deeper than the one-step backup).

Action type	Search algorithm			
	Greedy		One-step back up	
	Continuous	Discrete	Continuous	Discrete
Slide Closer	0.52	0.4	**0.81**	**0.81**
Slide Away	0.88	0.80	**0.92**	0.88
Slide Next To	0.67	0.56	**0.85**	0.79
Slide Past	0.87	0.85	**0.91**	0.9
Slide Around	0.82	0.75	**0.88**	0.84

TABLE 5.5: Averaged progress for different action types and search algorithms

There is, however, no common pattern could be seen in the performance of different algorithms when automatic evaluation methods are applied. For *Slide Closer*, the greedy version performs much better than the back-up version when the searching space is discrete (Table 5.6). The reason is that when we run back-up algorithm, the learner find out that it could increase the progress value, by first moving A away from B, than moving A closer to B again (comparing averaged progress value of greedy discrete 0.4 vs backup discrete 0.81). This strategy, however, is considered wrong by the automatic evaluation. For *Slide Away*, the result matches our expectation that searching on continuous space gives better result, because in discretized version, all positions of the moving block further than a threshold are considered the same. For *Slide Next To*, it is an advantage of the discrete algorithms that one of the discretized position are also considered an adjacent position. For *Slide Around*,

Action type	Search algorithm			
	Greedy		One-step back up	
	Continuous	Discrete	Continuous	Discrete
Slide Closer	0.67	**0.8**	0.67	0.13
Slide Away	**0.43**	0.27	0.40	0.3
Slide Next To	0.4	**1.0**	0.3	**1.0**
Slide Past	0.7	0.63	**0.77**	0.6
Slide Around	0.19	0.15	0.15	**0.20**

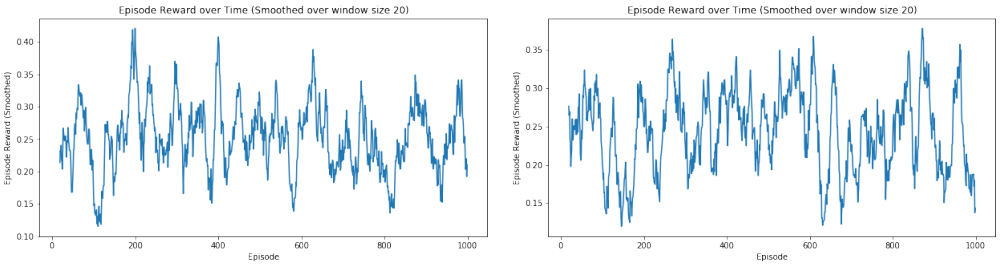
TABLE 5.6: Averaged score for different action types and search algorithms

RL on continuous space

Action type	Search algorithm			
	Greedy		One-step back up	
	Continuous	Discrete	Continuous	Discrete
Slide Closer	11.25	8.39	97.71	47.06
Slide Away	9.92	7.62	53.67	38.20
Slide Next To	11.62	8.01	95.27	53.14
Slide Past	8.26	7.74	46.41	16.87
Slide Around	8.97	6.57	59.39	29.66

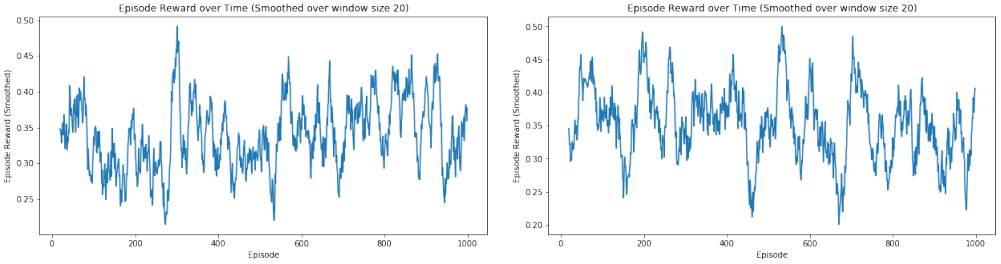
TABLE 5.7: Averaged time for different action types and search algorithms. Notice that time could not be compared between different actions

In this section I will show RL running results when we plan in continuous space, the algorithm is presented at 5.1.2. The hyperparameters in these experiments are set at $policy\_learning\_rate = 0.1$, $policy\_decay = 0.92$, $policy\_decay\_every = 100$, $value\_learning\_rate = 0.1$, $value\_decay = 0.92$, $value\_decay\_every = 100$



REINFORCE ACTOR-CRITIC

FIGURE 5.17: Average rewards of REINFORCE vs ACTOR-CRITIC with fix σ on continuous space for **Slide Around** after 2000 running episodes.



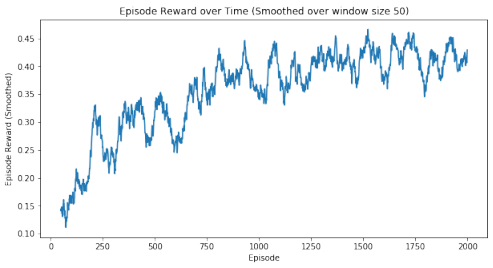
REINFORCE ACTOR-CRITIC

FIGURE 5.18: Same configuration as in Figure 5.17 but with $3 - action$ hybrid setup.

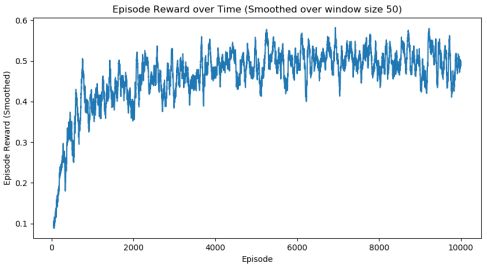
We can see in Figure 5.17 that both algorithms do not show any improving pattern. It is expected that the searching space is large and continuous, and the number of episodes might not be enough, but we will soon see that discretizing the searching space allows the RL algorithms to pick up learning pattern after a few hundreds episodes.

RL on discrete space

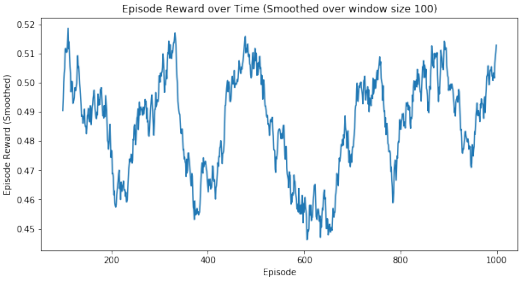
In this section I will show RL running results for one action **Slide Around** when we plan in discrete space, the algorithm is presented at 5.1.2. In the following runs of the experiments, the main hyperparameters are $policy\_learning\_rate = 1$, $policy\_decay = 0.98$, $policy\_decay\_every = 100$, $value\_learning\_rate = 0.1$, $value\_decay = 0.99$, $value\_decay\_every = 100$.



(A) True ACTOR-CRITIC with 2000 episodes



(B) True ACTOR-CRITIC with 10000 episodes



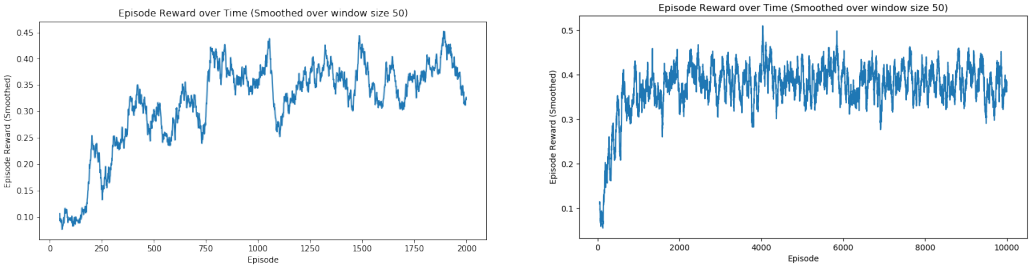
(C) Hybrid ACTOR-CRITIC with 2000 episodes

FIGURE 5.19: Average rewards of ACTOR-CRITIC on discrete space for **Slide Around** after 2000 and 10000 running episodes.

In Figure 5.19 (A, B), I report the accumulated reward (or final progress) of running 2000 and 10000 episodes of ACTOR-CRITIC algorithm. We can see that the algorithm seems to quickly improve from random search in the first 1500 episodes. Performance of 10000-episode experiment, after running for over 10 hours, peaked at around 0.5 and than plateaued.

In Figure 5.19 (C), we can see the result for a hybrid run between heuristic searching and ACTOR-CRITIC, i.e. when we generate one step of an episode, instead of picking a random action based on probabilities coming from action policy, we pick a number q of actions, and pick one that leads to highest reward. Based on a run of 1000 episode, we can observe that the average reward is improved, but the model does not seem to learn

anything. The local heuristic *force* is too strong for the algorithm to learn meaningful pattern.



(A) REINFORCE with 2000 episodes

(B) REINFORCE with 10000 episodes

FIGURE 5.20: Average rewards of REINFORCE on discrete space for **Slide Around** after 2000 and 10000 running episodes.

Results from REINFORCE algorithm is depicted in Figure 5.20. We can see REINFORCE performed a little bit worse than ACTOR-CRITIC, as the average episode reward peaked at around 0.4, and also reward variation is much more than when we ran with ACTOR-CRITIC.

A side note, not directly related to this learning setup, is that ACTOR-CRITIC algorithm is generally considered to be better than REINFORCE. ACTOR-CRITIC applies immediate update to the state value estimator and the policy learner, and therefore ensures faster convergence of the policy. This is confirmed in this experiment, as the run with REINFORCE has much larger variance and worse performance.

One possible shortcoming of this problem formulation is that states corresponding to higher progress values would have much less occurrences, and therefore would be much less visited while we generating episodes. The model, therefore, has not learned much of the action policy when the progress is high. Investigating in the action policy gives us some insights. Followings are action probabilities as given by ACTOR-CRITIC learned action policy for a few different states (recorded at 2000 episodes):

Based on observation from Table 5.8, we can observe that the algorithm has learned a reasonable policy at the beginning. When $progress = 0$, which means we are at some beginning steps, our model strongly prefers actions that move one object so that two objects still keep the same distance. In fact, out of 30 states that have $progress = 0$, 18 states produce 2 as the best option, 5 produces 4 as the best action. Action 2 is preferred over action 4 seems to be strongly preferred over action 4, even though there is no data

State			Action probabilities	Best action
Position	Previous action	Progress		
0	0	0	[0.007, 0.005, 0.974, 0.005, 0.008]	2
0	2	0	[0.017, 0.004, 0.967, 0.01, 0.002]	2
0	3	0	[0.212, 0.214, 0.194, 0.112, 0.268]	4
0	4	0	[0.173, 0.212, 0.179, 0.111, 0.324]	4
2	0	4	[0.165, 0.192, 0.187, 0.228, 0.227]	3
2	1	4	[0.210, 0.199, 0.247, 0.220, 0.122]	2
2	2	4	[0.192, 0.202, 0.224, 0.257, 0.125]	3
2	4	4	[0.193, 0.195, 0.210, 0.186, 0.216]	4

TABLE 5.8: Action policies, demonstrated by action probabilities given at different planning states. Recorded using ACTOR-CRITIC after 2000 episodes

State		Action probabilities	Best action
Position	Previous action		
0	0	0.017,0.008,0.956,0.009,0.011	2
0	2	0.051,0.004,0.933,0.007,0.005	2
1	0	0.036,0.003,0.953,0.004,0.004	2
1	2	0.070,0.041,0.882,0.004,0.003	2
2	0	0.023,0.010,0.940,0.010,0.017	2
2	2	0.021,0.007,0.968,0.003,0.001	2
3	0	0.025,0.008,0.939,0.011,0.016	2
3	2	0.034,0.004,0.954,0.005,0.003	2
4	0	0.005,0.002,0.987,0.003,0.003	2
4	2	0.012,0.973,0.013,0.001,0.000	1
5	0	0.008,0.006,0.973,0.005,0.008	2
5	2	0.973,0.011,0.012,0.002,0.002	0

TABLE 5.9: Action policies, demonstrated by action probabilities given at different planning states (state does not have progress component). Recorded using ACTOR-CRITIC after 2000 episodes. Only *legal* states that have action = 0 or action = 2 are included in this table. The full table could be seen in my repository.

bias (the number of clockwise captured *Slide Around* demonstrations is the same as counter-clockwise ones). We can also see that for the first two rows, the activation for action 2 is strong, because these states occur many times in generated episodes. We, however, do not see much difference in activation when $progress = 4$, implying that the model failed to generate enough number of demonstrations for effective learning at the end of action.

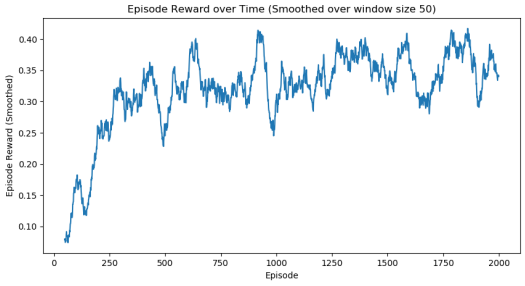


FIGURE 5.21: Average rewards of ACTOR-CRITIC on discrete space for **Slide Around** after 2000 episodes. State does not have quantized progress

Results from running ACTOR-CRITIC algorithm with state not incorporating quantized progress is shown in Figure 5.21. For this experiment, I only 2000 episodes, as the state space is 5 times smaller than when we has quantized progress as component. Average run results seems to be peak at around 0.40, which is worse than performance of the same algorithm when incorporating quantized progress. However, interestingly, as seen from Table 5.9, the model has learned a very simple rule for approximating **Slide Around**. Starting from positions from 0, 1, 3 and 4, the rule is simply that you keep choosing action 2 and move the object clockwise while keeping the distance to the static object, stop when progress function does not increase anymore. That is exactly how we could define **Slide Around**!

An exception is when we are at quantized position 5, and previous action is 2, the learned policy mysteriously decides that it should stop the episode by choosing action 0. A likely reason is that in the training data, there is actually no sample that has two objects too far from each other (this is a undesirable data bias stemmed from the limit of experiment space whereas we could not move the objects too far from each other without moving them out of field of view). Demonstration of this erroneous data bias could be seen at the first step of Figure 5.22 and Figure 5.23. The angle made by first movement of the red block relative to the green block are about the same in two figures, but in the first figure, the distance is smaller, and therefore the demonstration is closer to the training samples, the progress is much higher. This is also a kind of error we want to fix when incorporating users' feedback.

Results from running REINFORCE with similar setup give exact the same rule. Results are put in my Github repository, together with reward chart [1].

Human evaluation

Results of the system with heuristic search show that the progress learner can help to generate correct demonstrations (Fig. 5.22), but sometimes produces deviations (Fig. 5.23), probably because of the lack of negative training samples. This would be improved by incorporating feedback from evaluators as we will see in 5.3.5.

[1] miscellanous/run_from_cmd/run_re_no_progress_2000_3.txt

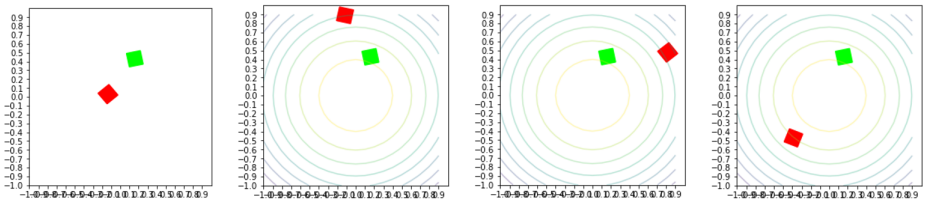


FIGURE 5.22: A good demonstration of "Move red block around green block."

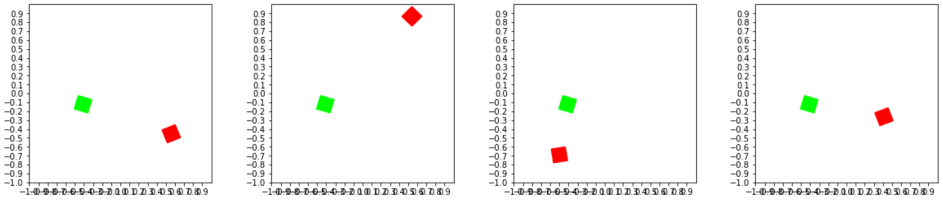


FIGURE 5.23: A bad demonstration of "Move red block around green block." The value beneath each frame is value predicted by the progress learner.

I also provide a quantitative breakdown of a small-scale human evaluation in Table 5.10 and 5.11. Two annotators (college students) are asked to give scores from 0 to 10 (higher scores are considered better) and are also asked to give comments on any video they graded between 3 and 7. **Evaluator Disparity** is the average of the absolute values of the differences between scores given by two annotators over the demonstrations of a particular action.

Action Type	Average Score	Evaluator Disparity
Slide Closer	5.4	1.57
Slide Away	6.48	2.37
Slide Next To	5.55	1.7
Slide Past	6.38	1.9
Slide Around	2.75	1.03

TABLE 5.10: Human evaluation for brute-force search on continuous space

Action Type	Average Score	Evaluator Disparity
Beam-search continuous	2.75	1.03
Beam-search discrete	5.95	0.7
RL with ACTOR-CRITIC	6.7	1.02

TABLE 5.11: Human evaluation of different algorithms on 30 demonstrations of **Slide Around**

Evaluator comments provide some insight into bad demonstrations. Following are typical comments for different actions types:

Action type	Text comments
Slide Next To	Need to be even closer
	Movement is slow
	It gets closer to the target, but than bounce back
Slide Closer	
	It gets closer to the target, but not enough
	Starting point is too close to the target
Slide Away	Need to be even closer
Slide Past	A touches B while passing B
	A slides past B than slides back
Slide Around	Lacks one or two more steps
	The last line goes to far from target
	The last line goes back to the opposite direction

TABLE 5.12: Samples of some comments given by annotators.

5.3.3.2 Mini conclusion

In this section, we have pitted different algorithms against each other, to find out which algorithm produces the best results. For all actions in our action set, excepts for "Slide Around", beam search on continuous space produce decent human-evaluation result. For "Slide Around", beam search on continuous space do not produce satisfactory results, but beam search on discrete space produces a more agreeable performance score.

We also ran policy gradient learning methods (ACTOR-CRITIC and REINFORCE) to improve upon the model of "Slide Around". On continuous space problem formalization, these algorithms failed to converge, but on discrete space, both algorithms successfully converged to interpretable policy. Applying the learned policy with greedy action selection produces the best human-evaluation score for this action.

5.3.4 Can the learner makes distinction between learned action types?

This experiment will examine the ability of the learner to distinguish these action types. The annotator will click on a *Randomize action* button, and the system will pick one of the available action types for Diana to perform. After the demonstration, a multiple choice panels will be displayed, and the annotator will then be asked to answer the question: *"Which of these sentences **best** describes the video shown?"*, with the aforementioned 5 descriptions, plus " None of these sentences describe the video" :

The task requires annotators to predict which sentence was used to generate the visualization in question. While the first experiment gives us a rough estimation of how well the learning algorithm is, only by juxtaposing different action types for annotators to select, we can measure if learning actions in isolated manner is enough. Some actions in this set are close enough for confusion, even when demonstrations are made by a human expert.

Because of time limitation, this is the only experiment that I carried out with the 3-D visualizer. With 3-D visualizer, demonstrations would be more realistic and closer to the way we perceive actions, and therefore, we could address some issues that does not occur when evaluation is taken in 2-D simulator. The first issue is the dependency of spatial perception on Point of view (POV). (Krishnaswamy and Pustejovsky, 2017) has shown that this dependency is not trivial, and changing of POV might lead to different human evaluation. The POV of the 2-D visualizer correspond a bird's-eye view, while the POV in 3-D visualizer would be similar to the captured environment, whereas the performers are moving and observing blocks on a table. The second issue is derived from the inherent property of three dimension objects, i.e. they can occlude each other. When that happens, our perception of spatial relationship is distorted, and we resort to prediction, rather than perception, for action recognition.

Results:

On the second experiment (Exp. 5.3.4), I expect high confusion between "moving next to" and "moving closer to", as the first action subsumes the second action. "Moving around" might be hard to recognize as well, as we generally think of the trajectory of "moving around" to be rather smooth.

5.3.5 Can we use the feedback from human evaluation to improve the learned model?

As mentioned in Section 2, feedback from human evaluation could be used as training data. Generated demonstrations from the first experiment (5.3.3) are complementary to real, captured data, because in learning by demonstration, we do not have any rigorous way to include negative samples. Imagine in real life, you teach your kids how to swing a baseball bat, they observe how you do it, and give a try. Unfortunately they did it so badly, because they have swap their bottom hand and their top hand. Now you give

an instruction to correct that, and they start to hit the ball more often. However, if you are required to give demonstration of *how to swing the bat really badly* right from the beginning, it might not come straightly to your mind that swapping hands is one bad way[2].

Reflecting on this thought experiment, we can see that learning of action is usually not one-shot learning. It is more of a process, in which we incorporating feedback from teachers to improve gradually. If that applies to humans, it should apply to machines learning from humans as well. In this section, we will investigate the possibility of using interactive methods to improve learned models.

I will focus on two different kinds of feedback, which I term *cold feedback* and *hot feedback*. The distinction between cold and hot here refers to the feedback time. *Cold feedback* is given at the end of a demonstration episode to evaluate how well the demonstration is, while *hot feedback* could be given in the middle of the demonstration, and giving the clue of not only *whether* the demonstration is bad, but also *where* it is bad, and *how* it should be corrected.

Projecting into the context of the CwC project, cold feedback corresponds to positive acknowledge (pos-ack) and negative acknowledge (neg-ack) (Krishnaswamy et al., 2017). In that communicating framework, pos-ack could be translated to a head nod or a thumbs up pose with either or both hands, or an utterance "right" or "yes", to signal agreement with a choice by the machine. Neg-ack could be mapped to a head shake, thumbs down with either or both hands, or palm-forward stop sign, or "no" utterance, to signal disagreement. In learning framework, this could be used as a binary evaluation signal for machines to update their learned models. [3].

Hot feedback corresponds to online correction in the middle of demonstration. In the context of CwC project, it might be mapped to a pointing (deixis) including the direction of the hand and/or arm motion. In CwC peer-to-peer setup, in which a person use multimodal communicative methods to direct an avatar, deixis is used to move attention of the avatar to a target object, or a target location. In learning from demonstration framework, we can use this kind of clue to correct an erroneous move of the avatar.

[2]This is actually related to my personal experience with baseball. The first time I came to a batting cage, I did not figure out the way to hold the bat, and swing very badly for 10 minutes without any of my friends noticing what is wrong, until some kids standing outside just yelled at me "You just hold it wrong, right hand on top"

[3]The terminology here is just to reflect how much of interactiveness the experiment is, and might not be similar to the terminology in literature in Dialog and Discourse

Followings I will discuss how we will incorporate *cold feedback* and *hot feedback* into this machine learning framework:

- *Cold feedback*: a binary value indicating whether a demonstration is good or bad. Using the grades that we get from annotators (which ranges from 0 to 10), we can pick out the very good and very bad demonstrations, by put a upper and lower threshold on the grades. We could to choose some consistent thresholds for all action types, or choose threshold depending on how many good or bad samples we received. A good landmark threshold for good and bad could be 6 and 3. We, therefore, can use any demonstrations that got higher than 6 to be additional positive samples, and ones that have grades lower than 3 as negative samples. The progress learner would be updated with those samples, and new demonstrations would be generated with the same initial configurations as old demonstrations.

  The reason we do not use the demonstration that got some medium score is that they are not obviously bad or good. Many of them start as a good demonstration, but make some mistake at the end, as we have seen at Table 5.12. Technically speaking, they are still "bad demonstrations", but how to use them as negative samples with cold feedback method is not obvious.

- *Hot feedback*: The 2-D simulator could be switched to interactive mode (as discussed in Section 2.4.1. In this interface, one can choose to loop from the beginning of a planned demonstration to the end, and choose an action replacement for one that is not appropriate.

  In this setup, we will assume that *hot feedback* information is additional to *cold feedback* information. The reason is that we have already been provided with the scores for each demonstration from the annotators, and we can pick ones that got low scores to analyze in the interactive visualizer, and make correction upon that. This approach reduces the labor cost to reanalyze and correct all saved demonstrations.

To test whether the newly learned model performs better than the original model, instead of using human evaluation, I will use the automatic evaluation methods shown in Section 5.3.2.1. This method will allow us to evaluate quickly effects of updating the original progress learner.

The reason I am looking for a way to improve the progress learner, and as consequence, improve the searching and RL algorithms depending on it, is that when analyzing sample demonstrations planned by one-step beam search algorithm, there are some demonstrations that finally get a high progress score, but fails to capture the meaning of the intended actions. Examples are shown in Fig 5.24. Both demonstrations, planned on discrete planning space, have final progress value in the range of 0.8 to 0.9, but fails to capture "Slide Around". In fact, the moving block only makes it half way in two opposite rotation directions.

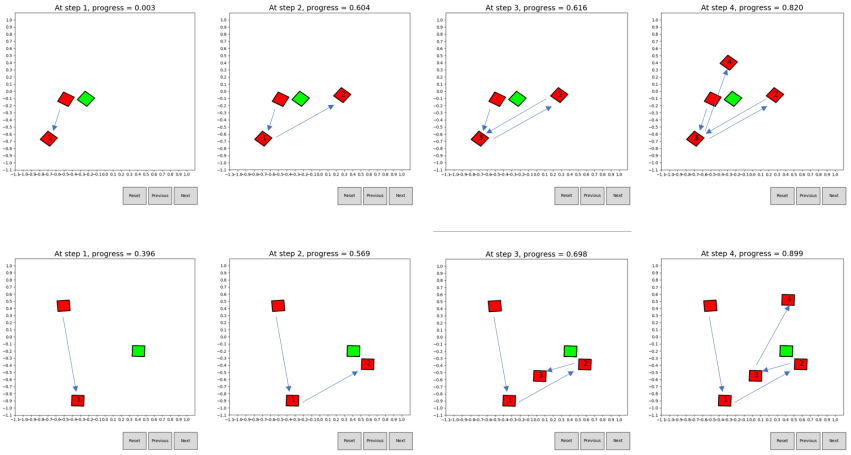


FIGURE 5.24: Two examples of demonstrations that the red block change the direction from counter-clockwise to clockwise

Followings are details of the algorithm to incorporate cold feedback into the progress learner:

input : A learned progress function f (Section 2.3.2) for one action, an update epoch number n, a learning rate θ
A list of demonstrations l created by the learner, rated as good or bad.
output: An updated (and hopefully) improved learned progress function
for *Epoch* $k \leftarrow 1$ **to** n **do**
  for *Demonstration* $D_i, i \leftarrow 1$ **to** l **do**
    Extract feature from the end of D_i as X;
    if D_i *is good* **then**
      Corresponding label $y = 1$;
    end
    else
      Corresponding label $y = 0$;
    end
    Update f with sample (X, y);
  end
end

Algorithm 8: Incorporating cold-feedback algorithm

For hot feedback, it is more difficult to decide the values of progress functions for different steps. Followings are details of the algorithm to incorporate cold feedback into the progress learner:

input : A learned progress function f (Section 2.3.2) for one action, an update epoch number n, a learning rate θ
A list of difficult setups l.
output: An updated (and hopefully) improved learned progress function
for *Setup* $S_i, i \leftarrow 1$ **to** l **do**
  Run greedy search with S_i in interactive simulator;
  Correct step **for** *Epoch* $k \leftarrow 1$ **to** n **do**
    Extract feature from the end of D_i as X;
    if D_i *is good* **then**
      Corresponding label $y = 1$;
    end
    else
      Corresponding label $y = 0$;
    end
    Update f with sample (X, y);
  end
end

Algorithm 9: Incorporating hot-feedback algorithm

Results:

Because of time limitation, in this section I will only present results from an experiment with Slide Around. This action, which has on average more number of steps than other actions, is of more interest in reinforcement learning framework. In this experiment, we will see the effect of updating the learned progress function with two sources of *cold feedback*: from human evaluation score and from automatic scorer (Section 5.3.2).

In particular, among 30 demonstrations of "Slide Around" that I have generated and sent to annotators to evaluate, 19 demonstrations got good scores, indexed with 0, 2, 3, 4, 6, 8, 9, 11, 12, 16, 17, 20, 21, 23, 24, 25, 27, 28 and 29, 4 demonstration got bad score, indexed with 10, 13, 18, 19. I divided them into two sets, one that has index smaller than 15, and the remaining. Using the *cold feedback* Algorithm 8, I update the original progress function to create one updated functions. The original one and the updated ones are all used to generate new demonstrations for another set of 30 setups (The first 30 demonstrations that I have created are given to annotators for human evaluation, reported at Figure 5.11. The second 30 setups are different initial setups, created at Section 5.3.3.1, and used only for machine evaluation. Here, again, *setup* means the starting position of a demonstration).

We can also use the automatic scorer to select the good and bad demonstrations from the original set of 30 setups, update the model, and apply the learned model on the second 30 setups. Result is given at Table 5.13. Notice that for continuous algorithm, I ran the experiment twice and average the scores (the reason is because continuous sampling has higher variance).

Progress function	Search algorithm			
	Greedy		One-step back up	
	Continuous	Discrete	Continuous	Discrete
Original model	0.19	0.15	0.16	0.20
Human-Feedback-Updated	**0.40**	**0.20**	0.58	0.42
Auto-Feedback-Updated	0.38	**0.20**	**0.60**	**0.65**

TABLE 5.13: Averaged score of 30 setups for action type = *Slide Around* with 4 different search algorithms. 3 different models of progress functions are compared: Original model, Update model with human feedback (Human-Feedback-Updated), Update model with automatic feedback (Auto-Feedback-Updated)

As reflected in Figure 5.13, there is not much difference between the result from two variances. Result for the method using automatic feedback performs a little bit worse with greedy algorithms, but performs better with backup algorithms. This just reflects different selections of bad and good demonstrations to update the model. It only proves

that the automatic scorer can also be used to improve the learned progress function. If we use them in a bootstrapping loop, we can improve the learned progress function to generate a policy performing as well as the expectation from the automatic scorer. We will not investigate further at that direction.

We will investigate whether use of hot feedback could improve performance, but not for the original model, but the model that has been improved with cold feedback (for a stronger baseline). Again we will start with selecting bad demonstrations from the first 30 setups, and apply updated model with the second 30 setups. In particular, using Human-Updated model, I ran greedy search on continuous space twice, and picked out 5 setups that generates lowest demonstrations scores. I reloaded them on interactive mode, making correction, updating the model following Algorithm 9, saving it into a new model file. Using that updated model and run searching algorithms over the second set of setups give the results as reported at Figure 5.14.

Progress function	Search algorithm			
	Greedy		One-step back up	
	Continuous	Discrete	Continuous	Discrete
Human-Feedback-Updated	0.40	0.20	**0.58**	0.42
Hot-Feedback-Updated	**0.5**	**0.37**	0.44	0.30

TABLE 5.14: Averaged score of 30 setups for action type = *Slide Around* with 4 different search algorithms. Comparison between the updated model with human feedback (Human-Feedback-Updated) and the model updated with hot feedback (Hot-Feedback-Updated)

Surprisingly, the updated model performs worse when paired with backup algorithm. This is rather counter-intuitive, but could be explained as following: the way we inspect and update the model step by step has made the model strongly favors the greedy algorithms. This issue might not be undesirable, as we always want the faster and simpler algorithm (greedy) to perform well.

5.3.5.1 Mini conclusion

In this subsection, we have gone over a few different techniques to improve learned models in an interactive way. Two techniques, *cold-feedback* and *hot-feedback*, and their relevance to CwC project, are addressed. While the methods are currently executed in a 2-D simulation environment, they could be extended to work with real interactive environment as well.

We have seen these methods could be used to improve the baseline model, with different advantages. *Cold-feedback* methods, which focus only on whether the final step of a demonstration is good or bad, has advantage when run with backup algorithms, while *Hot-feedback* methods, improving the progress function on each demonstration step-by-step, favor the greedy algorithms.

We also have seen that we do not need large amount of data to update the model. For *cold-feedback*, we have used about 20 feedback scores, while for *hot-feedback*, we have made correction to 5 different setups. While this is still not one-shot learning, the number of demonstrations are small enough for interactive learning.

5.4 Conclusion

As I have included a mini-conclusion after each experiment, in this section, I will just summarize the lessons that I have learned over the course of setting up and solving this problem.

The first lesson is that for the problem at hand, true RL methods, i.e. our policy gradient methods, are not necessarily better than simpler search algorithms. Even though we have successfully run policy gradient methods, producing satisfying action policy, it incurs high overhead. The most problematic issue is formalization of states and actions: firstly, we need domain knowledge to carry out space discretization in the manner as I have described; secondly, the way we represent states can not capture action trajectory from the beginning of the demonstration. The second issue is computational overhead. Running RL methods is often time consuming. Probably for this problem, heuristic search algorithms are more appropriate.

The second lesson is that we can improve learned models by interactively guiding the learning agents. This direction of research might be of high interest for machine learning community. More research needs to be done so that we can use this learning framework in a generic machine learning setup.

Last but not least, one issue with the capturing setup described in this chapter is that we do not have natural language input. All of our actions are constrained in a frame-based manner. Therefore, we move forward to the next chapter, flipping the learning problem on its head and considering it under another perspective.

Chapter 6

Performing action by observing artificial demonstrations

In this chapter, we will flip the problem on its head, by considering a different learning setup. In Chapter 5, we generate real observations of actions by first giving descriptions of the actions, then capturing demonstrators performing the actions. In this chapter, I will generate artificial demonstrations of actions in a search space, which I term *maze traversal space*, and record observations into video snippets. These videos are posed to annotators to solicit textual description of the trajectory in the visual scene.

The learning objectives in both setups are the same: given textual inputs describing actions (*instructions*), plan actions (sequence of action steps) on a grounded visual environment. In this setup, which I would term *artificial demonstration* to contrast with the previous setup, which I term *real demonstration* setup, I will generate demonstrations on my 2-D simulator. This approach usually allows quicker data collections, and also allows us to solicit instructions of natural language utterances (c.f. the previous approach, where we predefined a set of action types with binding arguments).

In training phase, we will learn a machine learning model by feeding into it a parallel corpus of instructions and corresponding sequences of actions as video captures. In evaluating phase, we will pose to the machine learning model a textual instruction, and the starting configuration of visual environment (*maze puzzle*). The task would be for the learned model to direct a selected block to traverse the maze toward some final target. The evaluation objective would be for the planned trajectory follows as close as possible to the intended trajectory.

6.1 Models

Firstly, in this setup, we could not use directly a progress function like in Chapter 5, because the textual instructions are in natural language form, preventing us to create different progress functions for different action types. Therefore, we will use a model that generates sequence of actions directly from the sequence of instructional utterances. Not surprisingly, this model is called Sequence-to-Sequence (or Seq2Seq) model. The input sequence is the stream of words in the input instruction, and the output sequence will be a sequence of the following actions: *left, right, up, down, STOP*.

In this section, I will first describe Seq2Seq models used for generation of actions from textual instructions. My focus will be an advanced form of Seq2Seq models, called Attention RNN, described in Section 6.1.2. These models are recently developed for neural machine translation (Wu et al., 2016) and image caption generation (Xu et al., 2015), achieving state-of-the-art results for both tasks (Note that Image-to-caption model is not exactly Seq2Seq, but instead maps 2-D convolutional features to sequence, but these two models can all be classified as Attention RNN, because their decoder components are identical, i.e. an attention RNN decoder).

6.1.1 Seq2Seq RNN for controlling

So far, in this dissertation we have met with a simple form of RNN model, which is a function from a sequence of features to a single numerical data, i.e. the action progress value. This is the typical setup for classification and regression. A more advanced form of RNN allows ones to produce a sequence from a sequential input. These models are commonly called sequence to sequence (Seq2Seq) models (actually the multi-layer LSTM implicitly uses Seq2Seq on the lower layers).

The simplest form of Seq2Seq models has *temporal alignment* between the input and output sequences, e.g. one output for one input at each time step, or the sampling rates of input and output are resonant. This setup could be used for *classical* controlling problem, e.g. there is a controlling signal recorded every second, and observations (e.g. of a drone) are recorded at 30 times per second. Translation from a textual instruction to controlling actions is usually not temporal aligned, though both the instruction and output actions follow a left to right narrative order.

A more appropriate form of Seq2Seq models for non-aligned sequence translations are **Encoder-decoder** models. In a simplest form, encoder-decoder models are two RNNs stacking on top of each other (Figure 6.1).

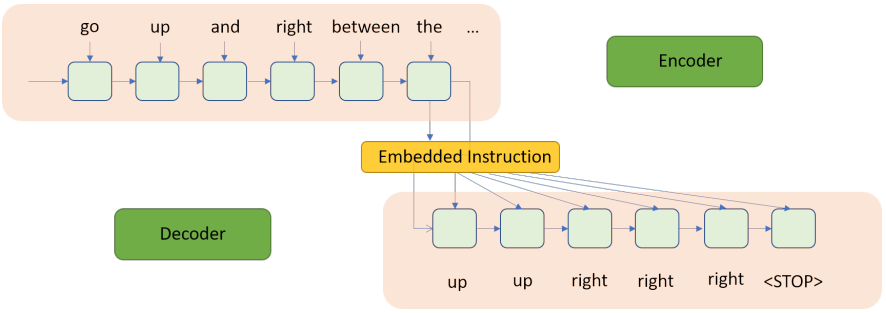


FIGURE 6.1: Encoder-decoder model

The encoder is a sequence-to-one RNN that takes a sequential input and produces an embedded representation of the input instruction, while the decoder is an one-to-sequence RNN that digests the embedded representation as inputs for all time steps, while the final state from the encoder is (optionally) passed on to be the initial state for the decoder.

6.1.2 Attention RNN

Attention RNN is developed firstly as an extension to the encoder-decoder framework in neural machine translation (Bahdanau et al., 2015). What is called *attention* is a trick applied to the embedded representation. Instead of having only one embedding as the information mediator between the encoder and decoder (shown in Figure 6.1), we will have multiple embedded representations (called *annotations*), one for each input time step, as shown in Figure 6.2. Prediction at each output time step is conditioned on a function over all annotations. In this dissertation, we will use condition of Bahdanau attention style with a **weighted sum** of input annotations (shown in Figure 6.2 as $\bigoplus$ operator). Weights are usually calculated as functions of current decoder state and each annotation (higher weight is depicted with more pronounced color in Figure 6.2). Intuitively, this method allows alignment between words (or phrases) in the input and the output so that words in input sentence that are more predictive for the current output step have more contribution to the prediction.

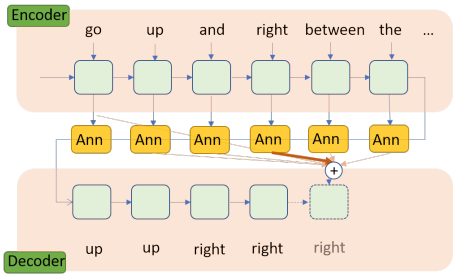


FIGURE 6.2: Attention RNN model (Bahdanau additive attention style)

Attention RNN is also a flavor of attention methods, which have garnered huge interest in machine learning community in recent years. Attention methods have been hailed as the "holy grail" to improve all deep learning networks, including Convolutional neural networks (CNNs) and Recurrent neural networks (RNNs) (See (Vaswani et al., 2017)). The attention trick has also been related to Neural Turing machines (Olah and Carter, 2016; Graves et al., 2014) and Neural Programmer (Neelakantan et al., 2016), two powerful frameworks for learning of execution (programs). In this dissertation, we would not dive deep into the working mechanism of attention (please refer to (Bahdanau et al., 2015) for mathematical formulas) as well as the recent discussion of attention methods.

Instead, we will just focus on how to adapt this method for the task at hand. While the previously described tasks have one input and one output, our task has two different inputs, one textual and one being the visual grounding. Applying the neural translation method directly (text sequence to text sequence) is tantamount to ignoring visual grounding. We, however, can still use it as a baseline method.

To incorporate visual grounding into Seq2Seq method, we can condition action prediction at each time step of the decoder with the current environment state (cf. (Tanti et al., 2018) for use of image input for caption generation). For this method, we can feed the current environment state as a visual image at each time step, concatenated with the input to the decoder (the weighted annotation). Output of the prediction (selected output action) is sent to the controller component, moving the purple block accordingly; we again feed the new environment state to the decoder to infer for the next step. This loop is presented at Figure 6.3.

Notice that in training phase, we ignore the controller loop, and use directly the frames from the visual input to feed into the inputs of the decoder. We will only address this issue in the conclusion part, under the term of *exposure bias*.

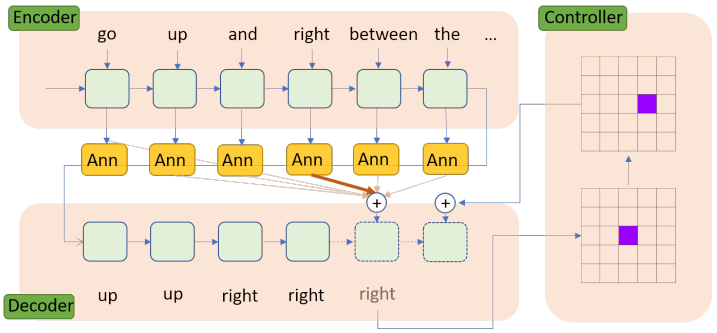


FIGURE 6.3: Seq2Seq model with controlling loop from visual state to input of decoder)

Common practices

Some common practices that I used in training Seq2Seq attention models:

- Beam search and greedy search in Seq2Seq models: analogous to beam search and greedy search in RL framework. The difference lies on the objective to optimize in search. While search algorithms as discussed in 2.4.2.1 aims to maximize the objective of the progress function, beam and greedy search in Seq2Seq models aims to maximize the probability of the output sequence. We need to search over the output space because in RNNs, there is no method to run efficient sequence inference, such as Viterbi method, to find sequence with global maximum probability. The reason is because cell states are propagated over time, and therefore, probability distribution of actions is not the same for different inference time (or the sequential model is not *stationary*). It is also noted that the sequence probability objective is related but not the same as the training objective (i.e. perplexity), as well as the external evaluation objective, that we will discuss shortly.

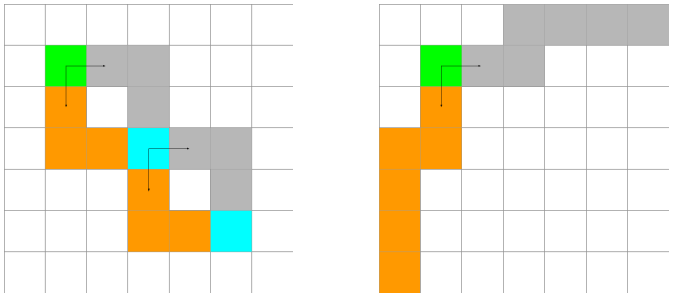
  All of my experimental models would use greedy search for simplicity.

- Word embedding: we need to turn from words into feature vectors before feeding into neural networks, by a method called word embedding. Because the size of our vocabulary is small, we will train our own word embedding. Even though we can use the same word embedding for input and output (shared word embedding), it usually does not hurt model's performance by training separate embeddings for source and target vocabularies. In fact, using shared word embedding might even hurt model's performance, because an instruction phrase like "move to the left of the green rectangle" might actually mean going to the right direction, if the green

rectangle lies further to the right. Using the same embedding for the word "left" at input and the word "left" at output will be incorrect in this case.

- Internal evaluation: is a measure of objective that is used in gradient optimization method. In typical Seq2Seq models, we usually use **perplexity** as the internal evaluation. The formula of **perplexity** is 2^H where H is the averaged number of bits needed to code the next symbol in the output sequence. At each time step, this value is calculated by the cross-entropy between the predicted symbol probabilities and the target probabilities. Smaller value of perplexity means higher predictive power of the model, and a model has absolute confidence in prediction when its perplexity (over some evaluation corpus) equals 1.

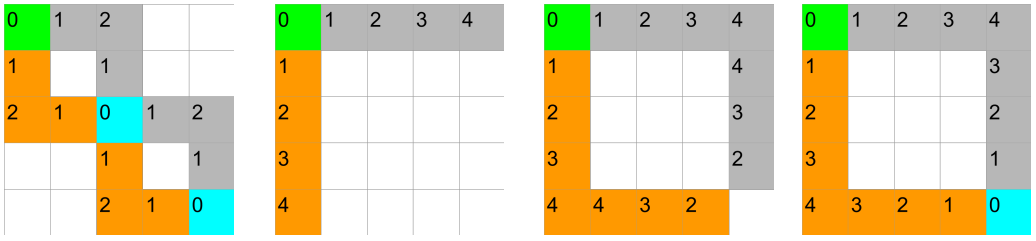
- External evaluation: is a measure of objective that usually reflects better qualitative comparison between an output candidate and a reference. For machine translation problem, BLEU score (Papineni et al., 2002) is an appropriate external evaluation, as it accounts for phrase matching between candidate and references outputs. For the problem at hand, an applicable external evaluation will be one that reflects whether two trajectories are convergent or divergent. In Figure 6.4, we see an example of this qualitative distinction that we are not be able to capture in internal evaluation method. In both figures, the orange trajectory corresponds to the reference path (one used to generate the textual instruction), and the gray one is to the inferred (candidate) trajectory. At Figure 6.4 A, the reference actions are *down, down, right, right, down, down, right, right*, the candidate (inferred) actions are *right, right, down, down, right, right, down, down*, i.e. the prediction is wrong at every time step. Same applies for Figure 6.4 B, but we can qualitatively judge that the candidate trajectory in A matches its corresponding reference better than in B.



(A) These two paths are convergent (B) These two paths are divergent

FIGURE 6.4: While the internal evaluation is agnostic to *convergent* versus *divergent* paths, external evaluation measure accounts for this distinction. Starting position is color *green*, intercepting positions are *blue*, candidate path follows the gray cells, and reference path follows orange cells

A good external evaluation score between two trajectories is the averaged distance from each cell of any trajectory to its closest neighbor on the other trajectory. Let's call that a NEIGHBOR score. Smaller evaluation score means more similar trajectories. Two identical trajectories has evaluation score of 0. Detailed calculation of a few sample pairs of trajectories is presented at Figure 6.5.



(A) Score = 16/15 = 1.07 (B) Score = 20 / 9 = 2.2 (C) Score = 38/15 = 2.53 (D) Score = 32/16 = 2

FIGURE 6.5: NEIGHBOR scores of some sample trajectories. Value on each cell is the distance to the closest cell on the other trajectory. Common cells have the value of 0.

This evaluation method provides an opportune measurement for the problem at hand, especially to make the distinction between convergent and divergent trajectories, as we see in the scores of Fig. 6.5 A versus Fig. 6.5 B. Between B and D, one can argue that trajectories in D are correcting B, and therefore D's score should be lower than B's score. C score is higher than B score, which is unexpected (we might expect the score to be somewhere between score of B and D). However, an evaluation method that is justifiable in every cases is difficult to find.

6.2 Data preparation

Generating demonstrations:

Demonstrations were generated simply by creating a grid of size 15x15, than scattering 3 rectangles, 3 triangle and 2 L shapes on top. To make the maze puzzle a little bit more interesting, I colored the shapes with a few different simple colors like yellow, red, green, blue and purple. The moving block is always a purple cell.

The source and target locations of purple block are so generated that they are of at least 15 moving step distance to each other. A path from the source to the target location is generated. For the trajectory to be less monotonous, this path is actually not the shortest path from the source to the target (a shortest path is first generated, than a blocking wall is generated to block this path, than a second path is generated to avoid the blocking wall). Details of demonstration generation are provided in the code.

Soliciting instructions:

There are 3 folders, named [0,1,2], each with 100 puzzles. Annotators were asked to choose one folder to give instructions by speaking to an audio recorder. They were asked to tell the name of the maze puzzle first, then give their instruction. Annotators are mostly master students in Brandeis' Computational Linguistic program.

In total, there are 15 annotators submitting their recorded audios. 8 annotators submitted for the first 100 puzzles, 8 other submitted for the second 100 puzzles, while only 1 annotator submitted for the third 100 puzzles. These were transcribed by an annotator into text. Because of time limitation, for each puzzle indexed from 0 to 199, 4 annotations were transcribed, for one indexed from 200 to 299, the only one was transcribed.

The following snapshots are taken from a puzzle indexed 0. 4 transcribed instructions given for this specific maze puzzle are also provided.

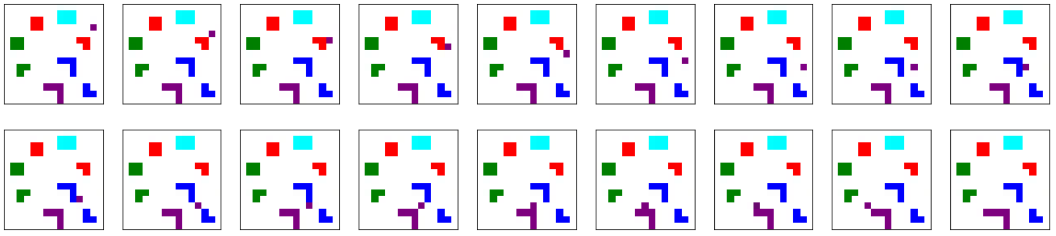


FIGURE 6.6: Frames of a recorded video snippet

move the purple square down on the right side of the blue L and red L then move the purple square left between red L and purple L and it ends at the left side of the purple L
move the purple block down then left so that's directly between the blue L shape and the red L shape then move it down again until it reaches the top of the red L shape then move it straight left until it reaches 1 cell beyond the purple L shape
move the purple block down until it is in position between the 2 red L shapes and move it to the left until it is in front of purple L shape then move it down 1 block space
move the purple block down on the right of the blue L left and between the 2 red Ls to the left of the purple L

I use all textual instructions of the first 200 puzzles for training and validation data so that one instruction is used for validation, and the remaining are used for training. Therefore, training and validation share the same set of puzzles. For testing, instructions of the last 100 puzzles are used (puzzles that are not seen in training).

6.3 Evaluation

In this section, I will evaluate two different methods: the baseline method of attention RNN *without* visual grounding and the improved model *with* visual grounding. Three evaluation measures will be provided: perplexity on the training and evaluation dataset, NEIGHBOR score on the evaluation and testing dataset and effective output length. Notice that effective output length is the number of output actions that are *legal* in the visual environment, i.e. it does not move the target block outside of the playground, or toward a cell occupied by an obstacle object.

The following chart and table shows results of running the baseline method with $num\_units = 128$, $optimizer = StochasticGradientDescent$, $learning\_rate = 0.2$, $num\_train\_steps = 2000$. Based on perplexity scores and NEIGHBOR scores in this running, early-stopping is a good strategy, as it shows that after step 1400, both evaluation perplexity and NEIGHBOR score increases. Output effective length is much shorter than the average reference length, because without *visual grounding*, the moving block *blindly* crashes into obstacles or walls.

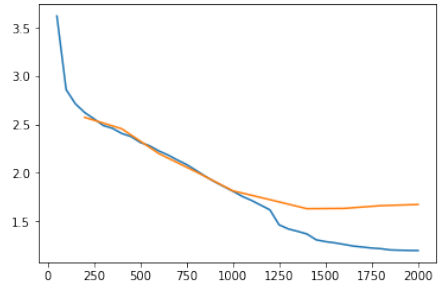


FIGURE 6.7: Perplexity of training (blue line) and evaluating (orange line) over 2000 steps (each step is one mini-batch update)

Step	NEIGHBOR		Avg. length	
	Eval	Test	Eval	Test
1400	4.04	4.10	10.4	8.84
1600	4.18	4.12	10.33	8.6
1800	4.15	4.00	10.06	8.94
2000	4.13	4.01	10.62	8.94
Ref.			18.24	17.89

FIGURE 6.8: External evaluation for different values of steps

6.4 Conclusion

In this chapter, I have given a brief discussion on a newer framework for learning of execution, namely sequence to sequence model for controlling. A technically challenging controlling problem is proposed, and we have applied Seq2Seq method with different levels of complexity to solve it. Contrasting the methods we have used in Chapter 5 and this chapter allows us to reflect on the similarity and difference between these methods. Even though I do not apply them for the same problem, we will shortly see that they are appropriate for different kinds of planning problems.

To a certain extent, Seq2Seq for execution is analogous to reinforcement learning algorithms. The progress value is intrinsically coded into the probability of the *STOP* signal at each time step, i.e. when this (progress) value is high, we decide to complete the action. The function that the Seq2Seq'2 decoder calculates has exactly the same signature with a learned policy in RL, i.e. it takes in a state that combines the encoded information of the instruction and current planning state (hidden state of Seq2Seq decoder cell vs formalized location/orientation environment states in RL), and outputs a probability distribution of actions.

Regarding formalization of states in these methods, we have seen that we formalize reinforcement learning states using domain knowledge, i.e. we know beforehand that the actions we want to learn only involve relative positions between objects. In contrast, states in Seq2Seq methods are of "black box" nature, i.e. we do not know meaning of different features in hidden states of Seq2Seq's decoder. Moreover, an interesting difference is that in RL methods, we found that mapping from continuous states to

discrete states through feature extraction methods (e.g qualitative spatial reasoning) allows RL algorithms to work more efficiently, whereas in Seq2Seq methods, we actually abstract the discrete states of output (chains of discrete actions) into continuous multi-dimensional hidden states.

Even though we have discussed strength of RL methods in Section 5.4, in this section, we will reiterate some advantages of reinforcement learning methods (e.g. REINFORCE) over Seq2Seq methods:

- Adaptation to novel environment setups: we can see that in Chapter 5, the original environment for capturing demonstrations is quite different from the environment for generating demonstrations. Moreover, even if we add other constraints into environment setup, the RL and searching methods will still work. For example, if we change size or shape of the playground (See Section 5.1.1) or add more objects into it, algorithms belonging to RL types can accommodate easily.

- Ability to learn skills with limited training dataset: by formalizing states and action space with domain knowledge, we can make reinforcement learning to work with limited amount of training data.

- Can be trained to optimize a *global* sequence objective, rather than *local* step objectives: we have seen in Section 6.1.2 that for Seq2Seq models, we usually use two different evaluation methods, cross-entropy is used for training because it is differentiable, and external evaluation is better for qualitative judgment but is not differentiable. In contrast, target of optimization in reinforcement learning is global objective, such as the progress function in Section 5.3.1, which is not differentiable for each action step.

- Avoidance of *exposure bias*: *exposure bias* is a problem of Seq2Seq models that reinforcement learning model does not have, regarding the difference between inputs of decoder in training versus inference phase. Even though Figure 6.2 shows that output of the decoder at one time step is fed as input to the next step, this only applies for inference phase. In training phase, we actually disregard the output from the previous time step, and use correct label to feed into the next step. In other words, the model is only exposed to correct data in training, but has to make prediction sequentially in inference. While it can be mitigated by a smart learning schedule alternating between feeding correct labels and passing previous predictions, this is inherently not an issue of reinforcement learning framework, where the model generate, then learn from its own samples.

Advantages of the Seq2Seq methods are the followings:

- Ability to learn skills with very large state spaces: given enough amount of data, Seq2Seq can be used for very large searching space. For problems such as Neural Machine Translation, the searching space for output has the size of V^T where V is the size of vocabulary (10^4), and T is output's average length (30), which easily makes it larger than the number of particles in the universe (estimated at 10^{86}). For RL methods, this space would be too large to search over, if the policy is started at random.

- Ability to learn skills without domain knowledge to formalize states and actions, therefore it can be applied to a wide spectrum of problems. This is probably the most important aspect of Seq2Seq model that makes it so popular and successful.

Recently, there are novel machine learning methods that can take advantages of both Seq2Seq and Reinforcement learning. One is called MIXER (Mixed Incremental Cross-Entropy Reinforce) (Ranzato et al., 2016). It is an optimization method that combines both cross-entropy and REINFORCE. In particular, it first learns a baseline policy with Seq2Seq loss function, than starts optimizing this policy with external evaluation objective using REINFORCE. This approach has been shown to work for three classical Seq2Seq tasks: machine translation, text summarization and image captioning. Another notable approach is called Seq2Seq Learning as Beam-Search Optimization (Wiseman and Rush, 2016), adopting beam search method to rank output sequences according to external evaluation scores.

This mixing between reinforcement learning methods and Seq2Seq methods is intellectually exciting and technically challenging. To the best of my knowledge, there is not yet any comparison study between these two methods, so there is no conclusive answer to the question of which method works better for the problem at hand. Probably this problem setup of translating from instructions to commands with visual grounding is an appropriate playground to compare between these two methods. This will be left for future work.

The problem of turning natural language instructions into commands with visual grounding is relatively new, and could be of interest for research community in AI and Natural language processing alike. One of the barriers for research in this direction is the lack of interesting dataset. Interactive games that require cooperative behaviors between

users to control characters might be a good resource to tap into. I would also leave that direction for future consideration.

We have not yet addressed in this chapter what will happen if the instruction is incorrect, i.e. there is no chain of actions that could satisfy the input instruction. There are a few ways to handle this issue,. An approach that uses domain knowledge and does not use machine learning can find mentions of objects in the instruction, than locating them in the visual environment; one can then find rules regarding relative positions between starting position of the moving block and the obstacles to decide if an instruction is executable or not. A machine learning approach could generate unlikely configurations semi-automatically from a given instruction. In training, it can pair an instruction with an unlikely configuration as a negative sample. In inference, it can produce a "NOT EXECUTABLE" to signal that it could not move forward given an incorrect instruction.

This chapter is brief, and there are issues have not been addressed at length, but hopefully it has shed a light into our problem from another perspective. I also hope that this conclusion has included useful discussion for future explorations.

Chapter 7

Conclusion and Future direction

7.1 Roadmap

Progress to date has been on the development of Event Capture and Annotation Tool and the machine learning algorithms used in this work:

- **September, 2015** — Start working on CwC project, devising the concept of an action/event learning toolkit.
- **December, 2015** — First built of ECAT, with simple annotation tools i.e. marking object boundary, event annotation.
- **March, 2016** — Second built of ECAT, integrated with 3-D tracking and mapping functionality for several object types (human rigs, blocks, table).
- **May, 2016** — Third built of ECAT, first attempt to integrate VoxML semantic structure with ECAT. ECAT is presented at ISA 2016.
- **August, 2016** — Used ECAT as an annotation tool for movie dataset (Do, 2016). See section 3.2 for a brief introduction.
- **November, 2016** — Implementation of an event classifier with LSTM and CRF 4.2.2.

- **March, 2017** — Presented ECAT and the event classifier at AAAI-SS 2017 in the framework of learning object and event representation from multimodal resource (Pustejovsky et al., 2017).
- **April, 2017** — Presented ECAT and the event classifier at ESANN 2017 (Do and Pustejovsky, 2017a).
- **August, 2017** — Developed feature extractor based on qualitative reasoning method. Presented the event classifier with QSR features at Qualitive Reasoning workshop (Do and Pustejovsky, 2017b).
- **November, 2017** — Developed 2-D simulator, searching and RL algorithms.
- **February, 2018** — Completion of the 3-D visualizer. Defense of the proposal.
- **March, 2018** — Completion of lab-scale human evaluation on 2-D.
- **April, 2018** — Completion of experiments 5.3.3. Analysis of experimental results.
- **May, 2018** — Completion of experiment 5.3.4 and experiment 5.3.5. Analysis of experimental results.
- **June, 2018** — Write-up complete with experimental data, results, and conclusions.
- **June, 2018** — Final thesis defense

7.2 Post-Thesis

There are two different lines of research that can be considered extension from this framework. One line of research involves learning mechanism for more complex actions, such as "make a row from given objects", and one line of research involves learning manner aspect of actions.

Learning complex actions from simpler actions requires additional semantics framework for objects and actions. For example, to learn "make a row from given objects", and the learner observes rows of length 2 and length 3, the learner needs to be equipped with the concept of *repetition*, the concept of composite object made from elementary

objects (e.g. size and shape of the composite object), and other abstract concepts, such as object axis.

Learning manner aspect of actions requires fine-grained treatment of object affordances. For example, for learner to make the distinction in performing "rolling a bottle" and "sliding a bottle", we need to equip it with reasoning mechanism related to how an object's pose and position dictating its affordances. These questions, hopefully, can be answered by making reference to a developing language for objects and events in our lab named VoxML (Pustejovsky and Krishnaswamy, 2016).

7.3 Conclusion

Bibliography

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *International Conference on Machine learning (ICML)*, page 1. ACM, 2004.

Eren Erdal Aksoy, Minija Tamosiunaite, and Florentin Wörgötter. Model-free incremental learning of the semantics of manipulation actions. *Robotics and Autonomous Systems*, 71:118–133, 2015.

Muhannad Alomari, Paul Duckworth, Nils Bore, Majd Hawasly, David C Hogg, and Anthony G Cohn. Grounding of human environments and activities for autonomous robots. In *In Proceedings of 17th International Joint Conferences on Artificial Intelligence (IJCAI-17)*, 2017.

UF Andrew, D Mark, and D White. Qualitative spatial reasoning about cardinal directions. In *Proc. of the 7th Austrian Conf. on Artificial Intelligence. Baltimore: Morgan Kaufmann*, pages 157–167, 1991.

Katharina Antognini and Moritz M Daum. Toddlers show sensorimotor activity during auditory verb processing. *Neuropsychologia*, 2017.

Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

Minoru Asada, Eiji Uchibe, and Koh Hosoda. Cooperative behavior acquisition for mobile robots in dynamically changing real worlds via vision-based reinforcement learning and development. *Artificial Intelligence*, 110(2):275–292, 1999.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

Collin F Baker, Charles J Fillmore, and John B Lowe. The berkeley framenet project. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1*, pages 86–90. Association for Computational Linguistics, 1998.

Mikhail Belkin, Siyuan Ma, and Soumik Mandal. To understand deep learning we need to understand kernel learning. *arXiv preprint arXiv:1802.01396*, 2018.

Mehul Bhatt, Hans Guesgen, Stefan Wölfl, and Shyamanta Hazarika. Qualitative spatial and temporal reasoning: Emerging applications, trends, and directions. *Spatial Cognition & Computation*, 11(1):1–14, 2011.

Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. Robot programming by demonstration. In *Springer handbook of robotics*, pages 1371–1394. Springer, 2008.

Gary Bishop, Greg Welch, et al. An introduction to the kalman filter. *Proc of SIGGRAPH, Course*, 8(27599-23175):41, 2001.

Daniel G Bobrow. *Qualitative reasoning about physical systems*, volume 1. Elsevier, 2012.

Robert Bogue. Domestic robots: Has their time finally come? *Industrial Robot: An International Journal*, 44(2):129–136, 2017.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Michael Buhrmester, Tracy Kwang, and Samuel D. Gosling. Amazon's mechanical turk a new source of inexpensive, yet high-quality, data? *Perspectives on psychological science*, 6(1):3–5, 2011.

Sylvain Calinon and Aude Billard. Teaching a humanoid robot to recognize and reproduce social cues. In *Robot and Human Interactive Communication, 2006. ROMAN 2006. The 15th IEEE International Symposium on*, pages 346–351. IEEE, 2006.

Sylvain Calinon, Florent Guenter, and Aude Billard. On learning, representing, and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(2):286–298, 2007.

Anthony G Cohn and Jochen Renz. Qualitative spatial representation and reasoning. *Foundations of Artificial Intelligence*, 3:551–596, 2008.

Matthias Delafontaine, Anthony G Cohn, and Nico Van de Weghe. Implementing a qualitative calculus to analyse moving point objects. *Expert Systems with Applications*, 38(5):5187–5196, 2011.

Tuan Do. Event-driven movie annotation using mpii movie dataset. 2016.

Tuan Do and James Pustejovsky. Fine-grained event learning of human-object interaction with lstm-crf. *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*, 2017a.

Tuan Do and James Pustejovsky. Learning event representation: As sparse as possible, but not sparser. *International Workshop on Qualitative Reasoning (QR)), at IJCAI-17*, 2017b.

Tuan Do, Nikhil Krishnaswamy, and James Pustejovsky. Ecat: Event capture annotation tool. *Proceedings of ISA-12: International Workshop on Semantic Annotation*, 2016.

James Dougherty, Ron Kohavi, Mehran Sahami, et al. Supervised and unsupervised discretization of continuous features. In *Machine learning: proceedings of the twelfth international conference*, volume 12, pages 194–202, 1995.

Krishna SR Dubba, Anthony G Cohn, David C Hogg, Mehul Bhatt, and Frank Dylla. Learning relational event models from video. *Journal of Artificial Intelligence Research*, 53:41–90, 2015.

Michael Firman. Rgbd datasets: Past, present and future. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 19–31, 2016.

Abdellah Fourtassi and Michael C Frank. Word identification under multimodal uncertainty. *Proceedings of the 39th Annual Meeting of the Cognitive Science Society (COGSCI)*, 2017.

Christian Freksa. Qualitative spatial reasoning. *Cognitive and Linguistic aspects of Geographic space*, 63:361–372, 1991.

Christian Freksa. *Using orientation information for qualitative spatial reasoning*. Springer, 1992.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1019–1027, 2016.

Anja Gampe, Jens Brauer, and Moritz M Daum. Imitation is beneficial for verb learning in toddlers. *European Journal of Developmental Psychology*, 13(5):594–613, 2016.

Barbara P Garner and Doris Bergen. Play development from birth to age four. *Play from birth to twelve: contexts perspectives, and meanings*, 2006.

Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel Jesús Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

Y Gatsoulis, M Alomari, C Burbridge, C Dondrup, P Duckworth, P Lightbody, M Hanheide, N Hawes, and AG Cohn. Qsrlib: a software library for online acquisition of qualitative spatial relations from video. In *Workshop on Qualitative Reasoning (QR16), at IJCAI-16*, 2016.

György Gergely, Harold Bekkering, and Ildikó Király. Developmental psychology: Rational imitation in preverbal infants. *Nature*, 415(6873):755, 2002.

Tobias Gerstenberg, Liang Zhou, Kevin A Smith, and Joshua B Tenenbaum. Faulty towers: A hypothetical simulation model of physical support. *Proceedings of the 39th Annual Conference of the Cognitive Science Society*, 2017.

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

David Gunning. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2017.

Nick Hawes, Christopher Burbridge, Ferdian Jovan, Lars Kunze, Bruno Lacerda, Lenka Mudrová, Jay Young, Jeremy Wyatt, Denise Hebesberger, Tobias Kortner, et al. The strands project: Long-term autonomy in everyday environments. *IEEE Robotics & Automation Magazine*, 24(3):146–156, 2017.

Junhu He. *Robotic In-hand Manipulation with Push and Support Method*. PhD thesis, University of Hamburg, 2017.

Lisa Anne Hendricks, Zeynep Akata, Marcus Rohrbach, Jeff Donahue, Bernt Schiele, and Trevor Darrell. Generating visual explanations. In *European Conference on Computer Vision (ECCV)*, pages 3–19. Springer, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Anthony Hoogs and AG Amitha Perera. Video activity recognition in the real world. In *AAAI*, pages 1551–1554, 2008.

Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

Ray Jackendoff. *Semantics and Cognition*. MIT Press, 1983.

Xiang Jiang, Erico N de Souza, Xuan Liu, Behrouz Haji Soleimani, Xiaoguang Wang, Daniel L. Silver, and Stan Matwin. Partition-wise recurrent neural networks for point-based ais trajectory classification. In *25th European Symposium on Artificial Neural Networks, Computtional Intelligence and Machine Learning*, pages 529–534. ESANN, 2017.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning (ICML)*, pages 2342–2350, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Hema S Koppula and Ashutosh Saxena. Anticipating human activities using object affordances for reactive robotic response. *IEEE transactions on pattern analysis and machine intelligence*, 38(1):14–29, 2016.

Hema Swetha Koppula, Rudhir Gupta, and Ashutosh Saxena. Learning human activities and object affordances from rgb-d videos. *The International Journal of Robotics Research*, 32(8):951–970, 2013.

Nikhil Krishnaswamy. *Monte Carlo Simulation Generation Through Operationalization of Spatial Primitives*. PhD thesis, Brandeis University, 2017.

Nikhil Krishnaswamy and James Pustejovsky. Voxsim: A visual platform for modeling motion language. In *Proceedings the 26th International Conference on Computational Linguistics: System Demonstrations*, 2016.

Nikhil Krishnaswamy and James Pustejovsky. Do you see what i see? effects of pov on spatial relation specifications. In *Proc. 30th International Workshop on Qualitative Reasoning*, 2017.

Nikhil Krishnaswamy, Pradyumna Narayana, Isaac Wang, Kyeongmin Rim, Rahul Bangar, Dhruva Patil, Gururaj Mulay, Ross Beveridge, Jaime Ruiz, Bruce Draper, et al. Communicating and acting: Understanding gesture in simulation semantics. In *12th International Conference on Computational Semantics (IWCS), Short papers*, 2017.

Benjamin Kuipers. How shall we learn how to learn how to grasp? *ICRA Workshop on Representations for Object Grasping and Manipulation*, 2010.

Quoc V Le, Will Y Zou, Serena Y Yeung, and Andrew Y Ng. Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3361–3368. IEEE, 2011.

Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015.

Wenbin Li and Mario Fritz. Recognition of ongoing complex activities by sequence prediction over a hierarchical label space. In *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*, pages 1–9. IEEE, 2016.

Wei Liang, Yibiao Zhao, Yixin Zhu, and Song-Chun Zhu. Evaluating human cognition of containing relations with physical simulation. In *CogSci*, 2015.

Tomas Lozano-Perez. Robot programming. *Proceedings of the IEEE*, 71(7):821–841, 1983.

John H Maunsell and David C Van Essen. Functional properties of neurons in middle temporal visual area of the macaque monkey. ii. binocular interactions and sensitivity to binocular disparity. *Journal of Neurophysiology*, 49(5):1148–1167, 1983.

William E Merriman, Laura L Bowman, and Brian MacWhinney. The mutual exclusivity bias in children's word learning. *Monographs of the society for research in child development*, pages i–129, 1989.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.

Chris Olah and Shan Carter. Attention and augmented recurrent neural networks. *Distill*, 2016. doi: 10.23915/distill.00001. URL http://distill.pub/2016/augmented-rnns.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics (ACL), 2002.

James Pustejovsky. Dynamic event structure and habitat theory. In *Proceedings of the 6th International Conference on Generative Approaches to the Lexicon (GL2013)*, pages 1–10. ACL, 2013.

James Pustejovsky and Nikhil Krishnaswamy. Generating simulations of motion events from verbal descriptions. *Lexical and Computational Semantics (* SEM 2014)*, page 99, 2014.

James Pustejovsky and Nikhil Krishnaswamy. Voxml: A visual object modeling language. *Proceedings of LREC*, 2016.

James Pustejovsky, Nikhil Krishnaswamy, and Tuan Do. Object embodiment in a multimodal simulation. *AAAI Spring Symposium: Interactive Multisensory Object Perception for Embodied Agents*, 2017.

Hossein Rahmani and Ajmal Mian. 3d action recognition from novel viewpoints. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1506–1515, 2016.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

Anna Rohrbach, Marcus Rohrbach, Niket Tandon, and Bernt Schiele. A dataset for movie description. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

Marcus Rohrbach, Sikandar Amin, Mykhaylo Andriluka, and Bernt Schiele. A database for fine grained activity detection of cooking activities. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1194–1201, 2012.

Amitava Roy and Sankar K Pal. Fuzzy discretization of feature space for a rough set classifier. *Pattern Recognition Letters*, 24(6):895–902, 2003.

Michael S Ryoo and JK Aggarwal. Ut-interaction dataset, icpr contest on semantic description of human activities (sdha). In *IEEE International Conference on Pattern Recognition Workshops*, volume 2, page 4, 2010.

Iulian Vlad Serban, Tim Klinger, Gerald Tesauro, Kartik Talamadupula, Bowen Zhou, Yoshua Bengio, and Aaron C Courville. Multiresolution recurrent neural networks: An application to dialogue response generation. In *AAAI*, pages 3288–3294, 2017.

Amir Shahroudy, Jun Liu, Tian-Tsong Ng, and Gang Wang. Ntu rgb+ d: A large scale dataset for 3d human activity analysis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1010–1019. IEEE, 2016.

Zhangzhang Si and Song-Chun Zhu. Learning and-or templates for object recognition and detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35 (9):2189–2205, 2013.

William D Smart and L Pack Kaelbling. Effective reinforcement learning for mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 4, pages 3404–3410. IEEE, 2002.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Jakob Suchan, Mehul Bhatt, and Paulo E Santos. Perceptual narratives of space and motion for activity interpretation. In *27th International Workshop on Qualitative Reasoning*, page 32, 2013.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Draft version, 2 edition, 2018.

Amir Tamrakar. Cwc blocks world (bw) apparatus, api reference manual. Technical report, 2015.

Marc Tanti, Albert Gatt, and Kenneth P Camilleri. Where to put the image in an image caption generator. *Natural Language Engineering*, 24(3):467–489, 2018.

Hamid R Tizhoosh. Reinforcement learning based on actions and opposite actions. In *International conference on artificial intelligence and machine learning*, volume 414, 2005.

Michael Tomasello and Ann Cale Kruger. Joint attention on actions: acquiring verbs in ostensive and non-ostensive contexts. *Journal of Child Language*, 19(2):311333, 1992. doi: 10.1017/S0305000900011430.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

Harini Veeraraghavan, Nikolaos Papanikolopoulos, and Paul Schrater. Learning dynamic event descriptions in image sequences. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–6. IEEE, 2007.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Sam Wiseman and Alexander M Rush. Sequence-to-sequence learning as beam-search optimization. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

Chenxia Wu, Jiemi Zhang, Silvio Savarese, and Ashutosh Saxena. Watch-n-patch: Unsupervised understanding of actions and relations. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4362–4370. IEEE, 2015.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine learning (ICML)*, pages 2048–2057, 2015.

Ying Yang and Geoffrey I Webb. Discretization for naive-bayes learning: managing discretization bias and variance. *Machine learning*, 74(1):39–74, 2009.

Jay Young and Nick Hawes. Learning by observation using qualitative spatial relations. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 745–751. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Xu-Yao Zhang, Fei Yin, Yan-Ming Zhang, Cheng-Lin Liu, and Yoshua Bengio. Drawing and recognizing chinese characters with recurrent neural network. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):849–862, 2018.