Extended Hybrid Code Networks for DSTC6 FAIR Dialog Dataset

Jiyeon Ham, Soohyun Lim*, Kee-Eung Kim*

KAIST, Republic of Korea

{jyham, shlim}@ai.kaist.ac.kr, kekim@cs.kaist.ac.kr

Abstract

Goal-oriented dialog systems require a different approach compared to chit-chat conversations in that they should perform various subtasks as well as the dialog itself. Since the systems typically interact with an external database, it is efficient to import simple domain knowledge in order to deal with the external knowledge changes. This paper presents extended hybrid code networks for sixth dialog system technology challenge (DSTC6) Facebook AI research (FAIR) dialog dataset. Compared to the original hybrid code networks (HCNs), we reduced the required hand-coded rules and added trainable submodules. Due to the additional learning components and reasonable domain-specific rules, the proposed model can be applied to more complex domains and achieved 100% accuracies for all test sets.

Index Terms: dialog system technology, goal-oriented dialog system, fully-trainable hybrid code networks, DSTC6

1. Introduction

Goal-oriented dialog systems help users to achieve their desired goals via natural spoken language, such as making phone calls, booking movie tickets, reserving restaurants, and scheduling. To perform these tasks, the system should not only chat with users, but also interact with external knowledge. For example, a movie booking system should know currently released movies, screening schedules, movie ratings, then provide the information to users or apply users' requirements to the knowledge base. Because knowledge bases are possibly updated frequently, goal-oriented dialog system should be able to deal with unseen knowledge.

DSTC6 FAIR dialog dataset is a simulated goal-oriented conversation. The goal of the bot is to reserve a restaurant which meets the user's criteria and to provide additional information to the user. The bot should perform extra tasks to achieve the goal such as asking, querying the database, recommending, and providing additional information.

Conventional goal-oriented dialog systems are designed with tracking the dialog state by slot-filling [1], [2], [3], [4]. Young et al. [5] and Wang et al. [6] treated the goal-oriented dialog systems as partially observable Markov decision process (POMDP) agent interacting with users to reach their goals. In this approach, the dialog state should abbreviate all dialog history so have to be carefully designed by domain experts. Furthermore, the action is highly dependent on the designed state so the systems have difficulty to be applied to other domains.

Recently, end-to-end dialog systems using recurrent neural networks (RNNs) have been proposed to solve such difficulties [7], [8], [9]. RNNs are trained to generate a latent state representation rather than a human-defined dialog state. Although RNNs have shown good performance mainly in chit-chat domains, learning to interact with external databases is difficult for RNNs. Therefore they cannot be straightforwardly applied to the goal-oriented settings.

Bordes et al. [10] have shown that end-to-end memory networks [11] have a capacity to learn the interactions with a database. However, some tasks which need to operate on the database such as sorting are hard to be trained even if the operation is elementary. It is also difficult to adapt to changes in domain knowledge, such as changes in the number of required fields to query the database.

Hybrid code networks (HCNs) [12] combined some hand-coded rules and RNN to manage external knowledge while learning latent dialog state. Our approach is based on HCN that is a state-of-the-art goal-oriented dialog system and gets perfect accuracy on the dialog bAbI datasets. Since the DSTC6 FAIR data is more complicated, applying HCN to DSTC6 needs carefully designed code or additional learning procedures.

We suggest extended hybrid code networks which use trainable networks not only to extract latent dialog state but also to interact with external knowledge. Extending the trainable networks facilitates more complex operations on external knowledge by understanding various user's intents. The proposed model shows good performance in DSTC6.

This paper is organized as follows. Section 2 describes dataset and reviews the previous related work. Section 3 explains our model part-by-part. Section 4 discusses our training method and the results then section 5 concludes.

Table 1: *List of action templates. To construct fully-formed outputs, symbols should be filled with corresponding values.*

Utterance type
ok let me look into some options for you
api_call `list of api slots`
i'm on it
hello what can i help you with today
sure is there anything else to update
you're welcome
what do you think of this option: `<R_name>`
great let me do the reservation
sure let me find another option for you
here it is `<R_address>`
here it is `<R_phone>`
whenever you're ready
the option was `<R_name>`
i am sorry i don't have an answer to that question
is there anything i can help you with
`request_api_slot`

*: These authors contributed equally.

Table 2: *List of slot requesting questions.*

Required slot	**Utterance type**
`R_cuisine`	any preference on a type of cuisine
`R_location`	where should it be
`R_number`	how many people would be in your party
`R_price`	which price range are you looking for
`R_atmosphere`	are you looking for a specific atmosphere
`R_restrictions`	do you have any dietary restrictions

2. Background

2.1. DSTC6 FAIR dialog dataset

DSTC6 FAIR dialog data, expanded version of dialog bAbI [10], is a set of goal-oriented dialogs between a human and a bot that aims to reserve a table at a restaurant. In the course of achieving the goal, the dataset is divided into the following subtasks.

- Task 1: Issuing API calls
- Task 2: Updating API calls
- Task 3: Displaying options
- Task 4: Providing extra information
- Task 5: Conducting full dialogs

Each dialog proceeds based on the underlying tabular knowledge base (KB) which consists of restaurant information. The system is able to query the KB via API call with particular fields. All the dialogs in the training data are generated on the same KB and have 5 required fields to query: a type of cuisine, a location, a price range, a party size, and an atmosphere.

Test data is divided into 4 types by two criteria. One is the underlying KB used to generate the data, and the other is the number of fields that should be filled with the API call.

The two of the test sets are generated with the KB that was used for the training data, and the other two are generated with another KB. These two KBs have disjoint sets of restaurants, locations, cuisines, phones, and addresses. The test data generated by the new KB is termed Out-Of-Vocabulary (OOV) because the bot will face the words that it has not seen in the training phase.

In addition, in two test sets, API call requires five slot values as in the training data. However, in the remaining two test sets, dietary restriction is additionally required to query the KB.

Therefore the 4 types of test data are: (1) no OOV and no unseen slot; (2) OOV and no unseen slot; (3) no OOV and one unseen slot; and (4) OOV and one unseen slot.

2.2. Hybrid Code Networks

Hybrid code networks are composed of four major components: entity extraction; domain-specific software; domain-specific action template; and RNN. Entity extraction and RNN are trainable components, whereas domain-specific software and domain-specific action template are hand-coded.

Entity extraction module identifies entities in utterance, like name, place, time, etc. If an utterance "I'd like to have an Italian food for dinner" is given, the module extracts 'Italian' as a cuisine entity. Then the entity is replaced with an entity symbol: "I'd like to have an `<R_cuisine>` food for dinner". Then the symbolized utterance is encoded to a bag-of-words (BoW) vector. Pretrained word embedding model can be used additionally.

Entity tracking module, which is a part of the domain-specific software, keeps entities that user intended. Based on the entities it defines hand-coded dialog state called context feature. In the dialog bAbI data set, the context feature vector can be characterized as the presence of each entity. Entity tracking module also produces action mask based on context feature, which masks illegal actions in the given context.

While user utterances are encoded as BoW vectors, bot utterances are specified as one of the domain-specific action templates. HCNs assume that the type of bot utterances are restricted. A templatized action is encoded as a one-hot vector which represents the type of action.

BoW vector, optionally embedded vector, context feature, and bot action vector in the previous turn are concatenated and passed to the RNN at that time step. The RNN computes latent dialog state and the latent state is passed to dense layer with softmax activation function to determine the bot action at the time step.

The templatized bot action becomes fully-formed action going through entity output module, which is a part of the domain-specific software. For example, an action template 'api_call `<R_price>` `<R_cusine>` `<R_location>` `<R_cuisine>` `<R_number>`' has unfilled entity slots. Entity output module fills in these slots using hand-coded rules and the stored entities from the entity tracking module.

3. Proposed approach

Our approach is an extension of HCN. We added trainable parts to the entity tracker and the entity output module which were designed with developer codes in the original HCN.

The entity tracking module in the original HCN is implemented with rules. Since there are much more various types of utterances in the realistic situations, it can be intractable to make proper rules. We revised the module to contain a trainable part to cover more realistic situations.

The entity output module for filling the entity also needs to interpret the user utterance. For example, in the templatized action output "the option was `<R_name>`", the value for the entity symbol `<R_name>` should be different by the previous user utterance. To understand and handle the various user demands, we added a trainable part to the entity output module.

3.1. Overall Structure

The overall structure of our model (Figure 1) inherits the original HCN. As we mentioned above, we added trainable parts named API slot-value tracker and API result selector to the domain-specific software. Domain-specific action template and the entity extraction module are also modified to deal with unseen slots.

Entity extraction finds out entities in user utterance by simple string match as original HCN. However, while earlier HCN replaces entities to entity symbols for each entity type, our model groups the entities needed for API slots and maps them to one entity symbol. Entity values corresponding to the types `<R_cuisine>`, `<R_location>`, `<R_number>`, `<R_price>`, `<R_atmosphere>`, and `<R_restrictions>` are replaced to `<R_value>` as grouped together. This makes our model be able to handle unseen slots and values.

Action selector, corresponding to the RNNs of the original HCN, determines the type of action a_t that the bot should take at time step t based on the dialog history H_{t-1}, current symbolized user utterance u_t, and a current context feature c_t.

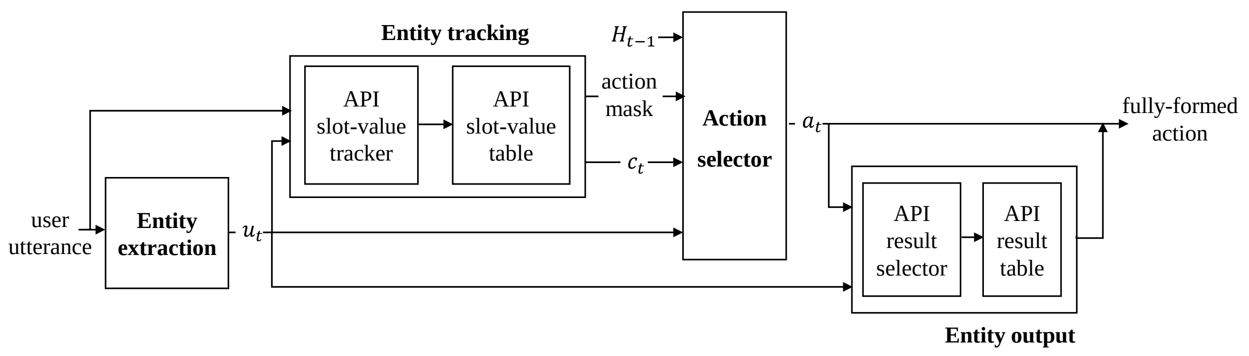

Figure 1: *Overall structure of extended HCN. The parts written in bold correspond to each part of the original HCN. Entity tracking and entity output are modified to be trainable, which were originally hand-coded.*

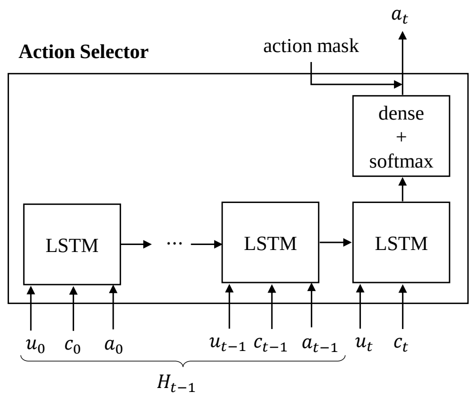

Figure 2: *An illustration of the action selector.*

The dialog history H_{t-1} represents the sequence of all user utterances, bot actions and context features until time step $t-1$ ($H_{t-1} = [[u_0, c_0, a_0], ..., [u_{t-1}, c_{t-1}, a_{t-1}]]$).

API slot-value tracker helps the system query the database by tracking the values of the API slots. It receives a symbolized user utterance u_t and finds entity symbols to track. In addition, this module produces a context feature c_t and action mask as the entity tracking in the original HCN.

API result selector comprehends a symbolized user utterance u_t and determines a row where the bot should pay attention in the API call results. In the DSTC6 data, each row of the API call result table represents a particular restaurant. With the selected restaurant, the entity output module fills the symbols, `<R_address>`, `<R_phone>`, and `<R_name>`.

3.2. Action Selector

16 domain-specific action templates for DSTC6 are shown in Table 1. We categorized 6 special utterances (Table 2) which ask users for empty API slot values as an action template '`request_api_slot`'. If the action selector chooses '`request_api_slot`' as an action, it is replaced with one of the fully-formed actions in Table 2. This allows the system to cope with the change in API slots. If the API slots change, the system only needs to change the action template of 'api_call' and the list of available utterances for '`request_api_slot`'. Then action selector can adapt to external knowledge changes as API slots.

Action selector (Figure 2) consists of LSTM [13], dense, and softmax layers. In order to determine the bot action a_t, the LSTM receives user utterance, bot action, and context of each turn until the $(t-1)$-th step. At the last time step t, LSTM receives only the user utterance u_t and the context c_t for t-th step. The output 16-dimensional vector represents the probability for each of the 16 action templates.

3.3. API slot-value tracker

API slot-value tracker (Figure 3) finds out the user intended value in an input sentence, and if so, fills the API slot-value table. This module observes one sentence and grasps its meaning, while the action selector sees the entire flow of the conversation. For example, in a symbolized user utterance "find me one in `<R_value>`, `<R_value>` will be too complicated", API slot-value tracker should find out that the first symbol is intended by the user. Furthermore, the value corresponding to the first symbol is stored in the API slot-value table.

Based on the API slot-value table, entity tracking module generates a context feature c_t and an action mask. context feature is a hand-coded dialog state which helps action selector to decide. In our model, context feature represents whether all the API slot values are known or not. An action mask enables developers to restrict the bot actions based on domain-specific knowledge. In the DSTC6 dataset, the action template 'api_call' requires all slot values, so we restricted the action template 'api_call' when any unfilled slot value exists.

The model is build up with LSTM, dense and softmax layers. For a given user utterance, each word in the utterance is embedded to a one-hot vector and goes into the LSTM sequentially. The softmax layer outputs the probability vector noting the symbol intended by the user.

3.4. API result selector

If a result table is obtained through an API call, the system should properly display the result and find out the user's intention in the API result table.

The policy to display the API results is implemented by a code. In the DSTC6 dataset, the system should display the API call result by recommending restaurants in order of rating. It needs a lot of data to learn the sorting operation while it can be expressed by a simple code.

API result selector (Figure 4) looks one user utterance at a time as in the API slot-value tracker. For DSTC6 dataset, a

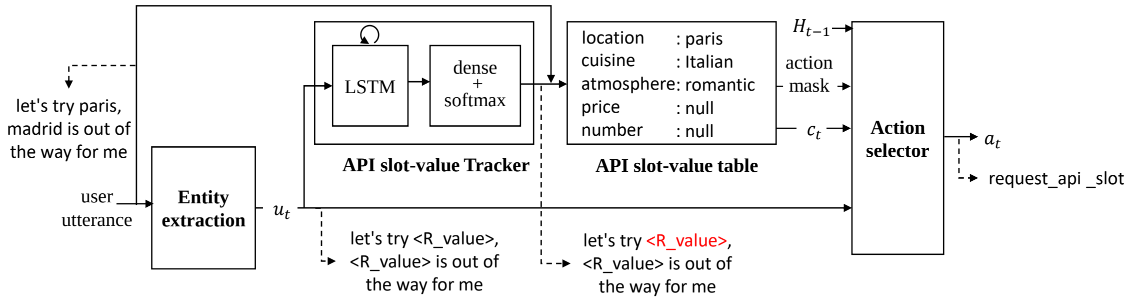

Figure 3: *An illustration of API slot-value tracker.*

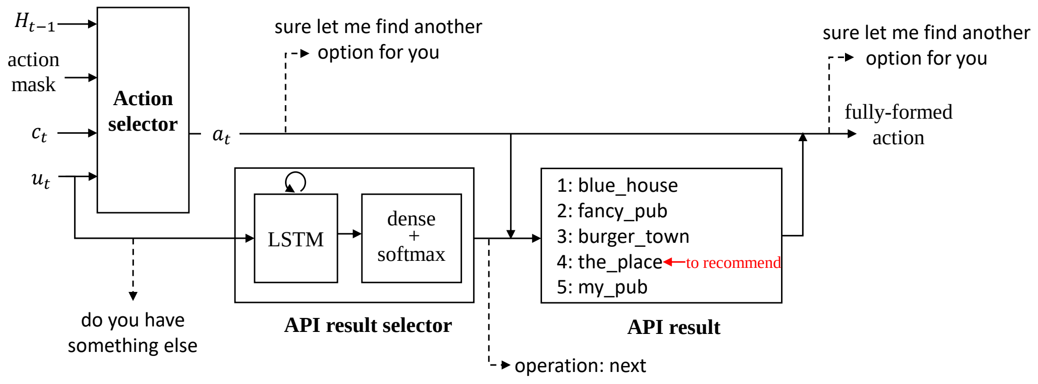

Figure 4: *An illustration of API result selector.*

user's intention means a restaurant which the user wants. While the system recommends a restaurant, the user can accept or reject. In addition to accepting, the user can also ask to mention a previously recommended restaurant again. The system should be able to interpret and track the user's intent correctly, since the user may accept not only the current but also the past recommendation.

The system must refer the API result table in order to perform some tasks such as recommending, repeating, and providing additional information. The system should remember restaurant to recommend next, last mentioned restaurant, and finally decided restaurant in the table. Because the recommendation order is fixed, it is sufficient to save the restaurant to be recommended in the table to remember the recommendation history.

A user intention is classified into the predefined operations on the API result table. These operations are executed only when the action selected by the action selector allows the operations. In this dataset, "sure let me find another option for you", "the option was <R_name>', and 'great let me do the reservation" are the only action that allows the operations. Our predefined operations in this dataset will be described in more detail in section 4. We can add additional predefined operations for the extended real-world data.

If the selected action template requires some information from the API result table, it can be done by putting the information of the stored restaurant. For example, if "here it is <R_address>" is selected, the entity output module fill the entity <R_address> by the address of the stored restaurant. On the other hand, if the action allows an operation, use the stored restaurant after the proper operation. For the selected action "the option was <R_name>", the module enters the name of the restaurant after the selected operation performed.

Table 3: *Labels for API slot value tracker. Red colored symbols are intended by users.*

label	utterance example
nowhere	i got <R_value> last time so i may be due for a change.
first	find me one in <R_value> <R_value> will be too complicated.
second	i was thinking <R_value> but my friend prefers <R_value> so let's do that.

4. Train method and results

To train API slot-value tracker and result selector, we should generate an additional labeled data. User utterances containing the API slot symbols are used to train API slot-value tracker. A label should note which symbol is intended by the user among the several symbols in the utterance. We created the labels according to the following observations on the data. First, the intended API slot values always appear in the api_call action of each dialog. Second, each user utterance contains at most two API slot values for each slot. Therefore we implemented the API slot-value tracker to classify each input user utterance to

Table 4: *Labels for restaurant requesting utterances to train API result selector.*

label	**utterance example**
first	what was the first option again
previous	sorry what was the previous option again
new	i still don't like that

Table 5: *Labels for restaurant determining utterances to train API result selector.*

label	**utterance example**
last recommended	let's go with the last one
last mentioned	let's do it

three classes by the position of the intended symbol: nowhere; the first symbol; and the second symbol. Examples are shown in Table 3.

API result selector module also needs additional labels to be trained. The labels represent the predefined operations on API result tables. API result selector plays two roles. First, it determines a restaurant to mention based on the user's previous request. We observed that user's request are one of these three types: repeat the first one among preceding recommendations, repeat the previous recommendation or recommend a new restaurant. Thus we labeled the user utterances prior to the bot actions that provide restaurant names to users. We determined whether the provided restaurant was a new one, or the first recommended one, or the previously recommended one. Examples are shown in Table 4.

The second role of API result selector is to determine a restaurant which a user finally confirm. In the DSTC6 dataset, while the entity output module stores the last recommended restaurant and the last mentioned restaurant, the finally decided restaurant is one of them. The bot action 'great let me do the reservation' means user made a decision by the utterance just before the bot action. User utterances prior to the bot action are labeled with the last recommended or the last mentioned. Examples are shown in Table 5.

Our model reached 100% precision for all the test sets and tasks. The results are shown in Table 6.

5. Conclusion

This paper presented extended hybrid code networks for DSTC6 FAIR dialog data. HCN is able to tackle the changes in domain knowledge while taking a hybrid of domain-specific rules and a trainable component. As we took the advantage of original HCN, we extended HCN to learn to interpret the user utterance. Therefore, our model showed a perfect score in DSTC6.

Although our model shows good performance on the DSTC6 dataset, the model should achieve some improvement to apply it to a more realistic domain. Since our extended modules, API slot-value tracker and API result selector are designed domain specifically, our future work is making these modules more flexible. At now the extended modules are designed to interpret user utterances one by one. Our idea for future work is making our extended modules to take a whole dialogue history and find words which the user intends. This is expected to be done by attention mechanism [14] and it can reduce domain-specific engineering. The proposed model has a potential to be applied to the real world as well as the simulated data.

Table 6: *Results on DSTC6 FAIR dialog dataset. Precision is calculated only for the first rank among the 10 candidates.*

Test set	**Task**	**Precision**
No OOV and no additional slot	1	1.0
	2	1.0
	3	1.0
	4	1.0
	5	1.0
OOV and no additional slot	1	1.0
	2	1.0
	3	1.0
	4	1.0
	5	1.0
No OOV and additional slot	1	1.0
	2	1.0
	3	1.0
	4	1.0
	5	1.0
OOV and additional slot	1	1.0
	2	1.0
	3	1.0
	4	1.0
	5	1.0

6. Acknowledgements

This material is based upon work supported by the Ministry of Trade, Industry & Energy (MOTIE, Korea) under Industrial Technology Innovation Program (No.10063424, Development of distant speech recognition and multi-task dialog processing technologies for in-door conversational robots).

7. References

[1] M. Henderson, B. Thomson, and S. Young, "Word-based dialog state tracking with recurrent neural networks," in *in Proceedings of SIGdial*, 2014.

[2] G. Mesnil, Y. Dauphin, K. Yao, Y. Bengio, L. Deng, D. Hakkani-Tur, X. He, L. Heck, G. Tur, D. Yu, and G. Zweig, "Using recurrent neural networks for slot filling in spoken language understanding," *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, vol. 23, no. 3, pp. 530–539, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1109/TASLP.2014.2383614

[3] Y.-Y. Wang, L. Deng, and A. Acero, "Spoken language understanding," *IEEE Signal Processing Magazine*, vol. 22, no. 5, pp. 16–31, 2005.

[4] Y. He and S. Young, "A data-driven spoken language understanding system," in *Automatic Speech Recognition and Understanding, 2003. ASRU'03. 2003 IEEE Workshop on*. IEEE, 2003, pp. 583–588.

[5] S. Young, M. Gai, B. Thomson, and J. D. Williams, "Pomdp-based statistical spoken dialog systems: A review," *Proceedings of the IEEE*, vol. 101, no. 5, pp. 1160–1179, May 2013.

[6] Z. Wang and O. Lemon, "A simple and generic belief tracking mechanism for the dialog state tracking challenge: On the believability of observed information," in *Proceedings of the SIGDIAL 2013 Conference*, 2013, pp. 423–432.

[7] I. V. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau, “Building end-to-end dialogue systems using generative hierarchical neural network models,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI’16. AAAI Press, 2016, pp. 3776–3783. [Online]. Available: http://dl.acm.org/citation.cfm?id=3016387.3016435

[8] L. Shang, Z. Lu, and H. Li, “Neural responding machine for short-text conversation,” *CoRR*, vol. abs/1503.02364, 2015. [Online]. Available: http://arxiv.org/abs/1503.02364

[9] A. Sordoni, M. Galley, M. Auli, C. Brockett, Y. Ji, M. Mitchell, J. Nie, J. Gao, and B. Dolan, “A neural network approach to context-sensitive generation of conversational responses,” *CoRR*, vol. abs/1506.06714, 2015. [Online]. Available: http://arxiv.org/abs/1506.06714

[10] A. Bordes and J. Weston, “Learning end-to-end goal-oriented dialog,” *CoRR*, vol. abs/1605.07683, 2016. [Online]. Available: http://arxiv.org/abs/1605.07683

[11] S. Sukhbaatar, a. szlam, J. Weston, and R. Fergus, “End-to-end memory networks,” in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2440–2448. [Online]. Available: http://papers.nips.cc/paper/5846-end-to-end-memory-networks.pdf

[12] 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*, vol. abs/1702.03274, 2017. [Online]. Available: http://arxiv.org/abs/1702.03274

[13] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov 1997.

[14] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: http://arxiv.org/abs/1409.0473