PyOpenDial: A Python-based Domain-Independent Toolkit for Developing Spoken Dialogue Systems with Probabilistic Rules

Youngsoo Jang[1*], **Jongmin Lee**[1*], **Jaeyoung Park**[1*],
Kyeng-Hun Lee[3], **Pierre Lison**[4], **Kee-Eung Kim**[1,2]
[1] School of Computing, KAIST, Daejeon, Republic of Korea
[2] Graduate School of AI, KAIST, Daejeon, Republic of Korea
[3] Samsung Electronics, Seoul, Republic of Korea
[4] Norwegian Computing Center, Oslo, Norway
{ysjang,jmlee,jypark}@ai.kaist.ac.kr,
kyenghun.lee@samsung.com, plison@nr.no, kekim@cs.kaist.ac.kr

Abstract

We present PyOpenDial, a Python-based domain-independent, open-source toolkit for spoken dialogue systems. Recent advances in core components of dialogue systems, such as speech recognition, language understanding, dialogue management, and language generation, harness deep learning to achieve state-of-the-art performance. The original OpenDial, implemented in Java, provides a plugin architecture to integrate external modules, but lacks Python bindings, making it difficult to interface with popular deep learning frameworks such as Tensorflow or PyTorch. To this end, we re-implemented OpenDial in Python and extended the toolkit with a number of novel functionalities for neural dialogue state tracking and action planning. We describe the overall architecture and its extensions, and illustrate their use on an example where the system response model is implemented with a recurrent neural network.

1 Introduction

Spoken dialogue systems (SDSs) allow interactions between users and machines through natural language conversations. These systems are composed of a broad range of components such as speech recognition, language understanding, dialogue management, language generation, and speech synthesis. Recent SDS frameworks, such as AT&T Statistical Dialogue Toolkit (Williams et al., 2010), OpenDial (Lison and Kennington, 2016), and PyDial (Ultes et al., 2017) aim to integrate these complex and diverse components through a modular architecture.

Spoken dialogue systems may either adopt symbolic or statistical approaches to perform language understanding, dialogue management and language generation. Statistical approaches, including but not limited to (Young et al., 2013; Ultes et al., 2017) rely on probabilistic models of dialogue interactions and allows these models to be estimated from data. Symbolic approaches, on the other hand, model the dialogue interaction using finite-state automata or logical methods designed by the developer. Of our particular interest is OpenDial, a Java-based open-source toolkit that combines the benefits of both statistical and symbolic approaches.

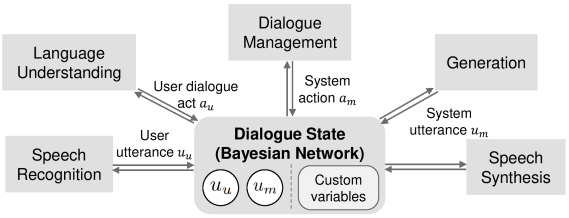

Figure 1: Information-state architecture for PyOpenDial.

In recent years, deep learning has shown to achieve promising performance results in many tasks related to dialogues, such as speech recognition (Graves et al., 2013), language understanding (Radford et al., 2018), dialogue management (Williams et al., 2017) and language generation (Yu et al., 2016). Given that one of the most popular programming languages for deep learning libraries is Python, e.g. Tensorflow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017), the integration of neural conversational models would be a straightforward task if OpenDial itself was written in Python.

To this end, we developed PyOpenDial, an open-source SDS framework that re-implements OpenDial in Python and integrates a range of novel functionalities, such as the possibility to

*:These authors contributed equally.

This work was supported by MOTIE (KEIT No. 10063424), MSIT (IITP No. 2017-0-01779 XAI, IITP No. 2019-0-00075-001), and Samsung Research.

track neural dialogue state and the use of Monte Carlo Tree Search for forward planning. PyOpenDial inherits the original architectural design of OpenDial, and extends the XML domain specification so that various deep learning models can be directly used for SDS components. We include the negotiation dialogue domain in (Lewis et al., 2017) as an example to show how to integrate with external components trained by deep learning.

2 Architecture

2.1 Dialogue State

PyOpenDial inherits the information-state framework in OpenDial (Larsson and Traum, 2000): All the components in the toolkit operates on the shared information state that represents the dialogue state arising from the interaction between the user and the machine (see Figure 1). The dialogue state is represented by a Bayesian network (BN), a directed graphical model for encoding the probability distribution of the dialogue state. Hence, the dialogue state consists of a factored representation of state variables on a dialogue, which are random variables, and conditional dependencies between them.

2.2 Workflow

PyOpenDial adopts the probabilistic rules used in OpenDial (Lison, 2014; Lison and Kennington, 2016) to update the dialogue state represented by a Bayesian Network during the conversation. These rules follow a *if...then...else* skeleton that map logical conditions on a subset of state variables to a probability or utility distribution on another subset of state variables. Two types of rules are provided: probability rules and utility rules. The probability rule defines the probabilistic change of state variables through a probability distribution over *effects*, each of which is an assignment on the state variables, given a logical condition of state variables. The utility rule defines utilities on the values of action variables given a logical conditions of state variables. These probabilistic rules are specified in the domain XML file. Examples are shown in Listing 1, where line 13-23 contains the probability rule and line 27-38 contains the utility rule.

These probabilistic rules can be grouped according to subtasks, such as language understanding, dialogue management, language generation and etc. Each group is defined as a *model*. Each

```
1 <initialstate>
2  <variable id="movie_rnn">
3   <value>@chatbot.MovieRNN</value>
4  </variable>
5  <variable id="music_rnn">
6   <value>@chatbot.MusicRNN</value>
7  </variable>
8 </initialstate>
9
10 <function name="gen_u_m">chatbot.generate</function>
11 <!-- User intent recognition -->
12 <model trigger="u_u">
13  <rule>
14   <case>
15    <condition>
16     <if var="u_u" value="movie" relation="contains"/>
17    </condition>
18    <effect prob="1">
19     <set var="a_u" value="movie"/>
20    </effect>
21   </case>
22       ...
23  </rule>
24 </model>
25 <!-- System utterance generation -->
26 <model trigger="a_u">
27  <rule>
28   <case>
29    <condition>
30     <if var="a_u" value="movie"/>
31    </condition>
32    <effect util="1">
33     <set var="u_m"
34      value="@gen_u_m({movie_rnn},{u_u})"/>
35    </effect>
36   </case>
37       ...
38  </rule>
39 </model>
```

Listing 1: A simple example domain XML specification for an RNN-based chat-bot. Here, `u_u` stands for the user utterance, `a_u` for the user intent and `u_m` for the system utterance.

model is associated with a subset of state variables, called *trigger variables*. Each model monitors the change of its trigger variables. When one or more trigger variables are updated during a conversation, the probabilistic rules on the corresponding model are applied to the dialogue state by instantiating the rule with the current dialogue state. Note that these updates may result to trigger other models, hence the procedure causes a chain of updates on the dialogue state through the probabilistic rules of the models. In summary, the workflow of PyOpenDial is basically a series of applications of probabilistic rules to the dialogue state.

Since this workflow is basically the same as in OpenDial, we refer the readers to Lison (2014), Lison and Kennington (2016), and the OpenDial toolkit website (`http://www.opendial-toolkit.net`) for more details on the XML specification of the domain modeling.

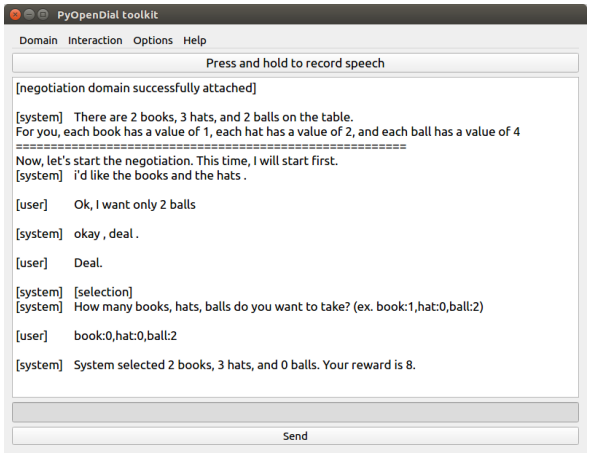

Figure 2: User interface for PyOpenDial.

2.3 Extensions to OpenDial

Custom Variable Types Neural models such as recurrent neural networks (RNNs) have become a popular choice for various dialogue processing tasks, given their capability to be trained end-to-end and infer complex latent representations of the dialogue state. In these models, the dialogue state is typically represented as a vector-valued prediction computed from a complex mapping from input to output. In contrast, the original OpenDial only supports updates on primitive data types (e.g. boolean, double, string, double array, and set) via human-readable probabilistic rules similar to decision trees. In order to overcome this limitation, we extend the specification of the dialogue state to include complex variable types and functional values to allow arbitrarily complex mappings of conditional variables, e.g. latent vectors in neural models that encode the dialogue context.

This is achieved by extending the domain XML specification to allow for variable types expressed through complex functions that can be integrated in probabilistic rules. In Listing 1, two such custom variables are defined: `movie_rnn` and `music_rnn`, which are instances of `MovieRNN` and `MusicRNN` respectively (line 2-7), representing pre-trained RNN-based generation models. Line 10 assigns `gen_u_m` to function `generate` in module `chatbot`, which executes the neural model. This particular function takes two arguments (namely the generation model and a user utterance) as input and returns the system utterance as output.

Predictive models Dialogue management is, at its core, a sequential decision-making problem, where the goal of the system is to select actions that fulfill the system objectives while minimising associated costs. One way to achieve this objective is through forward planning, i.e. enabling the system to search for actions that yield the maximum expected utility over a given horizon. Forward planning requires the specification of predictive models (such as user simulation models) to be able to predict the consequences of the system actions on the current interaction. PyOpenDial provides the `planning-only` option to models, which makes the model triggered only when forward planning is performed. This allows the system to explicitly differentiate between observed and predicted values in the dialogue state. The specific use-case of this feature is described in Section 4 with Listing 2 (line 30).

3 Implementation

PyOpenDial is implemented in Python and is released under the MIT opensource license. The toolkit is available through the GitHub code repository at (`https://github.com/KAIST-AILab/PyOpenDial`) [1]. The toolkit additionally provides a graphical user interface that helps fast-prototyping and test-driving the system. The graphical user interface, shown in Figure 2, displays the domain and dialogue history and take the user's next input (text or speech).

PyOpenDial implements all the core components and modules of OpenDial, including the BN inference algorithm and the probabilistic rule engine. We also introduced a number of new modules, including the Monte-Carlo tree search (MCTS) (Kocsis and Szepesvári, 2006) planner and the basic speech-to-text and text-to-speech modules using Google Speech APIs.

MCTS Planner In dialogue management, planning algorithms are often used to search for the system action that maximizes the sum of utilities in the long-term horizon so as to optimally react to the user.

The baseline planning algorithm in OpenDial is a lookahead forward planner that fully expands the search tree up to the planning horizon H:

$$Q_t(b,a) = U(b,a) + \gamma \max_{a'} \mathbb{E}_{o|b,a}\left[Q_{t+1}(b^{ao},a')\right]$$

where b is the dialogue state, a is the value of the action variables, o is the possible observation

[1] More dialogue domain examples combined with deep neural networks can be found in the GitHub repository.

when taking action a in the state b, b^{ao} is the dialogue state updated from b after action a and observation o, $\gamma \in [0, 1)$ is the discount factor, $U(b, a)$ is the instantaneous utility of a at the dialogue state b, and $Q_H(b, a) = 0$ for all b, a. After computing $Q_0(b, a)$ from the recursive equation, the forward planner chooses the final value of the action variable given by $\text{argmax}_a Q_0(b, a)$. A major limitation of the forward planner is that the search becomes infeasible in the planning horizon as well as the branching factor (i.e. the number of candidate actions and observations).

On the other hand, MCTS combines tree search with Monte-Carlo simulation so that the search effort is non-uniformly invested into promising nodes. One of the most basic MCTS algorithms is UCT (Kocsis and Szepesvári, 2006), which performs iterative simulation on the search tree by following the UCB rule to select actions at intermediate nodes

$$\text{argmax}_a \left[Q(b, a) + c\sqrt{\frac{\log N(b)}{N(b,a)}} \right]$$

where c is the exploration constant that balances exploration and exploitation trade-off, $Q(b, a)$ is the average of the sampled sum of utilities, $N(b)$ is the number of simulations performed through the dialogue state b, $N(b, a)$ is the number of times action a is selected in b. More recent versions of MCTS algorithms have shown great successes in many large sequential-decision making problems such as playing Go (Silver et al., 2016).

PyOpenDial includes an MCTS planner, and we shall demonstrate its effectiveness in the next section by comparing its performance with the forward planner, using the negotiation dialogue domain that requires long-term planning.

Google Speech Modules PyOpenDial provides new speech modules based on Google speech API: The speech recognition module that uses Google speech-to-text API and the speech synthesis module that uses Google text-to-speech API [2]. These modules can also be replaced with other custom-developed speech recognition and speech synthesis modules.

4 Application Domain

In this section, we demonstrate the aptitude of PyOpenDial using the negotiation dialogue domain (Lewis et al., 2017) as an example. In this domain, two agents (i.e. the user and the system) negotiate on 3 types of items, and the negotiation domain has the following unique characteristics: (1) the simulated utterances of the system and the user are generated from the RNN of system and user models, which is done seamlessly thanks to Python-based implementation of the framework, (2) a long-term dialogue planning is required to get a high reward in the negotiation since the utility signal is given only at the very end of the potentially very long dialogue; thus an MCTS planner is desirable.

[2] https://cloud.google.com/text-to-speech, https://cloud.google.com/speech-to-text

```
<initialstate>
 <variable id="rnn">
  <value>@nego.NegotiationRNN</value>
 </variable>
 <variable id="h"><value> </value></variable>
       ...
</initialstate>

<function name="gen_u_m">nego.gen_u_m</function>
<function name="gen_u_u">nego.gen_u_u</function>
<function name="reward">nego.reward</function>
                    ...
<model trigger="u_u">
 <rule>
  <case>
   <condition>
    <if var="current_step" value="Negotiation"/>
   </condition>
   <effect util="0.001">
    <set var="u_m" value="@gen_u_m({rnn},{h},0)"/>
   </effect>
     ...
   <effect util="0.001">
    <set var="u_m" value="@gen_u_m({rnn},{h},19)"/>
   </effect>
  </case>
 </rule>
</model>

<model trigger="u_m" planning-only="true">
 <rule>
  <case>
   <condition>
    <if var="current_step" value="Negotiation"/>
   </condition>
   <effect>
    <set var="u_u" value="@gen_u_u({rnn},{h})"/>
   </effect>
  </case>
 </rule>
</model>

<model trigger="current_step">
 <rule>
  <case>
   <condition>
    <if var="current_step" value="Result"/>
   </condition>
   <effect util="@reward({rnn},{h})">
    <set var="current_step" value="Terminated"/>
   </effect>
  </case>
 </rule>
</model>
                    ...
```

Listing 2: A simplified XML specification for the negotiation dialogue domain.

4.1 Domain Description

In the negotiation dialogue domain, 3 types of items (i.e. *books, hats, balls*) are divided between two agents through natural language dialogue. There is a finite amount of each item (5 to 7 total items and 1 to 4 individual items), and the agents have different utility functions that represent the agent's preference. The utility function for each agent is defined randomly while satisfying the following constraints: (1) The maximally achievable utility for each agent should be 10; (2) Each item must always have a non-zero utility for at least one agent; (3) At least one item must always have a non-zero utility for both agents. If an agreement is reached at the end of the negotiation, each agent receives a reward equal to the total utility of obtained items. If the decisions are in conflict, both agents receive a reward of 0. Figure 2 shows the negotiation dialogue example between user and system in PyOpenDial, and Listing 2 presents a simplified version of the domain XML specification.

4.2 RNN-based Natural Language Generation Model

In implementing this dialogue system, we first pretrained an RNN model that imitates negotiation dialogues between two humans, following the supervised training scheme described in (Lewis et al., 2017). This RNN model has the ability to generate natural language utterances, taking into account the previous dialogue history and the given context (i.e. value and count of each item). We use this RNN to generate candidates of system utterances (line 19-25 in Listing 2) and to generate user utterances for the user simulation model that is used during multi-horizon planning (line 37 in Listing 2). This RNN model uses PyTorch and is imported into PyOpenDial as decribed next.

4.3 Domain XML Specification

In this section, we briefly explain how the negotiation domain is specified in the XML format shown in Listing 2, which is an abbreviation of the full version distributed with PyOpenDial.

Declaration We declare `rnn` state variable, an instance of `NegotiationRNN` class (line 2-4). The class has a pre-trained RNN model, described in the previous section, as a member variable and generates actual (user or system) utterance through the RNN model. To represent the dialogue history, we also declare a state variable `h` (line 5), which maintains the user and system utterances up to the current turn. We then declare two functions, `gen_u_m` and `gen_u_u`, to generate utterances (line 9-10). `gen_u_m` is used to generate the set of candidate system utterances and `gen_u_u` is used as the user simulator in the planner to search for the best system action that maximizes the overall utility within the planning horizon. Finally, we declare the function `reward` which returns the reward at the end of the negotiation (line 11)

System Utterance Generation The utility rule specifies the utility associated with each candidate system action. In order to harness the system utterance directly generated by `NegotiationRNN` and support dialogue planning, we add 20 effects in the utility rule, each corresponding to a system utterance sampled from `NegotiationRNN`, and assign the same *immediate* utility of 0.001 (line 13-28)[3]. The actual, final utility is decided only when the negotiation has finished, and the planner described next will search for the best system utterance (among 20 candidates) using long-term planning.

Planning The planner requires a user simulation model for long-term planning. The user simulation model is given in line 30-41. Note that we set the `planning-only` tag for the user simulation model (line 30), in order to prevent the user simulation model from overwriting the actual user utterance `u_u` during planning. At the end of simulated dialogue, the final utility determined by negotiation is obtained from the python function `reward`. The variable `current_step` is set to "Terminated" to represent the end of a dialogue.

4.4 Experiments

Using the negotiation dialogue domain, we compare the performances of two planning algorithms, the forward planner and the MCTS planner, and a naive baseline that only maximizes the immediate utility without planning. The planning horizons were set to 3 for the forward planner and 7 for the MCTS planner, which made both planners take approximately same amount of search time.

As reported in Table 1, planning (using either Forward or MCTS) improves the negotiation outcome over the baseline in terms of both reward and

[3] (Py)OpenDial includes `None` action with utility 0 by default, thus we assigned small positive utility to the generated utterances to be distinguished from the `None` action.

	Reward	Planning time (s)	% Agreed
Baseline	4.96 ± 0.12	-	81.8
Forward	5.27 ± 0.12	2.28 ± 0.05	87.3
MCTS	5.68 ± 0.12	2.03 ± 0.03	88.9

Table 1: Experimental results for the negotiation example. Baseline denotes the result of negotiation between two RNN models (without planning). Forward and MCTS represents the negotiation result between the corresponding planner and the RNN model. All the results are averaged over 3000 dialogues and report the $2\times$(standard error).

agreement rate, and the MCTS planner further outperforms the forward planner. This is mainly due to the fact that the reward signal in the negotiation domain only comes at the very end of the dialogue, thus in the early stages of the dialogue, no meaningful reward signal can be obtained within the short planning horizon of the forward planner. In contrast, MCTS performs Monte-Carlo simulations all the way towards the end of the dialogue in most cases and thus captures the final utility.

5 Conclusion

In this paper, we presented PyOpenDial, a Python-based open-source dialogue system toolkit that inherits the architectural design of OpenDial and extends the domain XML specification for integrating deep learning models.

We showed the aptitude of PyOpenDial by presenting how the negotiation dialogue domain can be implemented, seamlessly integrating with deep learning model trained for natural language generation. We also demonstrated the efficacy of the new MCTS dialogue planner, significantly outperforming the basic forward planner.

We look forward to active contribution from the developer community towards refining and improving PyOpenDial.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293, 2006.

S. Larsson and D. R. Traum. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural language engineering*, 6(3-4):323–340, 2000.

M. Lewis, D. Yarats, Y. Dauphin, D. Parikh, and D. Batra. Deal or no deal? end-to-end learning of negotiation dialogues. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2443–2453, 2017.

P. Lison. *Structured Probabilistic Modelling for Dialogue Management*. PhD thesis, University of Oslo, February 2014.

P. Lison and C. Kennington. Opendial: A toolkit for developing spoken dialogue systems with probabilistic rules. In *Proceedings of ACL-2016 System Demonstrations*, pages 67–72, 2016.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. 2017.

A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. 2018.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, pages 484–489, 2016.

S. Ultes, L. M. Rojas Barahona, P.-H. Su, D. Vandyke, D. Kim, I. n. Casanueva, P. Budzianowski, N. Mrkšić, T.-H. Wen, M. Gasic, and S. Young. PyDial: A Multi-domain Statistical Dialogue System Toolkit. In *Proceedings of ACL 2017, System Demonstrations*, pages 73–78, 2017.

J. D. Williams, I. Arizmendi, and A. Conkie. Demonstration of AT&T "let's go": A production-grade statistical spoken dialog system. In *2010 IEEE Spoken Language Technology Workshop*, 2010.

J. D. Williams, K. Asadi, and G. Zweig. Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. *CoRR*, abs/1702.03274, 2017.

S. Young, M. Gai, B. Thomson, and J. D. Williams. POMDP-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5), 2013.

L. Yu, W. Zhang, J. Wang, and Y. Yu. Seqgan: Sequence generative adversarial nets with policy gradient. *CoRR*, abs/1609.05473, 2016.