A User Simulator for Task-Completion Dialogues*

Xiujun Li[†]   **Zachary C. Lipton**[⋆]   **Bhuwan Dhingra**[‡]
Lihong Li[†]   **Jianfeng Gao**[†]   **Yun-Nung Chen**[§]
[†]Microsoft Research, Redmond, WA, USA
[⋆]University of California, San Diego, CA, USA
[‡]Carnegie Mellon University, Pittsburgh, PA, USA
[§]National Taiwan University, Taipei, Taiwan
[†]{xiul,lihongli,jfgao}@microsoft.com [⋆]zlipton@cs.ucsd.edu
[‡]bdhingra@andrew.cmu.edu [§]y.v.chen@ieee.org

Abstract

Despite widespread interests in reinforcement-learning for task-oriented dialogue systems, several obstacles can frustrate research and development progress. First, reinforcement learners typically require interaction with the environment, so conventional dialogue corpora cannot be used directly. Second, each task presents specific challenges, requiring separate corpus of task-specific annotated data. Third, collecting and annotating human-machine or human-human conversations for task-oriented dialogues requires extensive domain knowledge. Because building an appropriate dataset can be both financially costly and time-consuming, one popular approach is to build a user simulator based upon a corpus of example dialogues. Then, one can train reinforcement learning agents in an online fashion as they interact with the simulator. Dialogue agents trained on these simulators can serve as an effective starting point. Once agents master the simulator, they may be deployed in a real environment to interact with humans, and continue to be trained online. To ease empirical algorithmic comparisons in dialogues, this paper introduces a new, publicly available simulation framework, where our simulator, designed for the movie-booking domain, leverages both rules and collected data. The simulator supports two tasks: movie ticket booking and movie seeking. Finally, we demonstrate several agents and detail the procedure to add and test your own agent in the proposed framework.

1   Introduction

Practical dialogue systems consist of several components. The *natural language understanding* (NLU) module maps free texts to structured semantic frames of utterances. The *natural language generation* (NLG) module maps the structured representations back into a natural-language form. *Knowledge bases* (KBs) and *state trackers* provide access to side information and track the evolving state of the dialogue, respectively. The *dialogue policy* is a central component of the system that chooses an *action* given the current state of the dialogue.

In traditional systems, dialogue policies might be programmed explicitly with rules. However, rule-based approaches have several weaknesses. First, for complex systems, it may not be easy to design a reasonable rule-based policy. Second, the optimal policy might change over time, as user behavior changes. A rule-based system cannot cope with such non-stationarity. Thus, reinforcement learning, in which policies are learned automatically from experience, offers an appealing alternative.

---

*The source code is available at: `https://github.com/MiuLab/UserSimulator`

1.1 Why Is User Simulation Needed?

Typically, researchers seek to optimize dialogue policies with either supervised learning (SL) or reinforcement learning (RL) methods. In SL approaches, a policy is trained to imitate the observed actions of an expert. Supervised learning approaches often require a large amount of expert-labeled data for training. For task-specific domains, intensive domain knowledge is usually required for collecting and annotating actual human-human or human-machine conversations, and is often expensive and time-consuming. Additionally, even with a large amount of training data, it is possible that some dialogue state spaces may not be explored sufficiently in the training data, preventing a supervised learner to find a good policy.

In contrast, RL approaches allow an agent to learn without any expert-generated example. Given only a reward signal, the agent can optimize a dialogue policy through interaction with users. Unfortunately, RL can require many samples from an environment, making learning from scratch with real users impractical. To overcome this limitation, many researchers in the dialogue systems community train RL agents using simulated users [2, 4, 6, 11, 12, 15, 18].

The goal of user simulation is to generate natural and reasonable conversations, allowing the RL agent to explore the policy space. The simulation-based approach allows an agent to explore trajectories which may not exist in previously observed data, overcoming a central limitation of imitation-based approaches. Dialogue agents trained on these simulators can then serve as an effective starting point, after which they can be deployed against real humans to improve further via reinforcement learning.

1.2 Related Work

Given the reliance of the research community on user simulations, it seems important to assess the quality of the simulator. How best to assess a user simulator remains an open issue, and there is no universally accepted metric [13]. One important feature of a good user simulator requires coherent behavior throughout the dialogue; ideally, a good metric should measure the correlation between user simulation and real human behaviors, but it is hard to find a widely accepted metric. Therefore, to the best of our knowledge, there is no standard way to build a user simulator. Here, we summarize the literature of user simulation in different aspects:

- At the granularity level, the user simulator can operate either at the *dialog-act*[2] level, or at the *utterance* level [8].
- At the methodology level, the user simulator could use a *rule-based* approach, or a *model-based* approach where the model is learned from training data.

Many models have been introduced for user modeling in different dialogue systems. Early work [4, 9] employed a simple, naive bi-gram model $P(a_u|a_m)$ to predict the next user-act a_u based on the last system-act a_m. The parameters of this model are simple, but it cannot produce coherent user behaviors, for two reasons: (1) this model can only look at the last system action, and (2) if the user changes its goal, this bi-gram model might produce some illogical behavior since it does not consider the user goal when generating the next user-act. Much of the follow-up work on user simulators has tried to address these issues. The first issue can be addressed by looking at longer dialogue histories to select the next user action [5, 6]; the second issue can be attacked by explicitly incorporating the user goal into user state modeling [19].

The recently proposed sequence-to-sequence approach [21] has inspired end-to-end trainable user simulators [1]. This approach treats user-turn dialogue to agent-turn dialogue as a source-to-target sequence generation problem, which might be suitable for chatbot-like systems, but may not work well for domain-specific, task-completion dialogue systems, which require the ability to interact with databases and aggregate useful information into the system responses. The benefit of such model-based approaches is they do not need intensive feature engineering, but they typically require a large amount of labeled data to generalize well and deal with user states not included in the training data. On the other hand, agenda-based user simulation [16] provides a convenient mechanism to explicitly encode the dialogue history and user goal. The user goal consists of slot-value pairs describing the user's requests and constraints. A stack-like format models the state transitions and user action

---

[2]Here, a dialog-act consists of one intent, as well as zero, one or multiple slot-value pairs. In the rest of the paper, we will use dialog-acts and dialog actions interchangeably

generation as a sequence of simple push and pop operations, which ensures the consistency of user behavior over the course of conversation.

In this paper, we combine the benefits of both model-based and rule-based approaches. Our user simulation for the task-completion dialogue setting follows an agenda-based approach at the dialog-act level, and a sequence-to-sequence natural language generation (NLG) component is used to convert the selected dialog-act into natural language.

2   Dialogue Systems for Task-Completion

We consider a dialogue system for helping users to book movie tickets or to look up the movies they want, by interacting with them in natural language. Over the course of conversation, the agent gathers information about the customer's desires and ultimately books the movie tickets, or identify the movie of interest. The environment then assesses a binary outcome (success or failure) at the end of the conversation, based on (1) whether a movie is booked, and (2) whether the movie satisfies the user's constraints.

Data: The data we used in the paper was collected via Amazon Mechanical Turk, and the annotation was done internally using our own schema. There are 11 intents (i.e., inform, request, confirm_question, confirm_answer, etc.), and 29 slots (i.e., moviename, starttime, theater, numberofpeople, etc.). Most of the slots are *informable* slots, which users can use to constrain the search, and some are *requestable* slots, of which users can ask values from the agent. For example, *numberofpeople* cannot be a requestable slot, since arguably user knows how many tickets he or she wants to buy. In total, we labeled 280 dialogues in the movie domain, and the average number of turns per dialogue is approximately 11.

3   User Simulator

In this work, we follow the agenda-based user simulation approach [16], in which a stack-like representation of user state provides a convenient mechanism to explicitly encode the dialogue history and user's goal, and user state update (state transition and user action generation) can be modeled as sequences of push and pop operations with stacks. Here, we describe the rule-based user simulator in detail.

3.1   User Goal

In the task-oriented dialogue setting, the first step of user simulation is to generate a user goal; the agent knows nothing about the user goal but its objective is to help the user to accomplish this goal. Hence, the entire conversation exchange is around this goal implicitly. Generally, the definition of user goal contains two parts:

- *inform_slots* contain a number of slot value pairs which serve as constraints from the user.
- *request_slots* contain a set of slots that user has no information about the values, but wants to get the values from the agent side during the conversation.

To make the user goal more realistic, we add some constraints in the user goal: Slots are split into two groups. For movie-booking scenario, some of elements must appear in the user goal, we called these elements as *Required slots*, which includes *moviename, theater, starttime, date, numberofpeople*; the rest slots are *Optional slots*; *ticket* is a default slot which always appears in the *request_slots* part of user goal.

We generated the user goals from the labeled dataset, using two mechanisms. One mechanism is to extract all the slots (known and unknown) from the first user turns (excluding the greeting user turn) in the data, since usually the first turn contains some or all the required information from user. The other mechanism is to extract all the slots (known and unknown) that first appear in all the user turns, and then aggregate them into one user goal. We dump these user goals into a file as the user-goal database for the simulator. Every time when running a dialogue, we randomly sample one user goal from this user goal database.

3.2 User Action

First user-act: The work focuses on user-initiated dialogues, so we randomly generated a user goal as the first turn (a user turn). To make the user-act more reasonable, we add further constraints in the generation process. For example, the first user turn is usually a request turn; it has at least one informable slot; if the user knows the movie name, *moviename* will appear in the first user turn; etc.

During the course of a dialogue, the user simulator maintains a compact stack-like representation named as *user agenda* [16], where the user state s_u is factored into an agenda A and a goal G, which consists of constraints C and request R. At each time-step t, the user simulator will generate the next user action $a_{u,t}$ based on the its current status $s_{u,t}$ and the last agent action $a_{m,t-1}$, and then update the current status $s'_{u,t}$. Here, when training or testing a policy without natural language understanding (NLU), an error model [14] is introduced to simulate the noise from the NLU component, and noisy communication between the user and agent. There are two types of noise channels in the error model: one is at the intent level, the other is slot level. Furthermore, at the slot level, there are three kinds of possible noise:

- *slot deletion*: to simulate the scenario that the slot was not recognized by the NLU;

- *incorrect slot value*: to simulate the scenario that the slot name was recognized correctly, but the slot value was not recognized correctly, e.g., wrong word segmentation;

- *incorrect slot*: to simulate the scenario that both the slot and its value were not recognized correctly.

When training or testing a policy with natural language understanding (NLU), it is not necessary to use the error model because the NLU component itself introduces noise.

If the agent action is *inform(taskcomplete)*, this is to inform that the agent has gathered all the information and is ready to book the movie ticket. The user simulator will check whether the current stack is empty, and also conduct constraint checking to make sure that the agent is trying to book the right movie tickets. This guarantees that the user behaves in a consistent, goal-oriented manner.

3.3 Dialogue Status

There are three statuses for a dialogue: *no_outcome_yet*, *success* and *failure*.   The status is *no_outcome_yet* if the agent has not issued the *inform(taskcomplete)* action and if the number of turns of the conversation has not exceeded the maximum value; otherwise, the dialogue is finished with either a *success* or a *failure* outcome. To be a *success* dialogue, the agent must answer all the questions (a.k.a. requestable slots of the user) and book the right movie tickets finally, within the maximum number of turns. All other cases are *failure* dialogues. For example, the whole dialogue exceeds the limit of max turns, or the agent books the wrong movie tickets for the user.

There is a special case, where the user's constraints are not satisfiable in our movie database, and the agent correctly informs that no ticket can be booked. One can argue this is a successful outcome, as the agent does what is correct. Here, we choose to treat it as a failure, as no ticket is booked. It should be noted that this choice does not affect algorithm comparison much.

3.4 Natural Language Understanding (NLU)

The natural language understanding (NLU) component is a recurrent neural network model with long-short term memory (LSTM) cells. This single NLU model [7] can do intent prediction, and slot filling simultaneously. For joint modeling of intent and slots, the predicted tag set is a concatenated set of IOB-format slot tags and intent tags, and an additional token *<EOS>* is introduced at the end of each utterance, its supervised label is an intent tag, while the supervised label of all other preceding words is an IOB tag. In this way, we can still use the sequence-to-sequence training approach, the last hidden layer of the sequence is supposed to be a condensed semantic representation of the whole input utterance, so that it can be utilized for intent prediction at the utterance level. This model is trained using all available dialogue actions and utterance pairs in our labeled dataset.

3.5 Natural Language Generation (NLG)

The user simulator is designed on dialog act level, but it can also work on utterance level, we provide a natural language generation (NLG) component in the framework. Due to the limited labeled dataset, our empirical tests found that a pure model-based NLG might not generalize well, which will introduce a lot of noise for the policy training. Thus, we use a hybrid approach which consists of:

- *Template-based NLG*: outputs some predefined rule-based templates for dialog acts
- *Model-based NLG*: is trained on our labeled dataset in a sequence-to-sequence fashion. It takes dialog-acts as input, and generates template-like sentences with slot placeholders via an LSTM decoder. Then, a post-processing scan is performed to replace the slot placeholders with their actual values [23, 22]. In the LSTM decoder, we apply beam search, which iteratively considers the top k best sentences up to time step t when generating the token of the time step $t+1$. For the sake of the trade-off between the speed and performance, we use the beam size of 3 in our experiments.

In our hybrid model, if the dialog act can be found in the predefined rule-based templates, we use the template-based NLG for generating the utterance; otherwise, the utterance is generated by the model-based NLG.

4 Usages

We conduct experiments training agents with our user simulator for the following two tasks. The first is a task-completion dialogue setting on the movie-booking domain [10]. Here, the agent's job is to engage with the user in a dialogue with the ultimate goal of helping the user to successfully book a movie. To measure the quality of the agent, there are three metrics: {*success rate*[3], *average reward, average turns*}; each of them provides different information about the quality of agents. There exists a strong correlation among them: generally, a good policy should have a higher success rate, higher average reward and lower average turns. Here, we choose *success rate* as our major evaluation metric to report for the quality of agents. In the appendix, Table 1 demonstrates some example dialogues for this task.

The second task pertains to training an KB-InfoBot [3]. The setting is a simplified version of the previous goal-oriented dialogues, in which an agent and user communicate with only two intents (*request* and *inform*). Accordingly, for this task the experiments in KB-InfoBot [3] engage a simplified version of the simulator described in this paper, using the two aforementioned intents and six slots. In this paper, the knowledge-base is drawn from the IMDB dataset. In the appendix, Table 2 demonstrates some example dialogues for KB-InfoBot.

5 Discussion

In this paper, we demonstrated that rule-based user simulation can be a safe way to train reinforcement learning agents for task-completion dialogues. Since rule-based user simulation requires application-specific domain knowledge to curate these hand-crafted rules, it is usually a time-consuming process. One improvement for the current user simulation in the task-completion dialogue setting is to include user goal changes which make the dialogue more complex, but also realistic. Another potential direction for future improvement is model-based user simulation for task-completion dialogues. The advantage of model-based user simulation is that it can be adapted to other domains easily as long as there are enough labeled data. Since model-based user simulation is data-driven, one potential risk is that it asks for a large amount of labeled data to train a good simulator, and it might be risky to use the user simulator to train RL agents due to the uncertainty of the model. When training reinforcement learning agents with such a user simulator, the RL agents can easily learn these errors or loopholes existing in the model-based user simulator and make the false dialogues "success". In this case, the quality of learned RL policy can be misleadingly high. But model-based user simulator for task-completion dialogue setting is still a good direction to investigate.

---

[3]*Success rate* is sometimes known as *task completion rate* — the fraction of dialoges that finish successfully.

6 Acknowledgments

References

[1] Layla El Asri, Jing He, and Kaheer Suleman. A sequence-to-sequence model for user simulation in spoken dialogue systems. *arXiv:1607.00070*, 2016.

[2] Heriberto Cuayáhuitl, Steve Renals, Oliver Lemon, and Hiroshi Shimodaira. Human-computer dialogue simulation using hidden markov models. In *IEEE Workshop on Automatic Speech Recognition and Understanding*. IEEE, 2005.

[3] Bhuwan Dhingra, Lihong Li, Xiujun Li, Jianfeng Gao, Yun-Nung Chen, Faisal Ahmed, and Li Deng. End-to-end reinforcement learning of dialogue agents for information access. *arXiv:1609.00777*, 2016.

[4] Wieland Eckert, Esther Levin, and Roberto Pieraccini. User modeling for spoken dialogue system evaluation. In *Automatic Speech Recognition and Understanding, 1997. Proceedings., 1997 IEEE Workshop on*, pages 80–87. IEEE, 1997.

[5] Matthew Frampton and Oliver Lemon. Learning more effective dialogue strategies using limited dialogue move features. In *ACL*. Association for Computational Linguistics, 2006.

[6] Kallirroi Georgila, James Henderson, and Oliver Lemon. Learning user simulations for information state update dialogue systems. In *INTERSPEECH*, pages 893–896, 2005.

[7] Dilek Hakkani-Tür, Gokhan Tur, Asli Celikyilmaz, Yun-Nung Chen, Jianfeng Gao, Li Deng, and Ye-Yi Wang. Multi-domain joint semantic frame parsing using bi-directional rnn-lstm. In *Interspeech*, 2016.

[8] Sangkeun Jung, Cheongjae Lee, Kyungduk Kim, Minwoo Jeong, and Gary Geunbae Lee. Data-driven user simulation for automated evaluation of spoken dialog systems. *Computer Speech & Language*, 23(4):479–509, 2009.

[9] Esther Levin, Roberto Pieraccini, and Wieland Eckert. A stochastic model of human-machine interaction for learning dialog strategies. *IEEE Transactions on speech and audio processing*, 8(1):11–23, 2000.

[10] Zachary C Lipton, Jianfeng Gao, Lihong Li, Xiujun Li, Faisal Ahmed, and Li Deng. Efficient exploration for dialogue policy learning with BBQ networks & replay buffer spiking. *arXiv:1608.05081*, 2016.

[11] Olivier Pietquin. Consistent goal-directed user model for realisitc man-machine task-oriented spoken dialogue simulation. In *2006 IEEE International Conference on Multimedia and Expo*. IEEE, 2006.

[12] Olivier Pietquin and Thierry Dutoit. A probabilistic framework for dialog simulation and optimal strategy learning. *IEEE Transactions on Audio, Speech, and Language Processing*, 2006.

[13] Olivier Pietquin and Helen Hastie. A survey on metrics for the evaluation of user simulations. *The knowledge engineering review*, 2013.

[14] Jost Schatzmann, Blaise Thomson, and Steve Young. Error simulation for training statistical dialogue systems. In *IEEE Workshop on Automatic Speech Recognition & Understanding*, 2007.

[15] Jost Schatzmann, Karl Weilhammer, Matt Stuttle, and Steve Young. A survey of statistical user simulation techniques for reinforcement-learning of dialogue management strategies. *The knowledge engineering review*, 2006.

[16] Jost Schatzmann and Steve Young. The hidden agenda user simulation model. *IEEE transactions on audio, speech, and language processing*, 17(4):733–747, 2009.

[17] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv:1511.05952*, 2015.

[18] Konrad Scheffler and Steve Young. Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning. In *Proceedings of the second international conference on Human Language Technology Research*. Morgan Kaufmann Publishers Inc., 2002.

[19] Konrad Haarhoff Scheffler. *Automatic design of spoken dialogue systems*. PhD thesis, University of Cambridge, 2003.

[20] Pei-Hao Su, Milica Gasic, Nikola Mrksic, Lina Rojas-Barahona, Stefan Ultes, David Vandyke, Tsung-Hsien Wen, and Steve Young. Continuously learning neural dialogue management. *arXiv:1606.02689*, 2016.

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

[22] Tsung-Hsien Wen, Milica Gašić, Nikola Mrkšić, Lina M. Rojas-Barahona, Pei-Hao Su, Stefan Ultes, David Vandyke, and Steve Young. Conditional generation and snapshot learning in neural dialogue systems. *EMNLP*, 2016.

[23] Tsung-Hsien Wen, Milica Gašić, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. *EMNLP*, 2015.

[24] Jason D Williams and Geoffrey Zweig. End-to-end lstm-based dialog control optimized with supervised and reinforcement learning. *arXiv:1606.01269*, 2016.

[25] Tiancheng Zhao and Maxine Eskenazi. Towards end-to-end learning for dialog state tracking and management using deep reinforcement learning. *arXiv:1606.02560*, 2016.

A   Recipes

This framework provides you a way to develop and compare different algorithms/models (i.e., agents in the dialogue setting). The dialogue system consists of two parts: agent and user simulator. Here, we walk through some examples to show how to build and plug in your own agents and user simulators.

A.1   How to build your own agent?

For all the agents, they are inherited from the Agent class (agent.py) which provides some common interfaces for users to implement their agents. In the agent_baseline.py file, five basic rule-based agents are implemented:

- *InformAgent* informs all the slots one by one in every turn; it cannot request any information/slot.
- *RequestAllAgent* requests all the slots one by one in every turn; it cannot inform any information/slot.
- *RandomAgent* requests any random request in every turn; it cannot inform any information/slot.
- *EchoAgent* informs the slot in the request slots of last user action; it cannot request any information/slot.
- *RequestBasicsAgent* requests all basic slots in a subset one by one, then chooses inform(taskcomplete) at the last turn; it cannot inform any information/slot.

All the agents just re-implement two functions: initialize_episode and state_to_action. Here state_to_action function makes no assumption about the structure of the agent, it is an interface to implement the mapping from state to action, which is the core part in the agent. Here is an example of *RequestBasicsAgent*:

```
1   class RequestBasicsAgent(Agent):
2       """ A simple agent to test the system. This agent should simply request all the basic slots and then
         issue: thanks(). """
3
4       def initialize_episode(self):
5           self.state = {}
6           self.state['diaact'] = 'UNK'
7           self.state['inform_slots'] = {}
8           self.state['request_slots'] = {}
9           self.state['turn'] = -1
10          self.current_slot_id = 0
11          self.request_set = ['moviename', 'starttime', 'city', 'date', 'theater', 'numberofpeople']
12          self.phase = 0
13
14      def state_to_action(self, state):
15          """ Run current policy on state and produce an action """
16
17          self.state['turn'] += 2
18          if self.current_slot_id < len(self.request_set):
19              slot = self.request_set[self.current_slot_id]
20              self.current_slot_id += 1
21
22              act_slot_response = {}
23              act_slot_response['diaact'] = "request"
24              act_slot_response['inform_slots'] = {}
25              act_slot_response['request_slots'] = {slot: "UNK"}
26              act_slot_response['turn'] = self.state['turn']
27          elif self.phase == 0:
28              act_slot_response = {'diaact': "inform", 'inform_slots': {'taskcomplete': "PLACEHOLDER"}, '
           request_slots': {}, 'turn':self.state['turn']}
29              self.phase += 1
30          elif self.phase == 1:
31              act_slot_response = {'diaact': "thanks", 'inform_slots': {}, 'request_slots': {}, 'turn':
           self.state['turn']}
32          else:
33              raise Exception("THIS SHOULD NOT BE POSSIBLE (AGENT CALLED IN UNANTICIPATED WAY)")
34          return {'act_slot_response': act_slot_response, 'act_slot_value_response': None}
```

Listing 1: RequestBasicsAgent

All the above rule-based agents can support only either inform or request action, here you can practice to implement a sophisticated rule-based agent which can support multiple actions, including inform, request, confirm_question, confirm_answer, deny etc.

agent_dqn.py provides a RL agent (agt=9), which wraps a DQN model. Besides the two above functions, there are two major functions in the RL agent: run_policy and train. run_policy implements an ϵ-greedy policy, and train calls the batch training function of DQN.

```
1   class AgentDQN(Agent):
2       def run_policy(self, representation):
3           """ epsilon-greedy policy """
4
5           if random.random() < self.epsilon:
6               return random.randint(0, self.num_actions - 1)
7           else:
8               if self.warm_start == 1:
9                   if len(self.experience_replay_pool) > self.experience_replay_pool_size:
10                      self.warm_start = 2
11                  return self.rule_policy()
12              else:
13                  return self.dqn.predict(representation, {}, predict_model=True)
14
15      def train(self, batch_size=1, num_batches=100):
16          """ Train DQN with experience replay """
17
18          for iter_batch in range(num_batches):
19              self.cur_bellman_err = 0
20              for iter in range(len(self.experience_replay_pool)/(batch_size)):
21                  batch = [random.choice(self.experience_replay_pool) for i in xrange(batch_size)]
22                  batch_struct = self.dqn.singleBatch(batch, {'gamma': self.gamma}, self.clone_dqn)
```

Listing 2: Two major functions for RL agent

agent_cmd.py provides a command line agent (agt=0), which you as an agent can interact with the user simulator. The command line agent supports two types of input: natural language (cmd_input_mode=0) and dialog act(cmd_input_mode=1). Listing 3 shows an example of command

line agent interacting with the user simulator via the natural language; Listing 4 shows an example of command line agent interacting with the user simulator via dialog act form. Note:

- When the last user turn is a *request* action, the system will show a line of suggested available answers in the database for the agent, like the turn 0 in the Listing 4. Both rule-based agents and RL agent, they will answer the user with the slot values from the database. Here a special case for command line agent is, human (as command line agent) might type any random answer to user's request, when the typed answer is not in the database, the state tracker will correct it, and force the agent to use the values from the database in the agent response. For example, in turn 1 of the Listing 4, if you input *inform(theater=amc pacific)*, the actual answer received by the user is *inform(theater=carmike summit 16)*, because *amc pacific* doesn't exist in the database, to avoid this wired behavior that agent informs the user a unavailable value, we restrict the agent to use the values from the suggested list.

- The last second turn of agent is usually an *inform(taskcomplete)* in dialog act form or something like "*Okay, your tickets are booked.*" in natural language, which is to inform the user simulator that the agent nearly completes the task, and is ready to book the movie tickets.

- To end a conversation, the last turn of the agent is usually a *thanks()* in dialog act form or "*thanks*" in natural language.

```
1  python run.py --agt 0 --usr 1 --max_turn 40 --episodes 150 --movie_kb_path .\deep_dialog\data\movie_kb.1k
       .json --goal_file_path .\deep_dialog\data\user_goals_first_turn_template.part.movie.v1.p --
       intent_err_prob 0.00 --slot_err_prob 0.00 --episodes 500 --act_level 0 --run_mode 0 --
       cmd_input_mode 0
2
3  New episode, user goal:
4  {
5    "request_slots": {
6      "ticket": "UNK"
7    },
8    "diaact": "request",
9    "inform_slots": {
10     "city": "seattle",
11     "numberofpeople": "2",
12     "theater": "amc pacific place 11 theater",
13     "starttime": "9:00 pm",
14     "date": "tomorrow",
15     "moviename": "deadpool"
16   }
17 }
18 Turn 0 usr: Can I buy tickets for deadpool at seattle?
19 Turn 1 sys: Which city do you want to buy the ticket?
20 Turn 2 usr: I want to watch at seattle.
21 Turn 3 sys: Which theater do you want?
22 Turn 4 usr: I want to watch at amc pacific place 11 theater.
23 Turn 5 sys: What date would you like?
24 Turn 6 usr: I want to set it up tomorrow
25 Turn 7 sys: And what start time do you like?
26 Turn 8 usr: I want to watch at 9:00 pm.
27 Turn 9 sys: How many tickets do you need?
28 Turn 10 usr: I want 2 tickets please!
29 Turn 11 sys: Okay, your tickets were booked.
30 Turn 12 usr: Thank you
31 Turn 13 sys: thanks
32 Successful Dialog!
```

Listing 3: An example of command line agent interacting with user simulator with natural language

```
1  python run.py --agt 0 --usr 1 --max_turn 40 --episodes 150 --movie_kb_path .\deep_dialog\data\movie_kb.1k
       .json --goal_file_path .\deep_dialog\data\user_goals_first_turn_template.part.movie.v1.p --
       intent_err_prob 0.00 --slot_err_prob 0.00 --episodes 500 --act_level 0 --run_mode 0 --
       cmd_input_mode 1
2
3  New episode, user goal:
4  {
5    "request_slots": {
6      "ticket": "UNK",
7      "theater": "UNK"
8    },
9    "diaact": "request",
10   "inform_slots": {
11     "city": "birmingham",
12     "numberofpeople": "2",
```

9

```
13      "state": "al",
14      "starttime": "4 pm",
15      "date": "today",
16      "moviename": "deadpool"
17   }
18 }
19 Turn 0 usr: Which theater will play the deadpool at 4 pm?
20 (Suggested Values: {'theater': ['carmike summit 16']})
21 Turn 1 sys: inform(theater=carmike summit 16)
22 Turn 2 usr: I need tickets at al.
23 Turn 3 sys: request(numberofpeople)
24 Turn 4 usr: I want 2 tickets please!
25 Turn 5 sys: request(city)
26 Turn 6 usr: I want to watch at birmingham.
27 Turn 7 sys: request(starttime)
28 Turn 8 usr: I want to watch at 4 pm.
29 Turn 9 sys: request(date)
30 Turn 10 usr: I want to set it up today
31 Turn 11 sys: inform(taskcomplete)
32 Turn 12 usr: Thank you
33 Turn 13 sys: thanks()
34 Successful Dialog!
```

Listing 4: An example of command line agent interacting with user simulator with Dialog Act

A.2 How to build your own user simulator?

Similarly, there is one user simulator class (usersim.py) which provides a few common interfaces for users to implement their user simulators. All the user simulators are inherited from this class, they should re-implement these two functions: initialize_episode and next. The usersim_rule.py file implements a rule-based user simulator. Here the next function implements all the rules and mechanism to issue the next user action based on the last agent action. Here is the example of usersim_rule.py:

```
1 def next(self, system_action):
2         """ Generate next User Action based on last System Action """
3
4         self.state['turn'] += 2
5         self.episode_over = False
6         self.dialog_status = dialog_config.NO_OUTCOME_YET
7
8         sys_act = system_action['diaact']
9
10        if (self.max_turn > 0 and self.state['turn'] > self.max_turn):
11            self.dialog_status = dialog_config.FAILED_DIALOG
12            self.episode_over = True
13            self.state['diaact'] = "closing"
14        else:
15            self.state['history_slots'].update(self.state['inform_slots'])
16            self.state['inform_slots'].clear()
17
18            if sys_act == "inform":
19                self.response_inform(system_action)
20            elif sys_act == "multiple_choice":
21                self.response_multiple_choice(system_action)
22            elif sys_act == "request":
23                self.response_request(system_action)
24            elif sys_act == "thanks":
25                self.response_thanks(system_action)
26            elif sys_act == "confirm_answer":
27                self.response_confirm_answer(system_action)
28            elif sys_act == "closing":
29                self.episode_over = True
30                self.state['diaact'] = "thanks"
31
32        self.corrupt(self.state)
33
34        response_action = {}
35        response_action['diaact'] = self.state['diaact']
36        response_action['inform_slots'] = self.state['inform_slots']
37        response_action['request_slots'] = self.state['request_slots']
38        response_action['turn'] = self.state['turn']
39        response_action['nl'] = ""
40
41        # add NL to dia_act
42        self.add_nl_to_action(response_action)
```

```
43          return response_action, self.episode_over, self.dialog_status
```
<div align="center">Listing 5: User Simulator Rules</div>

B Training Details

To train a RL agent, you can either start with some rule policy experience tuples to initialize the experience replay buffer pool or start with an empty experience replay buffer pool. We recommend to use some rule or supervised policy to initialize the experience replay buffer pool, many work [24, 20, 25, 10] have demonstrated the benefits of such strategy as a good initialization to speed up the RL training. Here, we use a very simple rule-based policy to initialize the experience replay buffer pool.

The RL agent is a DQN network. In the training, we use the ϵ-greedy policy and a dynamic experience replay buffer pool. The size of experience replay buffer pool is dynamic changing. One important trick of DQN is to introduce the target network, which is updated slowly and used to compute the target value in a short period.

The training procedure goes like this way: at each simulation epoch, we simulate N dialogues and add these state transition tuples (s_t, a_t, r_t, s_{t+1}) into experience replay buffer pool, train and update the current DQN network. In one simulation epoch, the current DQN network will be updated multiple times, depending on the batch size and the current size of experience replay buffer, at the end of simulation epoch, the target network will be replaced by the current DQN network, the target DQN network is only updated for once in one simulation epoch. The experience replay strategy is critic for the training [17]. Our experience reply buffer update strategy is as follows: at the beginning, we will accumulate all the experience tuples from the simulation and flush the experience reply buffer pool till the current RL agent reaches a success rate threshold (i.e. *success_rate_threshold* = 0.30), then use the experience tuples from the current RL agent to re-fill the buffer. The intuition behind is the initial performance of the DQN is not good to generate enough good experience replay tuples, thus we do not flush the experience replay pool till the current RL agent can reach a certain success rate which we think it is good, for example, the performance of a rule-based agent. Then in the following training process, at every simulation epoch, we estimate the success rate of the current DQN agent, if the current DQN agent is better enough (i.e. better than the target network), the experience replay buffer poll will be flushed and re-filled. Figure 1 shows a learning curve for RL agent without NLU and NLG, Figure 2 is a learning curve for RL agent with NLU and NLG, it takes longer time to train the RL agent to adapt the errors and noise from NLU and NLG.

C Sample Dialogues

C.1 Task-Completion Bot

Table 1 shows one success and one failure dialogue examples generated by the rule-based agent and RL agent interacting with user simulator in the movie-booking domain. To be informative, we also explicitly show the user goal at the head of the dialogue, but the agent knows nothing about the user goal, its goal is to help the user to accomplish this goal and book the right movie tickets.

C.2 KB-InfoBot

Table 2 shows some sample dialogues between the user simulator and SimpleRL-SoftKB and End2End-RL agents [3]. Value of the *critic_rating* slot is a common source of error in the user simulator, and hence all learned policies tend to ask for this value multiple times.
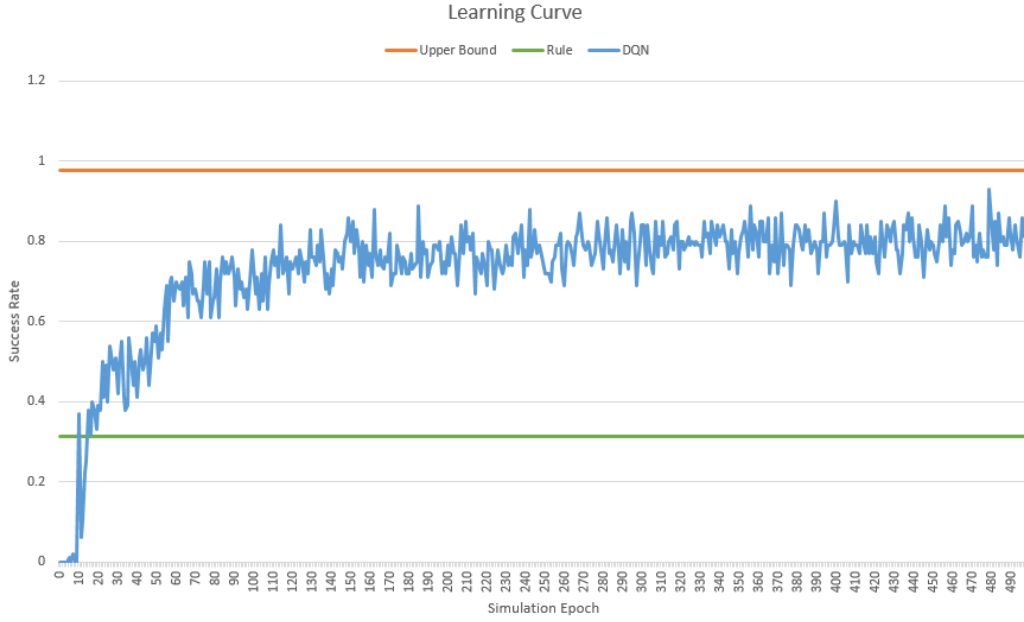
Figure 1: Learning curve for policy training, without NLU and NLG: Green line is a rule agent which we employ to initialize the experience replay buffer pool; the blue line is the learning curve for the RL agent; orange line is the optimal upper bound, which is computed by the ratio of the number of reachable user goals in the database of the agent to the total number of user goals.
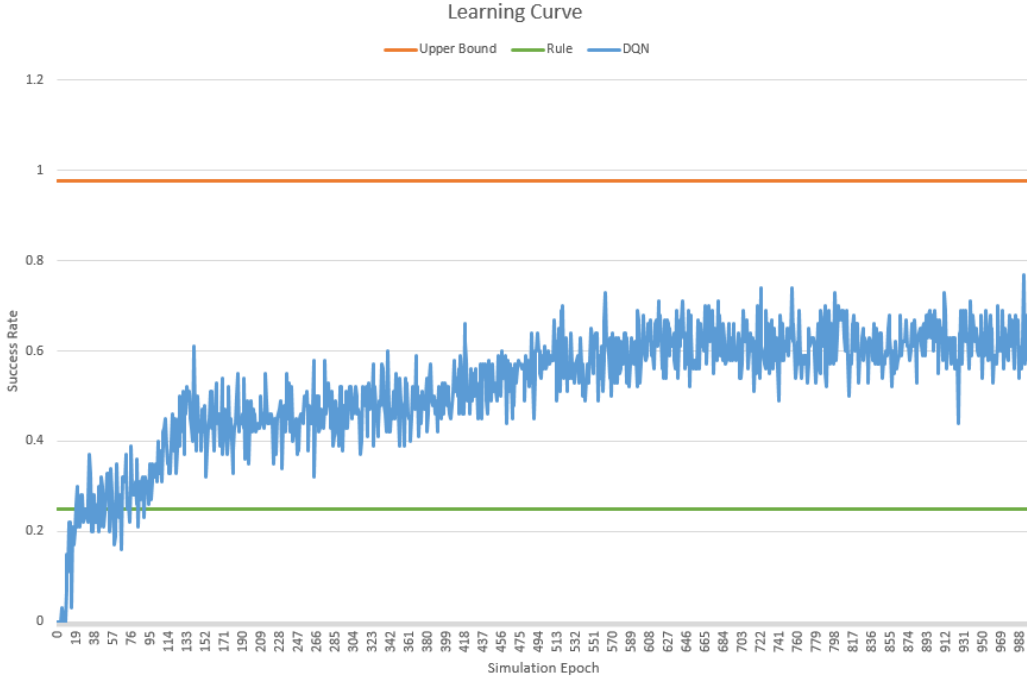

Figure 2: Learning curve for the end-to-end policy training, with NLU and NLG: Green line is a rule agent which we employ to initialize the experience replay buffer pool; the blue line is the learning curve for the RL agent; orange line is the optimal upper bound, which is computed by the ratio of the number of reachable user goals in the database of the agent to the total number of user goals.

Table 1: Two sample dialogues generated by rule-based agent and RL agent with user simulator: Left column shows both rule and RL agents succeed; Right column shows that rule-based agent fails, while RL agent succeeds.

New episode, user goal:
{
    "request_slots": {
        "ticket": "UNK"
    },
    "diaact": "request",
    "inform_slots": {
        "city": "seattle",
        "numberofpeople": "2",
        "theater": "regal meridian 16",
        "starttime": "9:25 pm",
        "date": "tomorrow",
        "moviename": "zoolander 2"
    }
}
	New episode, user goal:
	{
	    "request_slots": {
	        "ticket": "UNK"
	        "theater": "UNK"
	        "starttime": "UNK"
	    },
	    "diaact": "request",
	    "inform_slots": {
	        "numberofpeople": "3",
	        "date": "tomorrow",
	        "moviename": "10 cloverfield lane"
	    }
	}

Rule-based agent	**Rule-based Agent**
User: Can I get some tickets for zoolander 2? Agent: What movie are you interested in? User: I want to watch zoolander 2. Agent: What time would you like to see it? User: I want to watch at 9:25 pm. Agent: What city you would like? User: I want to watch at seattle. Agent: What date would you like to watch it? User: I want to set it up tomorrow. Agent: Which theater would you like? User: I want to watch at regal meridian 16. Agent: How many tickets do you need? User: I want 2 tickets please! Agent: Great - I was able to purchase 2 tickets for you to see zoolander 2 tomorrow at regal meridian 16 theater in seattle at 9:25 pm. User: Thank you. Agent: Thank you.	User: What is the start time for 10 cloverfield lane? Agent: What movie are you interested in? User: I want to watch at 10 cloverfield lane. Agent:What time would you like to see it? User: hat time is it playing tomorrow? I need 3 tickets. What city you would like? User:I do not care. What date would you like to watch it? User: I want to set it up tomorrow. Agent: Which theater would you like? User: Which theater is available? Agent: How many tickets do you need? User: I want 3 tickets please! Agent: Okay, I have booked 3 tickets for 10 cloverfield lane tomorrow. User: Thank you. Agent: Thank you.
Success	**Failure:** Agent does not answer all the questions (starttime, theater) from user.
RL agent	**RL Agent**
User: Can I get tickets for zoolander 2 tomorrow? Agent: Which city would you like? User: I want to watch at seattle. Agent: How many tickets do you need? User: I want 2 tickets please! Agent: 9:25 pm is available. User: I want to watch at regal meridian 16. Agent: Great - I was able to purchase 2 tickets for you to see zoolander 2 tomorrow at regal meridian 16 theater in seattle at 9:25 pm. User: Thank you. Agent: Thank you.	User: Which theater can I book 3 tickets for 10 cloverfield lane? Agent: What time would you like to see it? User: Which theater and start time are available tomorrow? Agent: 11:45am is available. User: Which theater is available? Agent: regal la live stadium 14 is available. User: Could you help me to book the tickets? Agent: Okay - I was able to book 3 tickets for you to see 10 cloverfield lane at regal la live stadium 14 at 11:45am tomorrow. User:Thank you Agent: Thank you
Success	**Success**

Table 2: Sample dialogues between user simulator and SimpleRL-SoftKB and End2End-RL agents. At the end of each dialogue, the agent informs top 5 results from the KB posterior. User target, if informed, is in bold.

SimpleRL-SoftKB	End-to-End
User: i would like to know the moviename whose genre is drama and critic_rating is 6.3. Agent: *request*(mpaa_rating)	User: i would like to know the moviename whose genre is drama and critic_rating is 6.3. Agent: *request*(actor)
User: i cannot remember Agent: *request*(actor)	User: i cannot remember Agent: *request*(director)
User: i cannot remember Agent: *request*(critic_rating) User: i think it is 6.3	User: milan cheylov director Agent: *inform*(**the countess**, hemingway & gellhorn, adios derek, unstoppable, wagers of sin)
Agent: *request*(critic_rating)	
User: i think it is 6.3 Agent: *inform*(kid svensk, hemigway & gellhorn, ry. ga gotuku, night catches us, tmnt)	
User: i am looking for a moviename with mpaa_rating 6 and director . Agent: *request*(actor)	User: i am looking for a moviename with mpaa_rating 6 and critic_rating 8.7. Agent: *request*(actor)
User: i cannot remember Agent: *request*(critic_rating)	User: i cannot remember Agent: *request*(director)
User: i think it is 7.5 Agent: *request*(genre)	User: i cannot remember Agent: *request*(critic_rating)
User: i cannot remember Agent: *request*(critic_rating)	User: i think it is 7.5 Agent: *request*(critic_rating)
User: i think it is 7.5 Agent: *request*(critic_rating)	User: i think it is 7.5 Agent: *request*(genre)
User: i think it is 4.0 Agent: *inform*(l'ennemi public, open season, **october road**, fleeting cheating meeting, final draft)	User: i cannot remember Agent: *inform*(fleeting cheating meeting, **october road**, open season, l'ennemi public, journeyman)
User: which moviename with director pg director khan and mpaa_rating . Agent: *request*(actor)	User: i am looking for a moviename with director khan critic_rating 2008 and mpaa_rating pg. Agent: *request*(actor)
User: i cannot remember Agent: inform(heber holiday, husbands and knives, not quite hollywood: the wild, the man who lost his head, **meet dave**)	User: i cannot remember Agent: *request*(critic_rating)
	User: i cannot remember Agent: *request*(genre)
	User: i cannot remember Agent: *inform*(heber holiday, not quite hollywood: the wild, **meet dave**, agent crush, mrs mcginty's dead)
User: i am looking for a moviename with actor dylan release_year 2008 and genre horror. Agent: *inform*(**parasomnia**, encarna..o do dem.nio, sense & sensibility, dead and gone, the broken)	User: i am looking for a moviename with actor dylan release_year 2008 and genre horror. Agent: *inform*(**parasomnia**, encarna..o do dem.nio, the broken, sense & sensibility, american teen)
User: which moviename has natasha as actor? Agent: *request*(mpaa_rating)	User: which moviename has natasha as actor? Agent: *request*(critic_rating)
User: i cannot remember Agent: *request*(genre)	User: i think it is 9.2. Agent: *request*(critic_rating)
User: music Agent: *request*(critic_rating)	User: i think it is 8.9. Agent: *request*(director)
User: 9.3 Agent: *inform*(the belgrade phantom, nine inch nails live, zhan. gu, taare zameen par, elvis lives)	User: i cannot remember Agent: *request*(mpaa_rating)
	User: i cannot remember Agent: *request*(actor)
	User: the actor is natasha alderslade. Agent: *inform*(**mugged**, the rage, wayne's world, arritmia, the broken)