Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science Architecture of Information Systems Research Group

# Using End to End Memory Networks for task oriented dialog systems

*Data Mining Seminar*

Alessandro Terragni

Supervisors: Mykola Pechenizkiy

version 1

Eindhoven, December 2017

# Contents

# Chapter 1

# Introduction

Conversational Agents or Dialog Systems, nowadays, are one of the artificial intelligence cutting edge technologies.
The services offered vary a lot: ranging from business to entertainment; they can be self standing or integrated in all the most famous messaging platforms like Facebook Messenger, Telegram and Slack.
In particular, there are two main types of Dialog Systems: task oriented dialog agents, which help users to achieve a pre-defined task, and conversational bots (commonly called chatbots), which aim is to mimic natural unstructured conversations.
In the last few years, a lot of encouraging results have been obtained in conversational bots, mainly thanks of deep learning, that makes possible to train these models in an end to end way, using only recorded dialogues and plain texts as inputs for the training algorithm.
On the other hand, improvements in goal oriented dialog systems got stuck: they still required a lot of domain specific handcrafted rules, limiting scalability and extension to new domains.
The aim of this study is to transfer the knowledge acquired in end to end conversational bots into a goal oriented scenario, using End to End Memory Networks [6] over an artificial generated data-set.
This data-set has been generated automatically combining natural languages patterns with the knowledge basics from a soccer training recommendation app. Then, an end to end memory network has been trained and tested achieving promising results, described in chapter 5.

This document will follow this structure: first the problem and the domain will be defined more deeply, then a literature analysis will be performed and finally, the implementation and the results obtained will be discussed.

# Chapter 2

# Problem Definition

This work tries to answer this question:

*is it possible to use end to end memory networks to train a task oriented dialog system using only a data-set generated from the knowledge basics ?*

## 2.1 Domain

The experiments will be carried out in the context of a soccer training recommendations app: Coach assist (https://www.thegreatmatch.com/de-coachassist-app/).
When preparing a training session, an experienced coach has a training objective in mind: e.g. improve the ball recovery tactics, defensive work or passing skills.
This app offers several training exercises for each training object, each one with different characteristics:

- **Title**: plain text
- **Main Focus**: Attack, Defense, Tactics, Techniques, Passing ...
- **Age**: Any, Under 9, Under 11, Under 13 ...
- **Phase**: Warm-up, Main, Cool-down, shot on goal ...
- **Group Size**: Individual, Group, Team
- **Goalkeeper**: 0, 1, 2, 3 ...

A coach can search through this exercise database and select the items that are suitable using filtering conditions.

The aim is to create a bot through which the coach can informally state his training objective like in Figure 2.1: at the end, the interaction with the bot should result in a recommendation of an appropriate exercise for the coach.
The task oriented dialogue agent should be able to conduct full dialogues in the depicted scenario, issuing api calls and displaying available options as in Figure 2.1

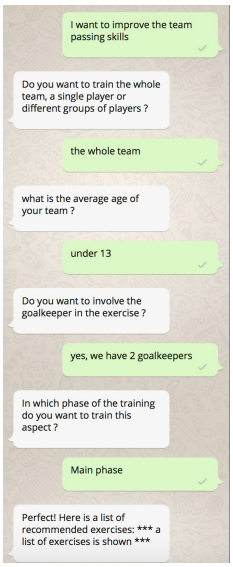


Figure 2.1: Example of soccer bot conversation

## 2.2 Approach

One of the main problem that has hindered the application of end to end models to a task oriented scenario is the shortage of data-sets: while conversational agents can be trained on corpus available from forums, newspapers, movie dialogues and so on, task oriented agents need dialogues labelled with the corresponding api calls.
In the example of Figure 2.1 the dialogue will be labelled, for example, with an api call that looks like this:

recommendTrainingProgram{MainFocus = 'passages', Age = 'U13', Phase='Main Phase', GroupSize='Team', Goalkeeper = 2}

Collecting this type of dialogues with the corresponding labels is of course very hard and limits the applicability of end to end models to the task oriented domain.
This brings to the need of finding a way to generate these dialogues automatically from the knowledge basics: I will use the elements of the data-set (the exercises available in the soccer app) combined with natural language patterns, in order to create a set of artificial dialogues labelled with api calls, and then train the End to End Memory Network algorithm over this artificial data-set, trying to create a method that allows easy expansion to different domains.
This approach has been already used by Bordes et al [1] to generate a data-set in the restaurant booking domain, achieving promising results (it will be described more deeply in paragraph 3.2.1.2).
My implementation will be described in chapter 4.

# Chapter 3

# Literature analysis

As already stated in the introduction, there are two types of Conversational Agents: conversational bots and task oriented dialog agents.

## 3.1 Conversational bots taxonomy [5, Chapter 29]

They are systems designed for extended conversations, set up to mimic the unstructured conversational or chats characteristic of human-human interaction, rather than focused on a particular task. An example of such type of conversation is shown in Figure 3.1.
Note that the word chatbot is often used in the media and in industry as a synonym for conversational agent. In this paper the common usage in the natural language processing community will be followed.

| | |
|---|---|
| User: | Do you like Doctor Who? |
| Cleverbot: | Yes, so funny. |
| User: | What's your favorite show? |
| Cleverbot: | I don't have one. |
| User: | What's your name? |
| Cleverbot: | Jason I already told you. |
| User: | No your real name. |
| Cleverbot: | Stephanie IS my real name. |
| User: | I thought it was Jason? |
| Cleverbot: | How did you come to that conclusion. |
| User: | Because you said so. |
| Cleverbot: | I have to do my homework. |

Figure 3.1: Example of a conversation between a user and Cleverbot (http://www.cleverbot.com), an IR-based chatbot

### 3.1.1 Rule based chatbots

Chatbots have evolved trough time, starting from rule based systems: ELIZA [8] was the first chatbot ever made; it was designed in 1966 to simulate a Rogerian psychologist, using a predefined sets of rules. An example of a conversation with Eliza is shown in Figure 3.2

```
Men are all alike.
IN WHAT WAY
They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE
Well  my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
```

Figure 3.2: Example of a conversation with ELIZA

### 3.1.2 Corpus based chatbots

Instead of using hand built rules, they learn patterns from human conversations or sentences in corpora of dialog and non-dialog text.
There are two types of corpus based chatbots: Information Retrieval (IR) based chatbots and Generative chatbots.
IR chatbots respond to a user question by looking at the most appropriate answer from a corpus of natural human text using information retrieval algorithms.
Generative chatbots use machine learning from a corpus to transduce a question to an answer.
These models generate new responses from scratch, using machine translation techniques, but instead of translating from one language to another, they translate from an input to an output response.

### 3.1.3 Deep Learning in Generative chatbots

Deep Learning techniques can be used for both IR or Generative models, but research seems to be moving into the generative direction.
A lot of different models have been developed, in this paper only Memory Networks and their developments will be described, because, as it will be discussed in paragraph 3.2.1, their application to task oriented environments has shown pretty good results.

#### 3.1.3.1 Memory Networks [9]

Memory networks are a new class of learning models developed by Facebook AI Research department in 2015.
Memory networks combine machine learning learning strategies with a long-term memory component that can be read and written to.
This model was developed because most machine learning models lack memory capabilities to effectively remember facts from the past.
Recurrent Neural Networks [7], for example, struggle in answering a question after reading a series of fact: their memory, encoded by hidden states and weights, is too small to remember facts from the past.

The overall structure of the memory networks, depicted in Figure 3.3, consists of:

- A memory $m$: an array of objects $m_i$, for example an array of vectors or strings.
- Four learned components:
  - $I$ ( input feature map ): converts the incoming input to the internal feature representation.
  - $G$ ( generalization ): updates old memories given the new input.
  - $O$ ( output feature map ): produces a new output (in the feature representation space), given the new input and the current memory state.
  - $R$ ( response ): converts the output into the response format desired. For example, a textual response or an action.

The flow of the model works as follow: an input x is converted into an internal feature representation $I(x)$, then memories are updated given the new input.
Output features $o$ is computed given $I(x)$ and the updated memory, and finally the output feature is decoded by $R$ and given as a final response.
It is important to notice that this architecture is very general: all the four components mentioned above can be learned using almost all the classical machine learning models.
In particular, when these components are neural networks, the memory network implementation is called MenNNs: Memory Neural Networks.

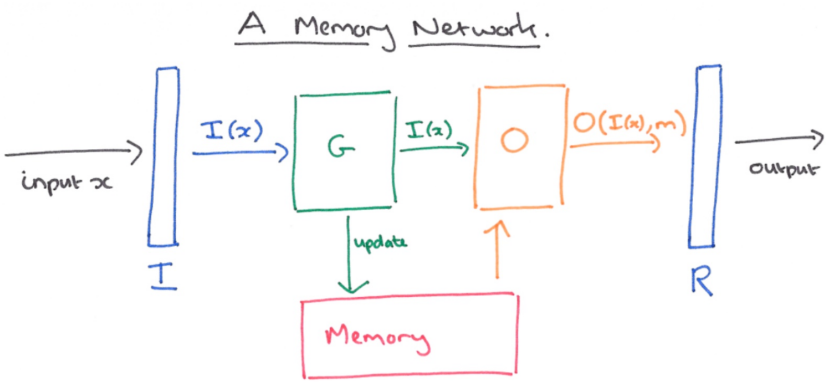


Figure 3.3: Memory network architecture

#### 3.1.3.2 MenNNS for Question Answering Systems

The basic version of MenNNS in a Question Answering Environment, like the one in Figure 3.4, works as follows:

- $I$ takes an input text $x$: it can be a fact or a question.
- $G$ stores the text in an empty memory slot, old memories are not updated.
- $O$ produces output features by finding k supporting memories given x, using max over a scoring function $s_o$.

$$o_1 = O_1(x, m) = argmax_{\mathrm{i}=1,\ldots,\mathrm{N}}(s_o(x, m_i)) \tag{3.1}$$

- $R$ produces the final response: it can simply return the highest scored memory retrieved by O or it can generate a new response using a Recurrent Neural Network.

Joe went to the kitchen. Fred went to the kitchen. Joe picked up the milk.
Joe travelled to the office. Joe left the milk. Joe went to the bathroom.
Where is the milk now? A: office
Where is Joe? A: bathroom
Where was Joe before the office? A: kitchen

Figure 3.4: Example of question answering environment

Comparing MenNns with RNN [7] and LTSM [4], memory neural networks outperform the others, especially in complex tasks.
An example of of this architecture in action is showed in Figure 3.5
The problem of memory networks is that they are not easy to train via back propagation because they require explicit supervision of attention at each layer of the network.
To solve this problem, end to end memory networks have been developed.

#### 3.1.3.3 End to end memory networks [6]

This particularly type of memory network can be trained end to end from input-output pairs and so it is applicable to more tasks.
The model takes a discrete set of inputs $x_1$, ..., $x_n$ that are to be stored in the memory, a query $q$, and outputs an answer $a$.
Each of the $x_i$, $q$, and $a$ contains symbols coming from a dictionary with V words.
The model writes all $x$ to the memory up to a fixed buffer size, and then finds a continuous representation for the $x$ and $q$.

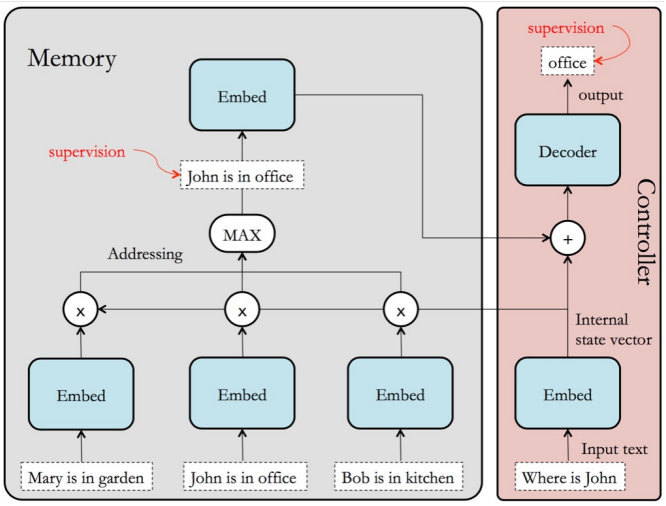


Figure 3.5: MenNNS Example Architecture

The continuous representation is then processed via multiple hops ( computational steps that correspond to lookups on memory ) to output a.

In Figure 3.6 the single layer architecture (a) and multiple layers architecture (b) are showed.

**Single layer architecture**

- **Input memory representation:** given an input set $x_1$, ..., $x_i$ to be stored in memory, they are converted into memory vectors $m_i$ of dimension d, using an embedding matrix A ( dim d x V). The query q is also embedded, using an embedded matrix B (of the same dimension of A ), to obtain an internal state u.
  Thus, the similarity between u and each memory is computed by taking the inner product followed by a softmax function:

$$p_i = Softmax(u^T m_i) \tag{3.2}$$

  where p is a probability vector over the inputs.

- **Output memory representation:** Each $x_i$ has a corresponding output vector $c_i$ (given in the simplest case by another embedding matrix C). The response vector o from the memory is then a sum over the transformed inputs $c_i$, weighted by the probability vector from the input:

$$o = \sum_i p_i c_i \tag{3.3}$$

  Because the function from input to output is smooth, we can easily compute gradients and back- propagate through it.

- **Generating the final prediction:** In the single layer case, the sum of the output vector o and the input embedding u is then passed through a final weight matrix W (of size $V$x$d$) and a softmax function to produce the predicted label:

$$\hat{a} = Softmax(W(o + u)) \tag{3.4}$$

  During training, all three embedding matrices A, B and C, as well as W are jointly learned by minimizing a standard cross-entropy loss between $\hat{a}$ and the true label a.
  Training is performed using stochastic gradient descent.

**Multiple layers architecture**
As shown in Figure 3.6 (b), the single layers can be stacked to handle K hop operations:

- The input to layers above the first is the sum of the output $o^k$ and the input $u^k$ from layer k

$$u^{k+1} = H * u^k + o^k \tag{3.5}$$

  H is linear mapping learned along with the rest of the parameters.

- Each layer has its own embedding matrices $A_k$,$C_k$, used to embed the inputs $x_i$.

- To reduce the complexity of the training phase, embedded matrixes are constrained. The most promising way of constraining is the layer wise method: input an output embedding matrices are the same across different layers.

It is interesting to notice that the three-layer version of the memory model shown is similar to the Memory Network model, except that the hard max operations within each layer have been replaced with a continuous weighting from the softmax.

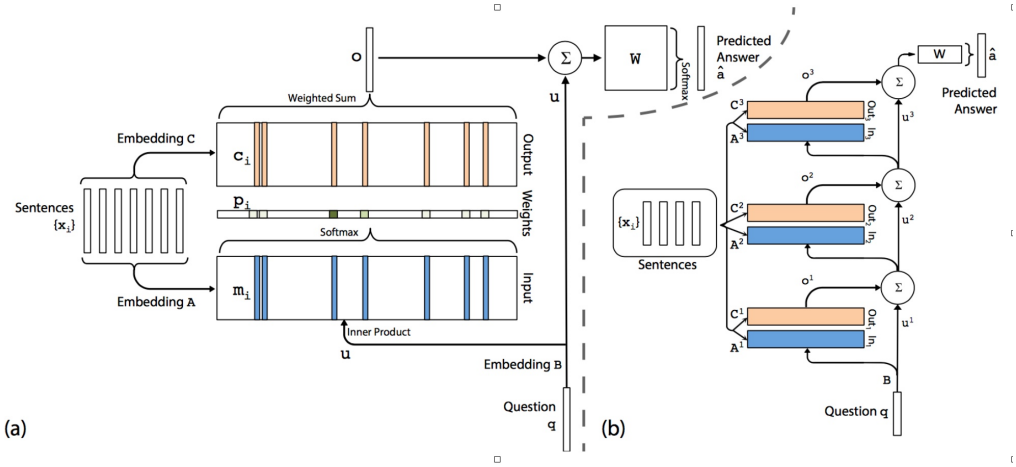


Figure 3.6: End to end memory network architecture: single layer(a), multiple layers(b)

## 3.2 Task oriented dialog agents [5, Chapter 29]

They are designed for a specific purpose, for example to book a flight or a table in a restaurant. They are highly domain specific and need to be handcrafted on the specific problem they are suppose to solve: usually, they are integrated in a bigger system, issuing api calls.
Even if various solutions have been developed through the years, the overall architecture of a task oriented dialog agent has remained the same, consisting in three steps:

- **Domain classification:** is the user talking about booking flights, creating an event on the calendar or making a call ?
  Of course this classifier is not needed in single domain applications like a flight booking dialog system.

- **Intent detection:** what general task or goal is the user trying to accomplish? Does wants to book or cancel a flight ?

- **Slot filling:** extract the particular slots and fillers that the user intends the system to understand from their utterance with respect to their intent.

For example, from a user utterance like this one:

*Show me the tables available for 3 tonight at Restaurant da Mario*

a system will build a representation like:

- DOMAIN: Restaurant
- INTENT: Show tables available
- PLACE: Restaurant da Mario
- DATE: Monday ( Today )
- TIME: evening

Also in this case, various techniques have been developed throughout the years.
At the beginning, this problem was solved using hand written rules; an approach that is still used nowadays because of its simplicity and performances.
Unfortunately, this approach is very time consuming and not scalable, thus, other approaches were developed.
At the moment, the most successful goal-oriented dialog systems that model conversation are partially observable Markov decision processes (POMDP) [10]. However, they still require many hand-crafted features for the state and action space representations, which restrict their usage to narrow domains.

End-to-end dialog systems, usually based on neural networks [2], escape such limitations: all their components are directly trained on past dialogues, with no assumption on the domain or dialog state structure, consequently making it easy to automatically scale up to new domains.
Unfortunately, these models face the same problem we have talked introducing memory networks in paragraph 3.1.3.1: they lack memory capabilities.
In fact, when the bot asks questions to fill some empty slot, the answer can be pretty ambiguous. For example, looking at the final answer in the Figure 3.7, without handcrafting rules, how can we detect if the user is talking about the number of people and not about the time ? We need memory.
That is why memory networks can be used effectively in this context.

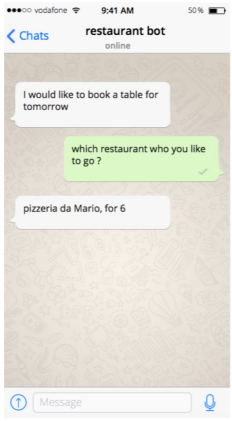


Figure 3.7: Example of answer ambiguity

### 3.2.1 Learning end to end goal oriented dialogues with End to End Memory Networks

In the work of Bordes et al [1], memory networks have been bench-marked against state of the art goal oriented dialog systems baselines. An open source restaurant reservation system database has been generated, and five different tasks have been addressed and tested. The way the dataset is generated is described in section 3.2.1.2.

- **Task 1: Issuing API calls**
  A user request implicitly defines a query that can contain some of the required fields. The bot must ask questions for filling the missing fields and eventually generate the correct corresponding API call.
- **Task 2: Updating API calls**
  Starting by issuing an API call as in Task 1, users then ask to update their requests.
- **Task 3: Displaying options**
  Given a user request, the knowledge basics (KB) is queried using the corresponding API call. The bot must propose options to users by listing the restaurant names sorted by their corresponding rating (from higher to lower) until users accept.
- **Task 4: Providing extra information**
  Given a user request, start the dialog as if users had agreed to book a table in a restaurant, adding all KB facts corresponding to it to the dialog.
- **Task 5: Conducting full dialogues**
  Combines tasks 1-4, generating full dialogues just as in Figure 3.9.
- **Task 6: Using real dialogues**
  Generates full dialogues using a data-set with real human-bot dialogues. Data from DSTC2 [3], that also refers to the restaurant booking domain, is used.

The simulation is based on an underlying KB: whose facts contain the restaurants that can be booked and their properties.
Each restaurant is defined by a type of cuisine (10 choices, e.g., French, Thai), a location (10 choices, e.g., London, Tokyo), a price range (cheap, moderate or expensive) and a rating (from 1 to 8). For simplicity, each restaurant only has availability for a single party size (2, 4, 6 or 8 people). Each restaurant also has an address and a phone number listed in the KB.
The KB can be queried using API calls, which return the list of facts related to the corresponding restaurants. Each query must contain four fields: a location, a type of cuisine, a price range and a party size.

#### 3.2.1.1 Comparison between End to End memory networks and state of the art goal oriented methods

In Figure 3.8 Memory Networks are bench-marked against rules based systems, classical information retrieval models ( TF- IDF Match and Nearest Neighbour ) and supervised embedding models. For tasks T1-T5 results are given in the standard setup and the out-of-vocabulary (OOV) setup, where words (e.g. restaurant names) may not have been seen during training.
Best performing methods (or methods within 0.1% of best performing) are given in bold for the per-response accuracy metric, with the per-dialog accuracy given in parenthesis.
Per-response accuracy counts the percentage of responses that are correct (i.e., the correct candidate is chosen out of all possible candidates).
Per-dialog accuracy counts the percentage of dialogues where every response is correct.
Match type is a particular technique used in the paper to deal with approximate match words and OOV.
As it is clear from the table, Memory Networks (without match type features) outperform classical

| Task | Rule-based Systems | TF-IDF Match no type | TF-IDF Match + type | Nearest Neighbor | Supervised Embeddings | Memory Networks no match type | Memory Networks + match type |
|---|---|---|---|---|---|---|---|
| T1: Issuing API calls | *100 (100)* | 5.6 (0) | 22.4 (0) | 55.1 (0) | **100** (100) | **99.9** (99.6) | **100** (100) |
| T2: Updating API calls | *100 (100)* | 3.4 (0) | 16.4 (0) | 68.3 (0) | 68.4 (0) | **100** (100) | 98.3 (83.9) |
| T3: Displaying options | *100 (100)* | 8.0 (0) | 8.0 (0) | 58.8 (0) | 64.9 (0) | **74.9** (2.0) | **74.9** (0) |
| T4: Providing information | *100 (100)* | 9.5 (0) | 17.8 (0) | 28.6 (0) | 57.2 (0) | 59.5 (3.0) | **100** (100) |
| T5: Full dialogs | *100 (100)* | 4.6 (0) | 8.1 (0) | 57.1 (0) | 75.4 (0) | **96.1** (49.4) | 93.4 (19.7) |
| T1(OOV): Issuing API calls | *100 (100)* | 5.8 (0) | 22.4 (0) | 44.1 (0) | 60.0 (0) | 72.3 (0) | **96.5** (82.7) |
| T2(OOV): Updating API calls | *100 (100)* | 3.5 (0) | 16.8 (0) | 68.3 (0) | 68.3 (0) | 78.9 (0) | **94.5** (48.4) |
| T3(OOV): Displaying options | *100 (100)* | 8.3 (0) | 8.3 (0) | 58.8 (0) | 65.0 (0) | 74.4 (0) | **75.2** (0) |
| T4(OOV): Providing inform. | *100 (100)* | 9.8 (0) | 17.2 (0) | 28.6 (0) | 57.0 (0) | 57.6 (0) | **100** (100) |
| T5(OOV): Full dialogs | *100 (100)* | 4.6 (0) | 9.0 (0) | 48.4 (0) | 58.2 (0) | 65.5 (0) | **77.7** (0) |
| T6: Dialog state tracking 2 | 33.3 (0) | 1.6 (0) | 1.6 (0) | 21.9 (0) | 22.6 (0) | **41.1** (0) | **41.0** (0) |

Figure 3.8: Test result comparison

IR and supervised embeddings across all of the tasks. They can solve the first two tasks (issuing and updating API calls) adequately. On the other tasks, they give improved results, but do not solve them. While the per-response accuracy is improved, the per-dialog accuracy is still close to 0 on T3 and T4.

Unsurprisingly, perfectly coded rule-based systems can solve the simulated tasks T1-T5 perfectly, whereas machine learning methods cannot.
However, it is not easy to build an effective rule-based system when dealing with real language on real problems: that is why the rule based system is outperformed by MemNNs on the more realistic task T6.

#### 3.2.1.2 Data-set generation: dialog bAbI tasks

The dataset can be downloaded at: `https://research.fb.com/downloads/babi/`.
Using the KB, conversations are generated as in Figure 3.9. Each example is a dialog comprising utterances from a user and a bot, as well as API calls and the resulting facts.
Dialogues are generated after creating a user request by sampling an entry for each of the four required fields: e.g. the request in Figure 3.9 is [cuisine: British, location: London, party size: six, price range: expensive].
Natural language patterns are used to create user and bot utterances. There are 43 patterns for the user and 20 for the bot (the user can use up to 4 ways to say something, while the bot always uses the same).
Those patterns are combined with the KB entities to form thousands of different utterances.

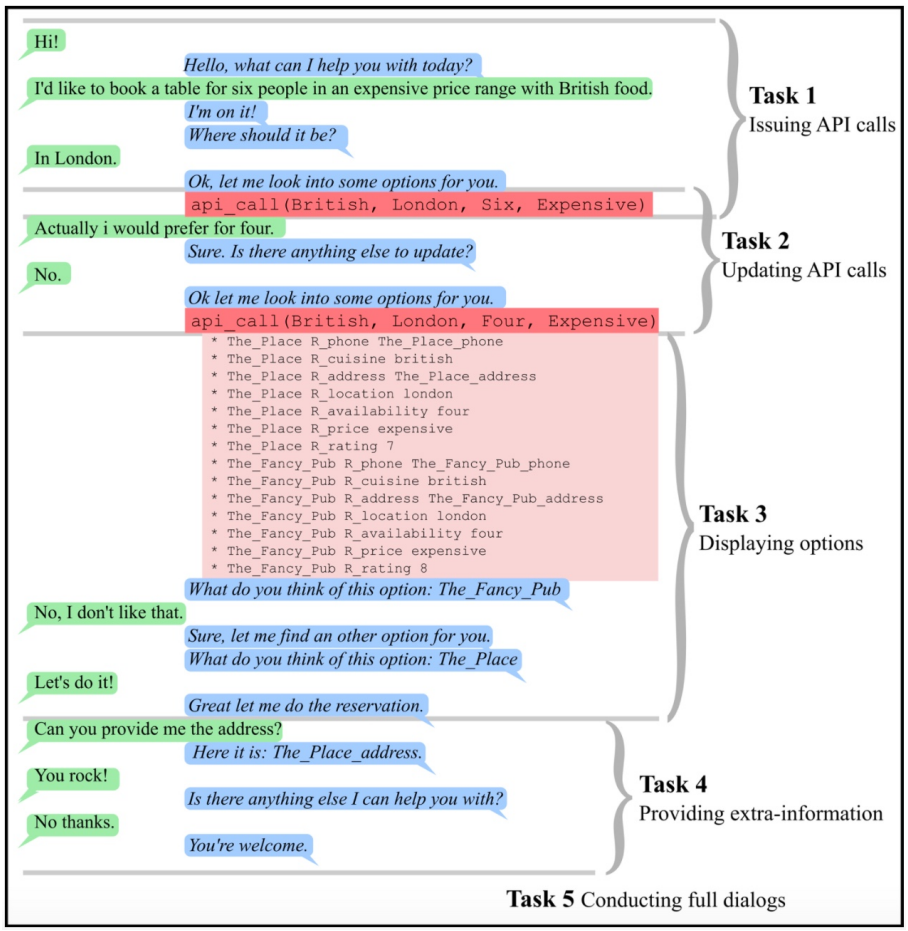


Figure 3.9: Goal oriented conversation example

# Chapter 4

# Implementation

In this chapter, the code implementations will be explained deeply.
All the code can be found in the github page:
https://github.com/alessandroterragni/EndToEndMemoryNetworkBot

## 4.1 Dataset generator

The dataset has been generated using the standards of the Babi Task dataset described in section 3.2.1: dialogues are divided between each other by a blank line. Moreover, each line of the dialogue is built like this:

```
rowLine Question "\t" Answer
```

An example can be seen in Figure 4.1

```
1 is everything fine?   hello!, what can I help you with today?
2 I would like to exercise the Defense  What is the average age of the training group? Under 9, Under 11, Under 13 or
Over13?
3 Under 9       In which phase do you want to train? Warm-up, Main, Cool-down, shot on goal?
4 I want to train during the Warm-up phase      Who you want to train: a single player, a Group or the whole Team?
5 I would like to train the Group       How many goalkeepers will join the training?
6 only 9        apicall:  MAIN_FOCUS = Defense  AGE = Under 9  PHASE = Warm-up  GROUP_SIZE = Group GOAL_KEEPERS = 9

1 hello hello!, what can I help you with today?
2 I want to train a Team on Attack      What is the average age of the training group? Under 9, Under 11, Under 13 or
Over13?
3 Under 11      In which phase do you want to train? Warm-up, Main, Cool-down, shot on goal?
4 I would like to train during the shot on goal phase   How many goalkeepers will join the training?
5 only 4        apicall:  MAIN_FOCUS = Attack  AGE = Under 11  PHASE = shot on goal  GROUP_SIZE = Team GOAL_KEEPERS = 4
```

Figure 4.1: Generated dialogue example

In particular, dialogues have been generate to cover only the first task: "Issuing API calls". A user request implicitly defines a query that can contain some of the required fields. The bot must ask questions for filling the missing fields and eventually generate the correct corresponding API call.

Here follows a brief description of how dialogues have been generated.
First of all, an API call is generated randomly from the knowledge basics described below, then, a dialogue is built using the element of the API call in combination with natural language patterns. Each response is also stored in the candidates file.

**Knowledge basics**:

- main focus = ('Attack', 'Defense', 'Tactics', 'Techniques', 'Passing')
- age = ("Over 13", "Under 9", "Under 11", "Under 13")

- phase = ("Warm-up", "Main", "Cool-down", "shot on goal")
- group size = ("Single player", "Group", "Team")
- goalkeeper = ("0","zero", "1","one", "2","two", "3","three","4","Four", "5","Five","6","six", "7","seven", "8","eight","9","nine")

To have a complete overview of the natural language patterns I used in combination with the knowledge basics, please refer to code in the github page.

Various types of dialogues have been generated:

- **Single intent dialogues**: in these dialogues the user states one item per sentence.
  For example:

```
1 hi    hello!, what can I help you with today?
2 I want to train on Defense    What is the average age of the training group?
3 Under 11    In which phase do you want to train?
4 Cool-down phase   Who you want to train: a single player, a Group or the whole Team?
5 I would like to train the Group    How many goalkeepers will join the training?
6 just 0    apicall:  MAIN_FOCUS = Defense  AGE = Under 11
PHASE = Cool-down  GROUP_SIZE = Group GOAL_KEEPERS = 0
```

- **Double intent dialogue**: in these dialogues two items can be stated in the same sentence.
  For example:

```
1 hi    hello!, what can I help you with today?
2 My Team struggles in Tactics   What is the average age of the training group?
3 Under 9    In which phase do you want to train??
4 During the shot on goal phase   How many goalkeepers will join the training?
5 only 8    apicall:  MAIN_FOCUS = Tactics  AGE = Under 9
PHASE = shot on goal  GROUP_SIZE = Team GOAL_KEEPERS = 8
```

  In this example, the user has stated two objects in one sentence: "My team struggles in Tactics", selecting both the group size and the main focus in just one sentence.

- **Triple intent dialogues** in these dialogues three items can be stated in the same sentence.
  For example:

```
1 hi  hello!, what can I help you with today?
2 I want to improve the Tactics skills of my Over 13 Single player
In which phase do you want to train? Warm-up, Main, Cool-down, shot on goal?
3 During the Warm-up phase   How many goalkeepers will join the training?
4 zero   apicall:  MAIN_FOCUS = Tactics  AGE = Over 13
PHASE = Warm-up  GROUP_SIZE = Single player GOAL_KEEPERS = zero
```

  In this example, the user states: "I want to improve the Tactics skills of my Over 13 Single player", selecting the main focus, the age and the group size in just one sentence.

- **four intent dialogues** in these dialogues four items can be stated in the same sentence.
  For example:

```
1 hello   hello!, what can I help you with today?
2 I want to improve the Defense skills of my Under 11 Single player
during the Cool-down phase
How many goalkeepers will join the training?
3 two   apicall:  MAIN_FOCUS = Defense  AGE = Under 11
PHASE = Cool-down  GROUP_SIZE = Single player GOAL_KEEPERS = two'
```

In this example, the user states: "I want to improve the Defense skills of my Under 11 Single player during the Cool-down phase", selecting the main focus, the age, group size and the phase in just one sentence.

Five datasets have been generated: one for each type of dialogue plus one in which all the types of dialogues are combined.

## 4.2 End to End Memory network

The original Facebook End to End memory networks implementation has been written in Torch7 and in Matlab. All these versions are available at this link `https://github.com/facebook/MemNN`
In this github page, other approved third parties version are present.
Because of my familiarity with Python, I decided to use one of these third party implementations (`https://github.com/domluna/memn2n/blob/master/memn2n/memn2n.py`), that is written using TensorFlow and ScikitLearn.
In particular, I strongly took inspiration from this implementation (`https://github.com/voicy-ai/DialogStateTracking`), that uses the end to end memory network implementation presented above on the Restaurant Babi task dataset, adapting it to my problem and to the dataset I generated.

**Data structures**
All the data structures used in the model are generated from the dataset calling

```
main.py --prep_data
```

These are the data structures used:

- **Data**
  - **Candidates**: array of the candidates: they are all the answers that can be outputted from a question.
  - **Test, Train and Validation**: they are made of dialogues encoded in a particular way, presented below.

    ```
    [(context,tokenized question, answer_id),...]
    ```

    In the context is encoded the story of the dialogue before the related question.
    For example, for this dialogue:
    Q: "Hello"
    A: "Hello, what can I help you with today ?"
    Q: "I want to train my team on attach"
    A: "Perfect, I will look at some exercises for you"

    The context related to the question "Hello" will be empty, because nothing had happened before.
    On the other hand, the context related to the question " I want to train my team on attach" will be:

    ```
    [['hello'],['hello', 'what', 'can', 'i', 'help', 'you', 'with', 'today'] ]
    ```

While the complete structure of the whole dialogue will be:

```
[
   ([ ], ['hello'], 149),
   (
      [['hello'], ['hello', 'what', 'can', 'i', 'help', 'you', 'with', 'today']],
      ['I', 'want', 'to', 'train', 'my', 'team', 'on', 'attach'],
      91
    )
]
```

Where 149 and 91 are, respectively, the ids of the answers: "Hello, what can I help you with today" and "Perfect, I will look at some exercises for you"

- **Metadata**
  - **w2idx**: is a vocabulary of all the words ( word: id )
  - **idx2w**: is a vocabulary of all the words ( id: word )
  - **candid2idx**: is a vocabulary of all the candidates ( candidate: id )
  - **idx2candid**: is a vocabulary of candidates ( id: candidate )

**End to End memory networks attributes:**

- **vocab size**
  The size of the vocabulary, it includes the nil word, which is encoded with 0.
- **sentence size**
  The size of the embedded sentences.
- **embedding size**
  The size of the word embedding.
- **hops**
  How many layers the memory network will have.
- **candidates**
  The vector of the candidates.
- **initializer**
  Weight initializer: it generates tensors with a normal distribution: I used the random normal initializer with 0.1 standard deviation
- **Session**
  Tensorflow Session the model is run with.
- **optimizer**
  Optimizer algorithm used for the stochastic gradient descent: I used the AdamOptimizer.

### 4.2.1 TensorFlow model

The model follows the usual TensorFlow architecture depicted in Figure 4.2

**Phase 1: assemble the graph**

1. **Placeholders** ( a placeholder is a promise to provide a value later ):

- **stories**
- **queries**

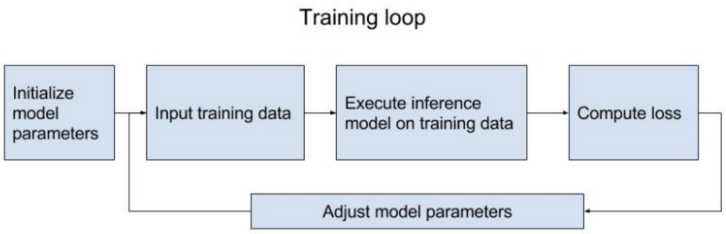


Figure 4.2: TensorFlow Architecture

- **answer**

These placeholders are vectors that will be filled with the respective data structures discussed before.

2. **Variables**:

- **matrix A**
  dimension ( voc size, embedding size)
  Each row corresponds to the representation vector of one word. In particular, A is used both to embed the stories and queries ( instead of using three different matrices as it is stated int he paper ).

- **matrix H**
  dimension ( embedding size, embedding size)
  It is a linear mapping used to update u between hops.

- **matrix W**
  dimension ( voc size, embedding size)
  It is used to embed candidates.

All these variables are initially filled with random weights coming from a normal distribution.

3. **Inference** (compute the forward path of the graph):
Here, a three layer architecture of the End To End memory is built as it is described in Figure 3.6
Queries are embedded with the matrix A using the TensorFlow method tf.nn.embedding_lookup() (To get the vector representation of all the words in our dictionary, we get the slice of all corresponding rows in the embedding matrix).
Then, the internal state $u$ is obtained just summing the value of each rows of A.
From now on, the process iterates trough the 3 layers of the network.
Stories are embedded using matrix A as well, and memories $m_i$ are computed summming the elements of each sentence vector.
Then, formulas of end to end memory networks are simply applied.

4. **Define the loss function**:
The loss function is defined as cross-entropy loss between the predicted answers and the real answers

5. **Define optimizer**:
We use the stochastic gradient descent.

6. **Minimize loss function**:
Cross entropy is minimized in three steps: the gradient of cross entropy loss is computed, then it is modified in order to take in account of the nill variables: the first rows of the matrix A and

W are a vector of all zeros, thus the first rows of the corresponding gradients are overwrited with zero as well. Finally, the operation to minimize these modified gradients is computed.

The End To End Memory Network class exposes two methods:

- `batch_fit(self, stories, queries, answers)`
  This function is used to train the network in batches.
- `predict(self, stories, queries)`
  This function is used to predict answers.

## 4.3 How to run the code

Once you have installed all the dependencies, just run
`python3 main.py --help`
to see all the options available.

To run the code:

1. prepare the data with:

   ```
   python3 main.py --prep_data
   ```

2. train the model with

   ```
   python3 main.py --train
   ```

3. start the conversation environment with

   - `python3 main.py --infer`
     To run it in terminal
   - `python3 app.py`
     To run it in a web app, using Flask

4. start chatting

**Dependencies**

- Python3
- tensorflow
- numpy
- scipy
- sklearn
- Flask
- nltk
- six

# Chapter 5

# Results

The model has been tested against five different datasets:

- **single intent dataset**: filled just with single intent dialogues.
- **double intent dataset**: filled just with double intent dialogues.
- **triple intent dataset**: filled just with triple intent dialogues.
- **quadruple intent dataset**: filled just with quadruple intent dialogues.
- **all intents dataset**: it is a random mix of all the previous datasets.

Refer to paragraph 4.1 to have more information about how they have been generated.

Each dataset, made of 9000 dialogues, has been splitted in three datasets of 3000 dialogues each: training, validation and testing.
The model has been trained iterating trough 200 epochs, taking the best model based on the validation accuracy.
In table 5.1 you can see the accuracy results of the trained models over the various datasets.

| Dataset | Training dialogues | Validation dialogues | Testing dialogues | Accuracy |
|---|---|---|---|---|
| Single intent dataset | 3000 | 3000 | 3000 | 0,861 |
| Double intent dataset | 3000 | 3000 | 3000 | 0,817 |
| Triple intent dataset | 3000 | 3000 | 3000 | 0,787 |
| Quadruple intent dataset | 3000 | 3000 | 3000 | 0,715 |
| Complete intent dataset | 3000 | 3000 | 3000 | 0,8 |

Figure 5.1: Results table

Per-dialog accuracy counts the percentage of dialogues where every response is correct.
As it is clear from the table above, the accuracy of the model decreases with increasing dialogues complexity: in the single intent dataset the accuracy is very high, while in the quadruple intent

one it decreases from 86% to 71%.
In the complete dataset, that randomly mixes all the different intent dialogues, the accuracy increases to 80%, promising good results also on real dialogues.

# Chapter 6

# Conclusions

As the results in chapter 5 showed, End to End memory networks trained over an artificial dataset have been proven to reach quite good results in the task oriented dialog systems scenario.
Unforunately, they decrease their accuracy with more complex dialogues, thus their applicability in real scenario can be still premature.
That is the reason why the model I built need to be tested against a real dialogue dataset, in order to really understand its potential.
In conclusion, the tactic of generating a dataset and train an end to end model on it, has been succesfull, and, if improved, can become a new standard for the task oriented dialog agents scenario.

## 6.1 Further researches

Good results have been achieved, but a lot of work must be done.
It would be interested to test the model on the other Babi Tasks and on a real dialogue dataset, benchmarking it versus other state of the art models.
It would also interesting to enrich the dataset with more complex scenarios, in order to improve the usability of the bot: for example, allowing the user to talk with the bot in a natural way as in Figure 6.1, where the user states the problem and let the bot deduce the training object.

**Coach**: we don't get many scoring chances
**Bot**: are your opponents significantly stronger than your team?
**C**: no; we do get the ball on the opponent's half many times, but then we lose the ball
**...**

Figure 6.1: Example of complex dialogue

.

# Bibliography


[1] Antoine Bordes and Jason Weston. Learning end-to-end goal-oriented dialog. *CoRR*, abs/1605.07683, 2016. 3, 10

[2] Jesse Dodge, Andreea Gane, Xiang Zhang, Antoine Bordes, Sumit Chopra, Alexander H. Miller, Arthur Szlam, and Jason Weston. Evaluating prerequisite qualities for learning end-to-end dialog systems. *CoRR*, abs/1511.06931, 2015. 9

[3] Matthew Henderson, Blaise Thomson, and Jason Williams. The second dialog state tracking challenge. In *Proceedings of SIGDIAL*. ACL Association for Computational Linguistics, June 2014. 10

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. 6

[5] Dan Jurafsky and James H. Martin. *Speech and Language Processing*. Stanford.edu, 2017. ii, ii, 4, 8

[6] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. Weakly supervised memory networks. *CoRR*, abs/1503.08895, 2015. ii, 1, 6

[7] Lukas Burget Jan Chernocky anjeev Khudanpur Tomas Mikolov, Martin Karafiat. Recurrent neural network based language model. 2010. 5, 6

[8] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966. 4

[9] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014. ii, 5

[10] Steve Young, Milica Gasic, Blaise Thomson, and Jason Williams. Pomdp-based statistical spoken dialog systems: A review. 101:1160–1179, 05 2013. 9