Memory Networks For Question Answering

Virgile Audi VAUDI@G.HARVARD.EDU
John A. Paulson School Of Engineering And Applied Sciences

Nicolas Drizard NICOLASDRIZARD@G.HARVARD.EDU
John A. Paulson School Of Engineering And Applied Sciences

Abstract

The focus of this final project is about non-factoid question answering. To tackle this problem, we chose to implement memory network type of models and in particular the dynamic memory network presented in Kumar et al. (2015). For this project update, we mainly worked on pre-processing the data from the bAbi dataset, implementing a count-base baseline as well as looking at a one hop memory weakly supervised memory network.
Keywords: Question Answering, Memory Network

1. Introduction

If we want to communicate and reason with a machine, then the machine will need to be able to ingest and understand the underlying logic of the sentences we communicate to it. Pick the Echo for instance, say you are to tell it that your mother just bought this great new phone on Amazon, and that it makes you jealous. Wouldn't it be great (or at least for Amazon) if the Echo understood that the answer to the question: "why am I jealous?" was the fact that you don't have the latest smartphone and replied by offering to order it for you immediately? This kind of tasks are called non-factoid question answering as they go beyond the scope of querying a knowledge base to answer a question such as "Who was the 1st President of the United States?". In this project, we would like to tackle the issue of non-factoid question answering by implementing Memory Network developped.

2. Problem Statement

The objective of this final project is to implement memory network in order to answer non-factoid questions as a human would do. Given a story, i.e. a collection of sentences, the model is expected to output an answer which can either be a single word or a list of words.

An example of such questions could be:

Story:
Sam walks into the kitchen.
Sam picks up an apple.
Sam walks into the bedroom.
Sam drops the apple.
Question:
Where is the apple?
A: Bedroom

Weston et al introduced a set of 20 tasks to test different text understanding and reasoning situation, to which a human should be able to get perfect scores. Such tasks include single to three supporting facts questions, yes/no questions, counting, or agent's motivations. The goal was therefore to build a unique model capable of solving these 20 taks.

3. Count Based Model

3.1 Intuition

In order to benchmark the results of the Memory Networks, we first decided to implement our own count based baseline. The intuition we had when creating this model was to keep in mind the way humans would answer the questions. Indeed, it seems that the type of question, i.e. what is the question word, gives significant information on what type of answer one would expect. For instance, if the question starts with "who" then it is very unlikely that the answer will "basketball" or "garden". Also when looking for an answer, it is very common to identify the sentences in a given text that will help answer the question. The simplest way to do so would be to look at occurences between words in the question and in the different sentences. This were the main ideas we tried to implement in the model that we will present now in more details.

3.2 Model

More formally, we created our baseline by using two different types of features that we present below.

Answer words counts in the story The first feature evaluate the presence of each answer word in the story, weighted with a decay depending on when it occurs in the story. The idea is basically to count the decayed occurrences of each possible answer word in the story and then to normalize it. It's a simple function taking as input a story, i.e. sequence of facts seen as a bag of words, and return a distribution over the possible answer words. (We know that all the answer words in the test questions have been answer word in the train).

We use a simple affine function for the decay with a smoothing parameter α_1 to be more resilient:

$$\tilde{f}_1(x_i) = \alpha_1 \sum_{k}^{|AW|} \delta_1(x_i = aw_k)(1 + \beta * i)$$

with x_i the word of index i in the story, AW the set of possible answer words and $\delta_1(x_i = aw_k)$ a one hot vector of size $|AW|$ with a one on the k^{th} if $x_i = aw_k$. We used in the experiments: $\alpha_1 = 0.1$ and $\beta = 0.15$.

For a given story $X = (x_1, ..., x_S)$, we have the corresponding f_1 feature:

$$f_1(X) = \sum_{i=1}^{S} \tilde{f}_1(x_i)$$

This feature is easy to build for a baseline but contains a lot of drawbacks. First, we just apply a dummy decay over the time. For instance if the answer of a question relies in the first sentence of a story and then comes several sentences without any extra information relative to the question but with possible answer words these wil get a higher score than the real answer. Furthermore, we are not using the question in this feature. Moreover, this feature is just extracted on the fly from the input and we don't take advantage of the train test we have.

Question embeddings This feature aims to use the question, especially the kind of answers it expects: yes/no question, locations, activity, person... This information relies mainly in the question word. As a result, we just extract the first word of the question and embedd it as a vector of size $|AW|$. This embedding is learned at train time and corresponds to the occurences of each answer words as answer to a question with this specific answer words (with a smoothing α_2). It will provide a prior information on the expected answer given the question word.

For a given question $Q = (q_1, ..., q_{|Q|}))$:

$$f_2(Q) = \tilde{f}_2(q_1) = \alpha_2 + \sum_{i=1}^{N_{train}} \mathbb{1}(q_1^{(i)} = q_1)\delta_2(q_1, aw_i)$$

with $q_1^{(i)}$ the first word of the i_{th} question fromt the train set and $\delta_2(aw_i)$ a one hot vector of size $|AW|$ with a one on the i^{th} . We use in the experiments $\alpha_2 = 0.1$
This feature takes advantage of the train set and uses part of a question on contrary to the first feature. We could still extend it while also using the rest of the question.

Prediction The input of the model is a tuple story, question (X, Q). We treat the feature as independant probability distribution as in the Naive Bayes approach so we just simply combine the feature witha product. In the experiment we treat them in the log space for computational purpose and take the argmax of the vector coordinates.

Concretely,

$$\hat{y}(X, Q) = argmax(\log(f_1(X)) + \log(f_2(Q)))$$

4. End-to-end Memory Network

We now introduce the Memory Network as presented in Sukhbaatar et al.

4.1 Architectures

The model takes for inputs the question and the sentences of the story. The sentences of the story are saved to memory by embedding them with two differents look-up tables (matrix A and C in the diagram below), corresponding to the input and output memory representation. Note that the embedded representation of sentences and questions are obtained by summing the embeddings of their words. These input representation of the story's sentences will then be combined with the embedded representation of the question (using the matrix B) using a dot product. The result of this dot product is then passed through a softmax layer to give a probability distribution over which sentence is likely to give information about the answer. The memory output vector o results from a weighted sum of the output embeddings of the sentences (using matrix C) using the probability distribution p. We then apply a weight matrix to the sum of question embeddings and the output vector o followed by a softmax to predict the answer. We can summarize this process with the formula:

$$\hat{a} = softmax\left(W(o+u)\right)$$

where: $o = \sum_{i=1} softmax(u^T m_i)$, with u being the embedded representation of the question and m_i the embedded represenation of the sentence i in the story.

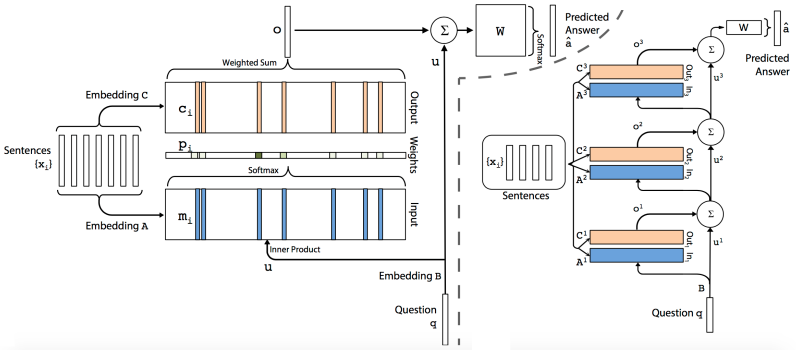

Figure 1: 1-Hop and 3-Hop MeMNetwork

But as one can see on figure 1, we can in fact stack mutliple hop to memory, in this case 3. This would increase performance when dealing with tasks that would require reasoning on more than one fact. In more details, we define the intermediate outputs o_k and u_k by:

$$o_k = \sum_{i=1} softmax(u_k^T m_i)$$

$$u_{k+1} = u_k + o_k$$

4.2 Parameters Tying

As mentioned in Sukhbaatar's paper, if using multiple hops to memory yields better results, it also has for consequence to increase significantly the number of parameters. We therefore investigated two different constraints on the embedding matrics A_i and C_i, known as parameter tying:

- **Adjacent**: $A_{i+1} = C_i$ i.e. the current output embedding is equal to the next input embedding,
- **Layer-wise or RNN-like**: $A_{i+1} = A_i$ and $C_{i+1} = C_i$ for all i

When using the RNN-like tying, it is prefered to use define u_{i+1} as a linear transformation of u_k, i.e. $u_{i+1} = Hu_i + o_i$. We will compare results using the two different tying methods in the next section.

5. Experiment

5.1 Data and Preprocessing

[A FAIRE] - padding with parameters set to 0 for padding - List Answers - Reponse similaire dans le train et le test

5.2 Implementation Details

We tested multiple hacks in order to improve performance that we will now detail.

Temporal Encoding In order to account for a certain notion of temporality in our baseline model, we weighted word occurences differently based on how close these words were from the question. For the memory network, we add to both the input and output memories, temporal matrices T_{A_i} and T_{C_i} that will be learn during training, and to which we apply the same parameter constraints as A_i and C_i. We refer to Temporal Encoding as TE.

Position Encoding We choose to represent sentence embeddings as the sum of its words' embeddings for simplicity reasons. Indeed, it was then possible to obtain a standard size of each entry in the memory, equal to the hidden embedding size. Nevertheless, this choice could potentially loose information about word position that concatenation would pick up. To avoid this loss of information, we replace the current memory representation:

$$m_i = \sum_j Ax_{ij}$$

by

$$m_i = \sum_j l_j \cdot Ax_{ij}$$

where $\cdot$ corresponds to the element-wise multiplication and l_j is a fixed vector with entries defined by:

$$l_kj = (1 - \frac{j}{J}) - \frac{k}{D}(1 - 2\frac{j}{J})$$

with j being the indexed of the sentence, J the number of sentences allowed in memory and D the hidden dimension of embeddings. We refer to position encoding as PE.

Linear Start Finally, we implemented what Sukhbaatar defines as Linear Start Training (LS). During training, we start by removing the softmax in each memory layer and use validation loss as our trigger. When loss stops decreasing, we add the softmaxes back and continue training.

5.3 Results

[A FAIRE]

Results for following configuration on test: - 1 hop adjacent joint - 1 hop TE adjacent joint - 2 hop TE adjacent joint - 3 hop TE adjacent joint - 1 hop TE PE adjacent joint - 2 hop TE PE adjacent joint - 3 hop TE PE adjacent joint - 3 hop TE PE adjacent by task - 3 hop TE PE RNN like joint

if time: - 1 hop TE PE adjacent joint LS - 2 hop TE PE adjacent joint LS - 3 hop TE PE adjacent joint LS

6. Future Work

6.1 Application to the MCTest dataset

[A FAIRE]

6.2 Dynamic Memory Networks

[A FAIRE]

7. Conclusion

[A FAIRE]

References

Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *CoRR*, abs/1506.07285, 2015. URL `http://arxiv.org/abs/1506.07285`.