

Demonstration of interactive dialog teaching for learning a practical end-to-end dialog manager

Jason D. Williams
Microsoft Research

jason.williams@microsoft.com

Lars Liden
Microsoft

laliden@microsoft.edu

1 Introduction

This is a demonstration of a platform for building practical, task-oriented, end-to-end dialog systems. Whereas traditional dialog systems consists of a pipeline of components such as intent detection, state tracking, and action selection, an end-to-end dialog system is driven by a machine learning model which takes observable dialog history as input, and directly outputs a distribution over dialog actions. The benefit of this approach is that intermediate quantities such as intent or dialog state do not need to be labeled – rather, learning can be done directly on example dialogs.

In practice, purely end-to-end methods can require large amounts of data to learn seemingly simple behaviors, such as sorting database results. This is problematic because when building a new dialog system, typically no in-domain dialog data exists, so data efficiency is crucial. Moreover, machine-learned models alone cannot guarantee practical constraints are followed – for example a bank would require that a user must be logged in before they are allowed to transfer funds. For these reasons, in past work we introduced *Hybrid Code Networks* (HCN) (Williams et al., 2017). HCNs make end-to-end learning practical by combining a recurrent neural network (RNN) with domain-specific software provided by the developer; domain-specific action templates; and a conventional entity extraction module for identifying entity mentions in text. Experiments on the public bAbI corpus (Bordes et al., 2017) have shown that HCNs can reduce the number of training dialogs required by an order of magnitude compared to state-of-the-art end-to-end learning methods which do not employ domain knowledge.

This demonstration shows a practical implementation of HCNs, as a web service for building task-oriented dialog systems. Once the developer

has provided their domain-specific software, they can add training dialogs in several ways. First, the developer can simply upload dialogs to the training set. Second, the developer can *interactively teach* the HCN, and making on-the-spot corrections. Finally, as the HCN interacts with end-users, the developer can inspect logged dialogs, make corrections if needed, and add the dialogs to the training set.

The next section describes the architecture and operation of the platform, and the final section describes how the developer uses the service – i.e., what the demonstration shows.

2 Dialog learning platform

The practical operation of the HCN is shown in Figure 1, where the left-hand block in white shows a messaging client used by an end user, the center block in blue shows a web service implemented by the system developer that hosts domain-specific logic, and the right-hand block in green is the HCN web service. A software development kit (SDK) facilitates using the HCN web service.

When interacting with end users, the process begins when the end user provides input text, such as “What’s the 5 day forecast for Seattle?”, shown as item 1 in Figure 1. This text is passed to the developer’s web service, which in turn calls the HCN service to perform entity extraction (item 2). The HCN service then returns entity mentions detected in text, such as “location=Seattle” (3). Domain-specific code on the developer’s service the runs to resolve entity mentions to a canonical form, such as a latitude/longitude pair, and to store entities for use in later turns in the dialog (4). The developer’s code then calls the HCN service again, optionally passing in *context* which can include which entities have been recognized so far in the dialog, as well as an action mask that limits which action

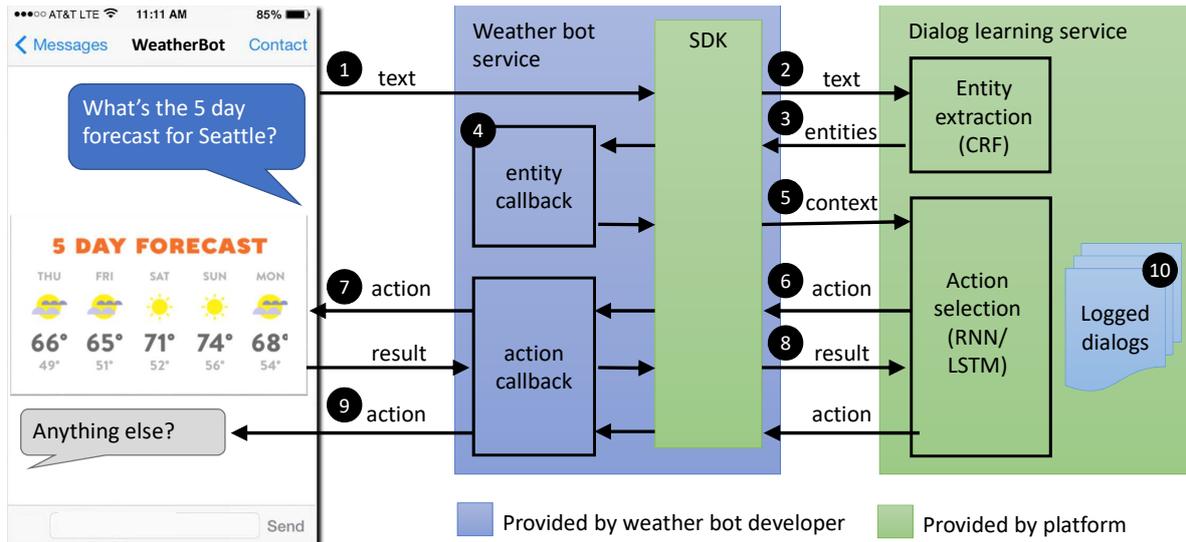


Figure 1: Development platform for interactive dialog learning. See text for details.

templates are available at the current step (5).

The HCN service returns a distribution over all un-masked action templates, and the developer code executes the highest-ranked action (6). If this action template is an API call – such as displaying rich content to the user, executing a transaction in a database, or raising a robot’s arm – that API is invoked (7), and the HCN service is called again to choose the next action. If the API call returns features, those can be passed to the HCN service as context features (8). If the action template is a textual action, the developer’s code can substitute in entity values such as a weather forecast before the text is rendered to the end user (9). The cycle then repeats.

Dialogs conducted with users are logged by the HCN service, and can later be reviewed and corrected by the system developer through a web user interface (9). Also, the cycle can be augmented to support interactive teaching. These aspects are described in the next section.

3 Illustrative interactions

When creating a new dialog system, typically no in-domain data exists. To address this, the dialog learning platform supports *interactive teaching*. In interactive teaching, the developer alternates between the role of the end user, and the role of the teacher. The operational loop shown in 1 is modified so that results of entity extraction and action selection can be corrected before being executed.

Figure 4 shows an example of interactive teach-

ing for pizza ordering. The developer – playing the part of the user – enters “medium pizza with olives”. The current entity extraction model finds entity mentions for the \$pizza and \$size entities, but not the “olive” \$stopping. So, the developer corrects this by adding a corrected entity label, and this *corrected label* is used going forward. The interface then displays the contents of the developer-defined state, and provides a list of actions, each with their score under the current RNN model. In this example, all of the actions are shown as “disqualified”, meaning that the action mask prohibits them. For example, the action “Would you like a Small, Medium, or Large \$crust pizza ...” is masked because the pizza size is already known. The developer enters the index of the action to take (“1”) and the dialog continues. At this point, the developer could have alternatively entered a new action – for example, by typing “So you want \$stoppings, is that right?”. As each correction is made, the model is re-trained.

Once a rudimentary model is in place, end-users can start using the system. An example dialog with an end-user is shown in Figure 2, which shows an error at the last system turn. Figure 3 shows how this dialog appears to the developer, and how a correction can be made. Each system utterance is shown in a drop-down box. If the developer identifies a turn where the system output the wrong action, the developer can select the correct action from the drop-down. When an action which differs from the action in the log is se-

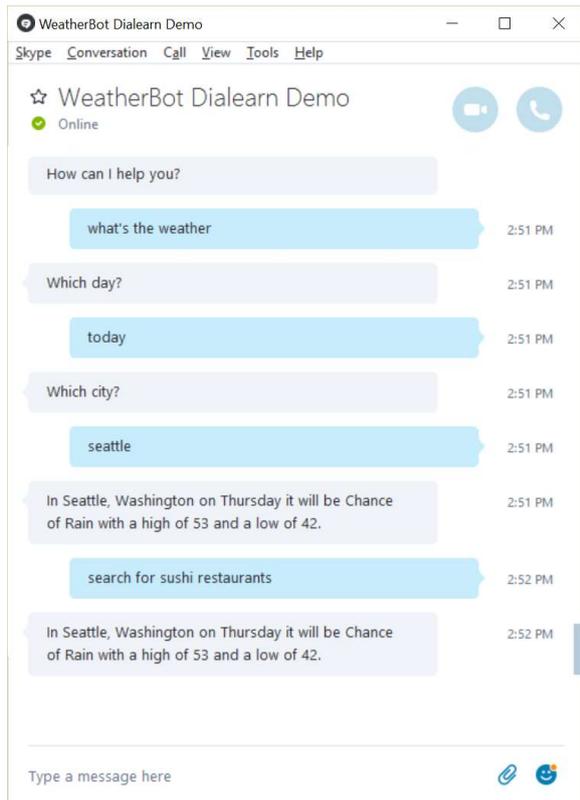


Figure 2: Example interaction with an end user. Note the system mistake after the user enters “search for sushi restaurants”.

lected, the remainder of the dialog is discarded, since it is no longer known how the user would have responded. If none of the actions is appropriate, the developer can choose “new action...”, and enter a new action into a provided text box. When the dialog has been corrected, the developer clicks on “submit”, which saves the labeled dialog to the training set, re-trains the model, and re-deploys the new model. In the example in Figure 3, the user’s fourth input was “search for sushi restaurants”, and the system had answered with a weather forecast. The developer changed this response to “new action...” and typed in the new action “Sorry, I can’t help with that”.

In the demonstration, we have three working dialog systems available, for pizza ordering, restaurant information, and weather forecasts. The demonstration shows applying the two interactive methods above to each of these three domains.

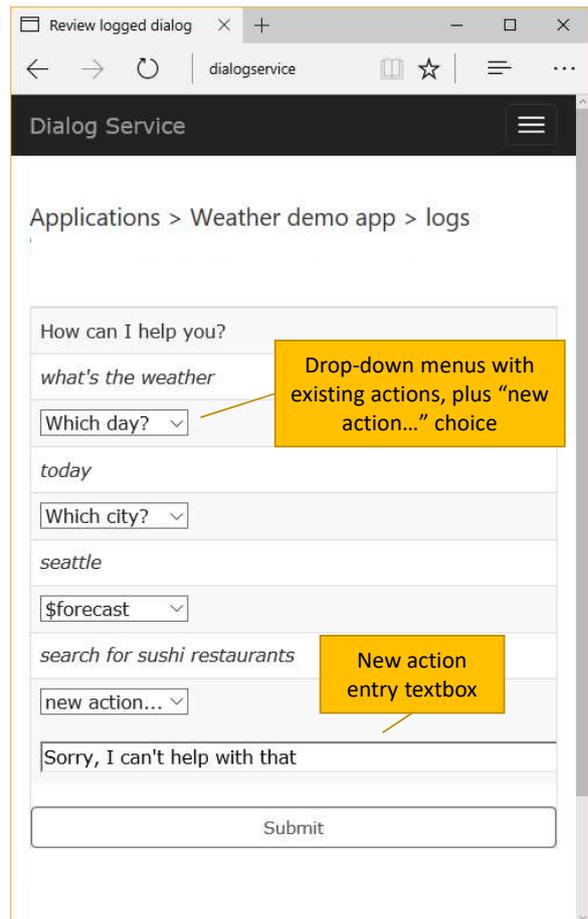


Figure 3: Example of off-line dialog correction, showing the dialog collected in Figure 2. After the user says “search for sushi restaurants”, the developer changed the action “\$forecast” to “new action...” and typed in “Sorry, I can’t help with that”.

References

- Antoine Bordes, Y-Lan Boureau, and Jason Weston. 2017. Learning end-to-end goal-oriented dialog. In *Proc Intl Conf on Learning Representations, Toulon, France*.
- Jason D Williams, Kavosh Asadi, and Geoffrey Zweig. 2017. Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. In *Proc Association for Computational Linguistics, Vancouver, Canada*.

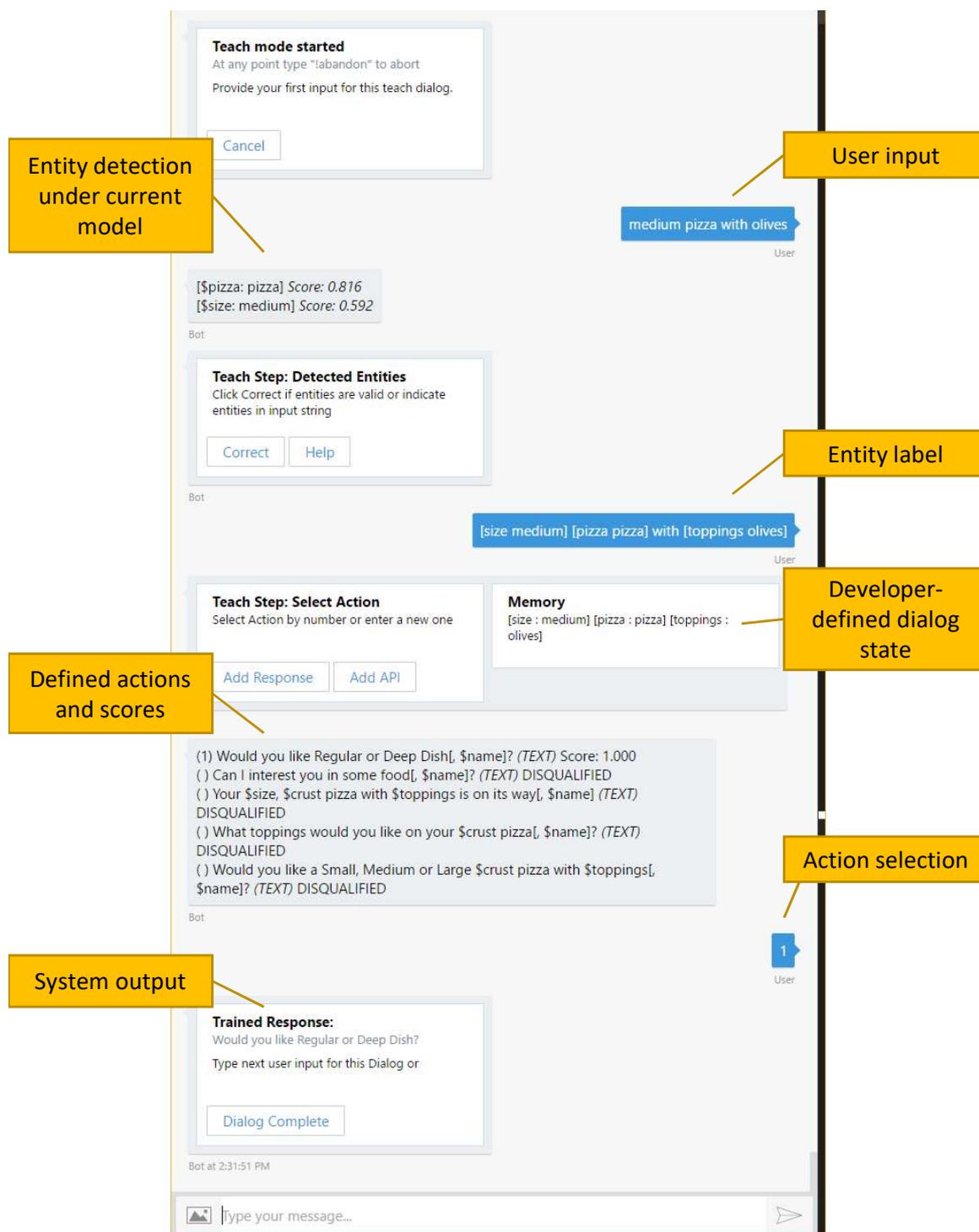


Figure 4: Example of interactive dialog teaching. The developer's input is in blue boxes on the right side, and the system's responses are in grey and white boxes on the left side. The developer alternates between playing the role of an end user, and providing corrective input.