MACA: A Modular Architecture for Conversational Agents

Hoai Phuoc Truong[*], **Prasanna Parthasarathi**[†], **and Joelle Pineau**[‡]
School of Computer Science
McGill University

Abstract

We propose a software architecture designed to ease the implementation of dialogue systems. The Modular Architecture for Conversational Agents (MACA) uses a plug-n-play style that allows quick prototyping, thereby facilitating the development of new techniques and the reproduction of previous work. The architecture separates the domain of the conversation from the agent's dialogue strategy, and as such can be easily extended to multiple domains. MACA provides tools to host dialogue agents on Amazon Mechanical Turk (mTurk) for data collection and allows processing of other sources of training data. The current version of the framework already incorporates several domains and existing dialogue strategies from the recent literature.

1 Introduction

Recent research in building sophisticated AI-based dialogue management systems has led to many new models supporting goal oriented or chit-chat style dialogue agents. These models have been applied to a variety of consumer domains, such as restaurant booking (Kim and Banchs, 2014), flight booking (Young, 2006), etc. However, the lack of tools for easy prototyping of newer models remains an impediment to developing new models and properly benchmarking against previous models. Furthermore, the different types of conversational agents– e.g., generative (Hochreiter and Schmidhuber, 1997; Serban et al., 2015, 2016), retrieval-based (Schatzmann et al., 2005a; Lowe et al., 2015a), slot-based (Young, 2006) or POMDP agents (Png and Pineau, 2011)– have different working mechanisms, which pose challenges to the development of a unified platform for conversational agents with multi-domain support.

To address this gap, we propose a new, ready-to-use, cross-platform framework for text-based conversational agents – MACA[1](**M**odularized **A**rchitecture for **C**onversational **A**gents)– that supports *plug-n-play* use of several existing dialogue agents, as well as facilitates easy prototyping of new dialogue agents. The architecture simplifies the specification of different types of dialogue agents and plugs in an already-built dialogue agent. The framework also maintains a clear separation between domain knowledge and the dialogue agent, which improves agent and domain knowledge reusability. MACA separates task definition from task selection and thereby supports multi-task agents that can extend to multiple turns.

The key characteristics of the MACA framework include:

- strong separation between domain knowledge and a dialogue agent
- a unified architecture to support goal-oriented, POMDP, generative, and retrieval-based dialogue agents
- easy plug-n-play of custom-built agents
- multi-task support for domain specification
- reusability of slots across different tasks
- tool to collect data from mTurk with ease
- template to construct dialogue agents within the framework
- independence from dialogue agents' implementation libraries
- open source code ready for public sharing

[*] phuoc.truong2@mail.mcgill.ca
[†] prasanna.p@cs.mcgill.ca
[‡] jpineau@cs.mcgill.ca

[1] https://github.com/ppartha03/MACA

Proceedings of the SIGDIAL 2017 Conference, pages 93–102,
Saarbrücken, Germany, 15-17 August 2017.

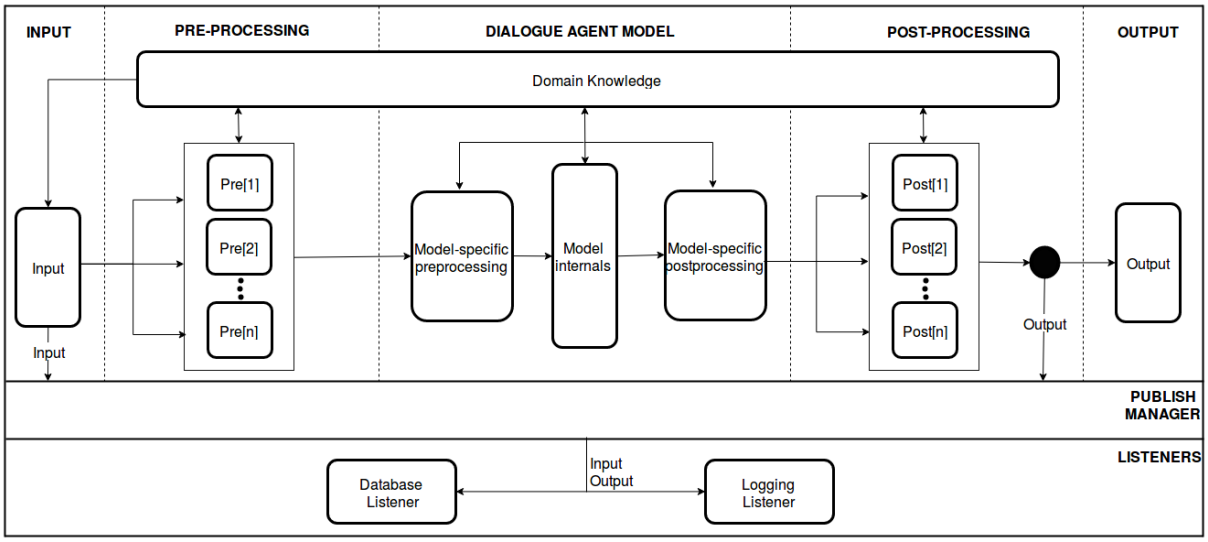

Figure 1: Overview of MACA: A Modular Architecture for Conversational Agents.

2 Related Work

There are a few proposed frameworks in recent years that provide easy prototyping of dialogue agents.

Ravenclaw (Bohus and Rudnicky, 2003), proposed as a successor to Agenda (Allen et al., 2001), is a two-tiered dialogue architecture supporting rapid development of dialogue agents. This flexible architecture provides a clear separation between the domain knowledge and dialogue agent, and maintains a hierarchical task structure. Systems can be built on the architecture with the hierarchical task layout but adding a new task requires the hierarchy to be rebuilt, which impedes application to new domains.

A hierarchical architecture similar to Ravenclaw, called Task Completion Platform (TCP) (Crook et al., 2016), addresses domain knowledge extensibility with minimal changes to a configuration file. In addition, it allows the goal oriented tasks to be defined easily using a TaskForm language to maintain slot information. Although TCP facilitates extension of slot-based agents to multiple domains, it cannot be extended for other dialogue agent types *viz.,* generative models and retrieval models.

Another notable architecture is ClippyScript (Seide and McDirmid, 2012), but its task definition is tied to a task condition by rule. Rules are therefore constrained to be explicitly defined on a per task basis. This is significantly more restrictive than our proposed architecture.

As much research focuses on proposing different architectures for dialogue models, there have also been some progress made in proposing efficient protocols for agent-agent interaction such as DialPort (Zhao et al., 2016), which provides tools for enabling multi-modal interaction between agents. Our proposed work is different from this line of research, focusing on a unifying architecture for dialogue agents and little on the inter-agent communication.

3 Architecture Description

An overview of the Modular Architecture for Conversational Agents (MACA) is presented in Figure 1. The system is setup as a pipeline with six major components: *Input, Pre-processing, Dialogue Model, Post-processing, Output, and Listeners*. Each component contains independent subcomponents that interact across it. All components within the architecture abstract away their underlying implementations and therefore allow their extensions to be straightforward. This helps in block-wise designing of newer systems by preserving the original functionality, yet also providing a free hand in customizing of each component.

3.1 Component Details

3.1.1 Domain Knowledge

Domain knowledge contains static background information about the conversation topic. This can take the form of training data (e.g. transcribed conversations), constants, dictionaries, or restrictions on produced responses (e.g. sentence length, banned phrases). Data stored in domain knowl-

edge must be independent of the model implementation, and can be shared between different models and components.

3.1.2 Input

The *Input* module provides or generates input utterances (i.e. statements, sentences) to the conversation pipeline. This component represents an abstract input device whose source of context varies depending on the use case. This could include a database of previous collected conversations, a terminal interface (i.e. stdin) to acquire data in real-time, or a web interface to a data source (e.g. mTurk).

3.1.3 Preprocessing

The *Pre-processing* module serves as a bridge between raw data acquired via the *Input* component and the input format required of components of the *Dialogue model* module. The system architect may choose to include one or several pre-processing operations within this module. These pre-processing operations by default are performed in parallel and their results are fed into the next component as an array. This allows the dialogue model to have multiple input representations. Alternatively, the framework also allows these operations to be sequentially processed in a specified order (e.g. spelling correction, followed by stemming).

Pre-processing operations currently implemented in MACA include: getting POS tags, removing stop-words, sentence tokenizing (Loper and Bird, 2002), Byte-Pair encoding (BPE) (Gage, 1994) and can be extended to accommodate trained sentence2vec model (Le and Mikolov, 2014), trained word2vec model (Mikolov et al., 2013), etc. These nodes can also interact with the *Domain Knowledge* component to acquire domain specific information required for the operations.

3.1.4 Dialogue Model

This module is the core of the architecture, and contains implementations of agents capable of producing dialogue acts in response to the pre-processed Input information. This module can have up to three sub-components: *Model Specific Pre-processing*, *Model Internals* and *Model Specific Post-processing*, to accommodate dialogue agent models with various interface requirements.

The *Model internals* sub-module contains the central dialogue model, which may be an existing model, such as a POMDP (Png and Pineau, 2011), Dual Encoder (Lowe et al., 2015a), HRED agent (Serban et al., 2015), or a newly designed model. This sub-module receives inputs from the *Model Specific Pre-processing* sub-module. The space of possible responses, vocabulary or dialogue acts are stored in the *Domain Knowledge* module. The *Model internals* and *Model specific Pre/Post-processing* sub-modules share the model information. Similar to the *Pre-processing* component, they can access any information required for their operations by querying the *Domain Knowledge* component. A specific illustration of this interaction is in goal-oriented dialogue agents, where the slot information – *askQueries* and other attributes of the slot and these slot objects – are maintained in the domain knowledge, which enables the framework to support multiple agents. In such settings, the Dialogue Model is initialized with a generic agent that tries to gauge the user intent, and then queries the domain knowledge for the appropriate slots.

Model specific Pre-processing and Post-processing sub-components are provided to give the luxury of designing fine-tuned pre-processing for a model. *Model Specific Pre-processing* sub-component transforms pre-processed input(s) into appropriate representations compatible with the model internals (e.g. array of word indices into vector, matrix or lookup table, etc). On the other hand, *Model Specific Post-processing* sub-component transforms model outputs into more comprehensible forms for the next independent component in the system (e.g. matrix/vector representation to array of words/sentences).

Although certain interpretations suggest analogies between the above sub-modules and conventional units of a goal-oriented dialogue system such as Dialogue Manager (DM) as *Model internals*, Natural Language Understanding (NLU) as *Model specific Pre-processing*, and Natural Language Generation (NLG) as *Model specific Post-processing*, MACA does not impose any restriction on how the framework's sub-modules should correspond with these conventional parts of a dialogue system. For example, the architect may choose to have the *Model internals* sub-module act as a NLU unit, while *Model specific Post-processing* act as both NLG Unit and DM unit.

In addition, as the model may also be an ensemble of dialogue models, the model specific pre-

and post-processing sub-components can also be used to keep processing units specific to each of the model in the architecture. For clarification, in a typical implementation of an ensemble of models, the *Model specific Pre-processing* sub-component can be used to provide separate inputs parsed from the *Pre-processing* component to the corresponding models, while *Model specific Post-processing* sub-component can be used to perform a majority voting or other ensemble techniques to select the response *pool*.

3.1.5 Postprocessing

The Posprocessing component connects the *Dialogue Model* and the *Output* components. It allows the architect to choose the response in the case of multi-response retrieval, to alter responses based on linguistic characteristics, or to modify a response in accordance with the conversation domain. It may also serve as a translation of text to system calls, which is useful in the case where a dialogue agent placed as the front-end interface to another software system. Similar to the *Pre-processing* module, this component includes one or multiple post-processing operations, which process the output in parallel or in sequence, depending on the specification of the designer. In addition, these post-processing operations within the *Post-processing* component can also query the *Domain Knowledge* component for relevant data required for the generation of text response.

3.1.6 Output

Through the output component, the architecture provides a generic way to output the response to appropriate audience(s) depending on the use case. Currently, implemented options are *command line, file based, web based*, and *database*. Similar to the *Input* component, the output component provides flexibility for the architect to change the destination of produced outputs and to separate the output programming logic from that of other components.

3.1.7 Pubsub system/Listeners

In addition to the main pipeline presented above, the proposed system also includes a passive pubsub layer to facilitate monitoring, conversation recording, and independent evaluation of the model. This pubsub system allows the architect to choose or plug in a wide range of peripheral components (called *Listeners*) to passively monitor the main system for execution behaviors and performance. On top of several default channels (see Operation modes section below) that the system writes to and reads from, users can freely add their own channels to communicate between the main system and the pubsub layer hosting the peripherals.

Listeners, as previously mentioned, are optional modules that can be plugged in to passively monitor the system over different channels. These modules are useful when the architect is interested in observing the system inputs and/or outputs, or visualizing internal parameters or states of the dialogue model at execution time. Passive monitoring logic can be independently introduced into the system without modifying the other components' implementations.

3.2 Operation modes

MACA can be operated in three different modes: *Data Collection*, *Training* and *Execution*. This section describes the data flow in the architecture along with abstract setups of the framework's components in these different operation modes for several dialogue models from the recent literature.

3.2.1 Data Collection Mode

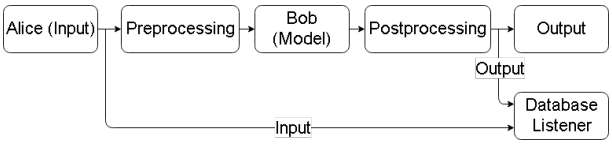

Figure 2: Data flow in data collection mode.

The goal of the data collection mode is to collect conversations as training datasets for dialogue models. In this mode, the two agents *Alice* and *Bob* involved in the conversation are considered the *Input* component and the *Dialogue Model* component respectively. Figure 2 describes a typical setup for the data collection process with said configuration. The conversation is recorded using a database listener that receives both input (context) and output (response) for each speaking turn, similar to the scheme presented in section 3.2.3 above.

This setup realizes the infrastructure required for two common dialogue data collection scenarios. The first scenario is collection of both contexts and responses. In this case, both agents are humans. In the second scenario, the goal is to collect human responses for a given set of contexts. In this case, agent Alice can be an implementation

of the *Input* component fetching contexts from a database, while Bob is a human agent responding to the fetched contexts.

3.2.2 Training Mode

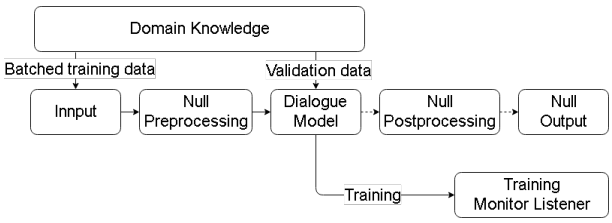

Figure 3: Data flow in training mode.

The goal of the training and validation mode is to use the data obtained in the data collection stage to train one or multiple dialogue models, as illustrated in figure 3. Assuming a dataset is available from the *Domain Knowledge* component, training data can be fetched as batches by the *Input* component and fed into the *VoidPreprocessing* component. This component simply forwards the data as is to the *Dialogue Model* component, which performs model training, and occasionally queries the domain knowledge for validation data to verify its training progress. Since system output is irrelevant within the training scenario, *Post-processing* and *Output* components are implemented with null operations, which simply discard their received contents. Once certain validation accuracy is achieved, the model can save its internals on to the disk and terminate the system. In addition to the core training process, the architect may opt to emit training information to a listener through the *training* channel to monitor the training progress.

3.2.3 Execution Mode

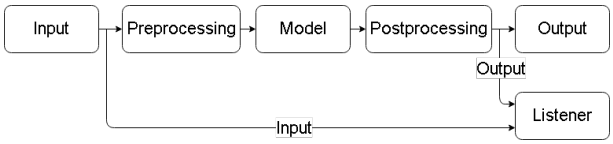

Figure 4: Data flow in execution mode.

Data flow in execution mode is illustrated in figure 4. In this mode, all core components in the system are enabled and active. Given that the dialogue model has been successfully trained and fine-tuned, its internal states (e.g. weights, hyperparameters) are loaded into the *Dialogue Model* component at system initialization time. Input data is retrieved in real time (through local user interface (e.g. terminal, GUI) or via an interface with the Internet (e.g. web page, chat client)). This input then enters the pipeline and goes through *Preprocessing*, *Dialogue model*, *Postprocessing* and finally *Output* component. At the end of the pipeline, the output component is responsible for sending the generated responses to relevant audiences (e.g. print to stdout, HTTP response, ...).

From the peripheral components perspective, conversation logging and system monitoring can be done through two default channels: *input* and *output*. Specifically, as shown in figure 4, the passive *listener* receives a notification for every input received from the *Input* component on the *input* channel, and a notification for every output received by the *Output* component on the *output* channel.

4 Feature Highlights

As discussed in the previous sections, MACA can be used to plug in different types of existing dialogue agents. The architecture abstracts the implementation details, similar to popular machine learning libraries such as Theano (Theano Development Team, 2016), Tensorflow (Abadi et al., 2016), or PyTorch. The modular design enables rapid prototyping and should facilitate reproducing previous results. The support for experimentation, extension, and development of slot-based dialogue agents for goal-oriented tasks has also been provided. In addition, the current implementation has rule-based approach for slot disambiguation and has provisions for the easy extension of slot disambiguation to *machine learning* (ML) based modules. The clear separation of domain knowledge from the agent aids in multi-agent systems with little dependence on the domain – the intent identification is provided at a higher level to identify and trigger the task, defined as a set of slots and *ask queries*. Intent identification supports hosting of multiple tasks.

The framework provides tools for easy hosting of dialogue tasks as HIT (Human Intelligence Task) on Amazon mTurk to collect human responses; the framework also supports modelling dialogue tasks as an agent-agent interaction that can be used to test a dialogue agent against simulated users (Schatzmann et al., 2005b). A summary of MACA's features is provided in Table 1.

	MACA	TCP	Ravenclaw
Multi Domain Support	✓	✓	✓
Plug-and-Play	✓	✓	✗
Adaptation for FCA	✓	✗	✗
Agent Abstraction	✓	✗	✗
Integration with mTurk	✓	✗	✗

Table 1: Feature Comparison of MACA with existing similar frameworks. Note: FCA: Frequently used Conversational Agents.

5 Implementation Highlights[2]

MACA's current implementation is in Python and includes standard libraries to ensure the framework's portability, as well as to facilitate rapid prototyping of different dialogue model strategies. Each component of the framework (e.g. *Input* component) is described with an abstract Python class, whose concrete implementation instances (i.e. Python objects) are manifestations of that component (e.g. Command line input, Database input). This corresponds to the abstraction layer of the architecture's module to foster independence of the pipeline implementation from that of the underlying dialogue model(s). The assembly of these components are then specified in a central configuration file representing an instantiation of the architecture. With this design, changes in the instantiation specifications can be done within the central configuration file by modifying the names of invoked modules. On the other hand, this setup allows system specifications to be completely contained within the central configuration file, which reduces maintenance effort and simplifies configuration modification during development. In addition, the open source nature of the framework encourages sharing and reusing of components, which allows researchers to easily develop from existing models and save time by reusing common components written by others.

6 Case Studies

MACA was deployed for several studies within our research group. All conducted studies have the same template for the central configuration file, whose content is then modified corresponding to the purpose of each study. Listing 1 shows the configuration template representing a system with a simple dialogue agent, which repeats its input (echo agent). The configuration file requires several attributes to be mentioned and provides a general outlook of the experiment being run. The template contains the following attributes: *input, output, preprocessing, postprocessing, agent, domain knowledge* and *listeners*. The *class* sub-attribute of the attributes refers to the Python class implementation of the component being invoked.

6.1 Building a simple agent

The Echo agent is designed to simply listen and store the input to file; this is a good first test case for new users of MACA. In this setup, the *input* attribute is instantiated with *StdinInputDevice*, which is the commandline inputs, and the *output* attribute is instantiated with *FileOutputDevice*, which writes the results to a file. Likewise, the instantiations of the other attributes, like *postprocessing*, *preprocessing* and *domain_knowledge*, point to *VoidPostprocessor*, *VoidPreprocessor*, and *EmptyDomainKnowledge* respectively, since Echo agent does not require them. The *agent* attribute is instantiated with the appropriate dialogue agent, which in this case is Echo agent. Along with these components, *LoggingListener*, which logs the input and output of the system on to an output file, is included as a *listener* component.

```
'input' : { 'class' : StdinInputDevice },
'output' : {
  'class' : FileOutputDevice,
  'args' : ['out.gods']
},
'preprocessing' : {
  'modules' : [{ 'class' : VoidPreprocessor, }],
  'parallel' : False, # Optional
},
'postprocessing' : {
  'output_index' : 0, # Index of the pipe to output
  'parallel' : False, # Optional
  'modules' : [ { 'class' : VoidPostprocessor, } ]
},
'agent' : { 'class' : EchoAgent },
'domain_knowledge' : { 'class' : EmptyDomainKnowledge },
'listeners' : { 'unnamed': [{ 'class' : LoggingListener }] }
```

Listing 1: Configuration Template.

6.2 Building a goal oriented system

Next, we consider using MACA to build goal oriented agents for the restaurant, flight booking, and other toy domains. These slot-based agents were developed using the tools provided in the framework that aids in hierarchical task decomposition and slot sharing across tasks (as in the example reusing the same Python variables). With regard to hosting a multi-task agent, the invocation of Goal

[2]Some of the configuration file samples provided in the listings in this section are slightly modified to fit the page limit constraint.

oriented policies/sub-agents for each task happens with the description of slots – askQuery, disambiguation strategy etc. As with providing multi-agent support, the architecture can handle multiple intents with intent triggers defined for each of them. For example, "I would like to book a flight" will trigger the flight booking policy which will fill in slots specific to this task based on the information provided in the domain knowledge, whereas "What's a good restaurant nearby?" will trigger the restaurant booking policy. The configuration file modification in the agent and domain knowledge attributes is provided in Listing 2.

```
first_name_slot = Slot('first_name')
last_name_slot = Slot('last_name')
'agent' : {
  'class' : PersonalInformationAskingModel,
  'kwargs' : {
    'intents' : [
      AddressAskingAgent('address'),
      NameAskingAgent('name')
    ]
  }
},
'domain_knowledge' : {
  'class' : GoalOrientedDomainKnowledge,
  'args' : [{
    'address' : [
        first_name_slot, last_name_slot,
        Slot('street', ['apt', 'street_name']),
        Slot('city'),
        Slot('country'),
        Slot('zip_code', enabling_condition = \
        lambda slots: slots['country'].value() == "US")
    ],
    'flight_booking' : [
        first_name_slot, last_name_slot,
        Slot('origin'),
        Slot('destination'),
        Slot('return_date')
    ]
  }]
},
```

Listing 2: Sample Agent attribute in Goal Oriented Dialogue models' Configuration.

An overview of the architecture components in the goal oriented setting is provided in Table 2.

6.3 Building a neural response generation agent

We also used MACA to prototype neural response generation agents based on the Hierarchical Encoder-Decoder framework (Serban et al., 2015).

6.3.1 HRED in training mode

MACA's *training* mode was tested with the training process of an HRED agent. The modifications for the central configuration files for this setup are presented in Listing 3. *HREDTrainingInputDevice* simply invokes the training process by sending an *initiate* message to the model while the dialogue model *HREDAgent*, configured to be in *training* mode, starts its regular training process and writes the trained weights to disk. The training dataset is specified using the *prototype* sub-attribute (in compliance with the HRED code base) within the *train_args* attribute of *agent*. All other components of the pipeline are unchanged as it is unnecessary to postprocess or to output data. The HRED agent was trained using both the Twitter Corpus (Ritter et al., 2011) and Ubuntu Dialogue Corpus (Lowe et al., 2015b).

Component		Description	Note
Domain Knowledge		GoalOriented DomainKnowledge	Specifying slots information for known domains.
Input		StdInputDevice	Inputs from stdin.
Preprocessing		VoidPreprocessor	None.
Model	Preprocessing	VoidProcessing	None.
	Postprocessing	Model specific	None.
	Internal	PersonalInformationAskingModel	Intent disambiguation and execution policies.
Postprocessing		VoidProcessing	None.
Output		FileOutputDevice	Output to a file.
Listeners		LoggingListener	Log all pubsub notifications to file.

Table 2: Setup for goal oriented system in execution mode.

```
'input' : {
  'class' : HREDTrainingInputDevice
}, ...
'agent' : {
  'class' : HREDAgent,
  'kwargs' : {
    'train_args' : { 'prototype' : 'ubuntu_HRED' },
    'mode' : system_modes.TRAINING,
  }
},
```

Listing 3: Modified attributes for HRED training.

6.3.2 HRED in execution mode

We also tested using a trained HRED agent in *execution* and *data collection* modes. In the execution mode, MACA used the command-line as the input and the output units to fetch user responses and show model responses from HRED. In the data collection mode, MACA was hosted on a local psiTurk (Gureckis et al., 2016) server emulating mTurk. A layout that lets the users chat and score the model responses was provided, and user inputs were logged by a database listener through the pubsub architecture. In this scenario, the pre-

trained HRED model can be seen as a case of custom built dialogue agent adapted to MACA.

```
1  'agent' : {
2    'class' : HREDAgent,
3    'kwargs' : {
4      'ignore_unknown_words' : True,
5      'normalize' : False,
6      'prototype' : 'prototype_twitter_HRED',
7      'train_dialogues' : 'Training.dialogues.pkl',
8      'test_dialogues' : 'Test.dialogues.pkl',
9      'valid_dialogues' : 'Validation.dialogues.pkl',
10     'dictionary_path' : 'Dataset.dict.pkl',
11     'model_prefix' : './334.74_Model'
12   }
13 },
```

Listing 4: Agent attribute in HRED Configuration.

The central configuration file from Listing 1 is updated for HRED in *execution* mode, as shown in Listing 4. The model specific arguments, provided between lines 3 and 14, in Listing 4 demonstrate MACA's support for plugging in customized or pre-trained dialogue agents. Furthermore, an overview of the architecture, with the instantiated components, and their roles is provided in Table 3.

Component		**Description**	**Role**
Domain Knowledge		EmptyDomainKnowledge	An empty domain.
Input		StdInputDevice	Inputs from stdin.
Preprocessing		HredPreprocessing	Tokenize input sentence.
Model	Preprocessing	Model specific	Add model specific tokens.
	Postprocessing	Model specific	Remove speaker tokens.
	Internal	HredAgent	HRED internals.
Postprocessing		VoidProcessing	None.
Output		FileOutputDevice	Output to a file.
Listeners		LoggingListener	Log all pubsub notifications to file.

Table 3: Setup of HRED system: Execution mode.

6.4 Building a neural response retrieval agent

Finally, we built an architecture that incorporates a neural response retrieval agent operating using the Dual Encoder method (Lowe et al., 2015a).

6.4.1 Dual Encoder in training mode

Listing 5 presents changes to the template configuration to incorporate a Dual Encoder dialogue agent in training mode. Similar to the HRED model training case, we replace the *Input* and *Model* modules in the template configuration. In the case of Dual Encoder, the specified data set will be loaded into *DomainKnowledge* and will become accessible after initialization. During the training process, *RetrievalModelTrainingInputDevice* retrieves the data from the specified training data set via *DomainKnowledge* and feeds it to the *Dialogue Model* while the *RetrievalModelAgent* contains the relevant training parameters. Once training finishes, *RetrievalModelTrainingInputDevice* issues a message to the agent to write out trained weights to disk.

```
1  'input' : {
2    'class' : RetrievalModelTrainingInputDevice,
3    'kwargs' : { 'n_epochs' : 500, 'shuffle_batch' : False }
4  }, ...
5  'agent' : {
6    'class' : RetrievalModelAgent,
7    'args' : [ 'twitter_dataset/W_twitter_bpe.pkl' ],
8    'kwargs' : {
9      'model_fname' : 'model.pkl',
10     'mode' : system_modes.TRAINING,
11     'model_params' : {
12       'encoder' : 'lstm',
13       'batch_size' : 512, 'hidden_size' : 200,
14       'optimizer' : 'adam', 'lr' : 0.001,
15     }
16   }
17 }, ...
18 'dataset' : {
19   'class' : RetrievalTwitterDataset,
20   'args' : [ 'twitter_dataset', 'dataset_twitter_bpe.pkl' ]
21 },
```

Listing 5: Modified attributes for Dual Encoder training.

6.4.2 Dual Encoder in execution mode

We also tested the Dual Encoder agent in *execution* mode, which is an instance of adapting a retrieval based model to the proposed framework. The execution mode in this case obtained inputs from a database of previously collected context-response pairs. The configuration file for the Dual Encoder model looks mostly similar to the generic template, with modification on the *agent* attribute, described in Listing 6.

```
1  'preprocessing' : {
2    'modules': [{
3      'class' : RetrievalModelPreprocessor,
4      'args' : ['./retrieval/BPE/Twitter_Codes_5000.txt']
5    }],
6  }, ...
7  'agent' : {
8    'class' : RetrievalModelAgent,
9    'args' : [ '../../twitter_dataset/W_twitter_bpe.pkl' ],
10   'kwargs' : {
11     'model_params' : {
12       'encoder' : 'lstm',
13       'batch_size' : 512, 'hidden_size' : 100,
14       'input_dir' : '../../twitter_dataset',
15       'W_fname' : 'W_twitter_bpe.pkl'
16     }
17   }
18 },
```

Listing 6: Agent attribute in Dual Encoder (Retrieval Model) Configuration.

The configuration file's flexibility allows customized agents to be plugged in with ease, while providing the parameters for the model to run in the *model_params* sub-attribute. Further, an overview of MACA with its instantiated components and their roles is provided in Table 4; specification of these attributes within MACA is achieved through the configuration file.

Component		Description	Role
Domain Knowledge		EmptyDomainKnowledge	An empty domain.
Input		StdInputDevice	Inputs from stdin.
Preprocessing		RetrievalModelPreprocessing	Compute BPE on all utterances.
Model	Preprocessing	Model specific	None.
	Postprocessing	Model specific	None.
	Internal	RetrievalModelAgent	Dual Encoder internals.
Postprocessing		VoidProcessing	None.
Output		FileOutputDevice	Output to a file.
Listeners		LoggingListener	Log all pubsub notifications to file.

Table 4: Setup for Dual Encoder system in execution mode.

7 Discussion

MACA offers a unified architecture for dialogue agents that supports the plug-n-play of different types of dialogue agents and different domains. We hope that this will facilitate the fast development of new models, but also foster reproducibility in dialogue system research.

A few possible limitations in the current implementation of MACA include simplicity of the pubsub system, lack of support for distributed hosting of different components of the architecture, and lack of support for parallel conversations. As future work, the pubsub system could be improved by capturing a wider range of system information with more monitoring pubsub channels. In addition, we plan to incorporate new domains and agents as they become available, along with comprehensive ML based slot-disambiguation modules.

Acknowledgments

Acknowledgements The authors gratefully acknowledge financial support for this work by the Samsung Advanced Institute of Technology (SAIT) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S Corrado, A. Davis, J. Dean, and M. et. al Devin. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467* .

J. F Allen, D. K Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent. 2001. Toward conversational human-computer interaction. *AI magazine* .

D. Bohus and A. I Rudnicky. 2003. Ravenclaw: Dialog management using hierarchical task decomposition and an expectation agenda .

PA Crook, A Marin, V Agarwal, K Aggarwal, T Anastasakos, R Bikkula, D Boies, A Celikyilmaz, S Chandramohan, and Z et. al Feizollahi. 2016. Task completion platform: A self-serve multi-domain goal oriented dialogue platform. *NAACL HLT* .

P. Gage. 1994. A new algorithm for data compression. *The C Users Journal* .

T. M. Gureckis, J. Martin, J. McDonnell, A. S. Rich, D. Markant, A. Coenen, D. Halpern, J. B. Hamrick, and P. Chan. 2016. psiturk: An open-source framework for conducting replicable behavioral experiments online. *Behavior research methods* .

S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural computation* .

S. Kim and R. E. Banchs. 2014. R-cube: a dialogue agent for restaurant recommendation and reservation. In *Asia-Pacific Signal and Information Processing Association, 2014 Annual Summit and Conference (APSIPA)*. IEEE.

Q. V. Le and T. Mikolov. 2014. Distributed representations of sentences and documents. In *ICML*.

E. Loper and S. Bird. 2002. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*.

R. Lowe, N. Pow, I. Serban, and J. Pineau. 2015a. The ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. *arXiv:1506.08909* .

R. Lowe, N. Pow, I. Serban, and J. Pineau. 2015b. The ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. In *16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. Efficient estimation of word representations in vector space. *arXiv:1301.3781* .

S. Png and J. Pineau. 2011. Bayesian reinforcement learning for pomdp-based dialogue systems. In *Acoustics, Speech and Signal Processing (ICASSP), IEEE International Conference on*. IEEE.

A. Ritter, C. Cherry, and W. B. Dolan. 2011. Data-driven response generation in social media. In *EMNLP*.

J. Schatzmann, K. Georgila, and S. Young. 2005a. Quantitative evaluation of user simulation techniques for spoken dialogue systems. In *6th SIGdial Workshop on DISCOURSE and DIALOGUE*.

J. Schatzmann, K. Georgila, and S. Young. 2005b. Quantitative evaluation of user simulation techniques for spoken dialogue systems. In *6th SIGdial Workshop on DISCOURSE and DIALOGUE*.

F. Seide and S. McDirmid. 2012. Clippyscript: A programming language for multi-domain dialogue systems. In *Thirteenth Annual Conference of the International Speech Communication Association*.

I. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau. 2015. Building end-to-end dialogue systems using generative hierarchical neural network models. *arXiv:1507.04808* .

I. Serban, A. Sordoni, R. Lowe, L. Charlin, J. Pineau, A. Courville, and Y. Bengio. 2016. A hierarchical latent variable encoder-decoder model for generating dialogues. *arXiv:1605.06069* .

Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv:1605.02688* .

S. Young. 2006. Using pomdps for dialog management. In *Spoken Language Technology Workshop*. IEEE.

T. Zhao, K. Lee, and M. Eskenazi. 2016. Dialport: Connecting the spoken dialog research community to real user data. *arXiv:1606.02562* .