Extending Dynamic Structure in Memory Network for Response Generation

André Cibils
andre.cibils@epfl.ch

***Abstract*—Question Answering is an important and growing field in the Natural Language Processing domain, which concern multiple and diverse real life applications.**
Dynamic Memory Networks are a new and powerful way to handle the Question Answering problem. They are a neural network architecture which learns to answer questions by using raw input-question-answer triplets. However, they only produce one-word answer and thus cannot be easily adapted to construct chat-bots.
In this paper, two major modifications are made to allow the model to produces multiple-words answer. The first modification, inspired by the Encoder-Decoder model, gives great results but fails to scale up with difficult tasks. The second modification, the Pointer Net problem, is an interesting direction to look at but fails to solve this problem on its own.
This two modifications and their implementations are presented here.

I. Introduction

Natural Language Processing (NLP) is a complex field concerned by the interactions between computers and human language. Question Answering (QA) is a discipline within the NLP field concerned with building systems that automatically answer questions asked in natural language. This is a particularly complex and interesting task, as it requires an understanding of the meaning of a text and the ability to reason over relevant and multiple facts.
Moreover, QA is an important and developing field, as it's useful for multiple domains, like job automation (professional chat bots), personal assistant (Siri), etc. In 2001, Watson, which is a question answering computer system developed by IBM competed against former player in the game $jepoardy$! and won the first prize. Nowadays, a lot of people use artificial neural network to handle the QA problem. Theses types of models can provide a great performance, but they need a lot of data to train them. Indeed, artificial neural networks are computational models used in machine learning, which consist of large collections of connected simple units called artificial neurons.
The Dynamic Memory Network (DMN) is a fairly new artificial neural network based framework for general question answering. It is trained using raw input-question-answer triplet. DMNs proved to be powerful and are a really interesting direction to look at if one wants to work on QA.
In this paper, the goal is to propose modification to the memory structure of the dynamic memory network to answer questions with sentences instead of single words only. Indeed, this would have some useful openings, such as more accurate chat-bot building. DMN can in fact already perform part-of-speech tagging, which consists of attributing grammatical tags to each word of the input.
Here, in the first place, a description of the DMN architecture is presented, then, in a second time, different modifications to the model are detailed. Finally, results and analyzes of these changes are provided.

II. Dynamic Memory Network

The Dynamic Memory Network (Kumar et al. (2016)) is an end-to-end neural network based framework that can be used for general question answering tasks. Generally, it can solve sequence tagging tasks, classification problems, sequence to sequence tasks and question answering problems that require transitive reasoning.
The DMN has four distinct parts, which together take care of the task. It starts by computing an intern representation of the context and the question, then trigger an iterative attention process which will search the inputs to retrieve important facts. It will then produce an intern memory vector representing all relevant information, which will be used to compute the answer, i.e. the prediction.

In the following subsections, the different parts of the DMN, called module, will be examined.

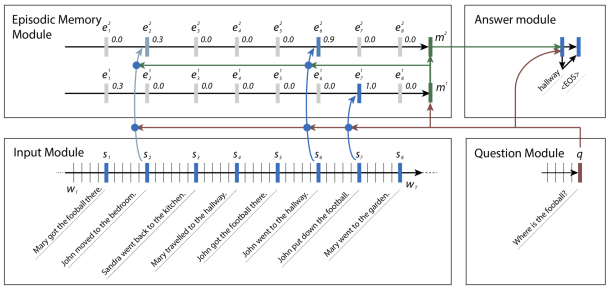


Figure 1: A real example of how the original DMN works.

A summary of the DMN can be seen at fig 1.

A. Input Module

This module encodes raw text inputs into distributed vector representation. It uses a well-known Natural Language Processing technique, called word embedding. The input may be a sentence, an article, or even several documents such as movie review or Wikipedia articles.

1) Word Embedding: For this problem and most natural language processing problems, the input is a sequence of words. The sentences, which are concatenated, will here be called S_I and are constituted by the words $w_1^I, w_2^I, ..., w_{T_I}^I$. The sequence is translated using word embedding, which is a classic way to handle words in a model.
Word embedding is an efficient way to represent words from a dictionary into vectors. The size of the vector must be manually chosen, as it is an hyper-parameter. The main advantage of using word embedding is that words appearing in similar contexts will have similar vectors. For this task, pretrained GloVe vectors are used, as training specific vectors would take a lot of time and probably be less efficient. GloVe, or Global Vector for Word Representation is a model which learns geometrical encoding, or vector, from their co-occurrences information.

2) Gated Recurrent Network: A Recurrent Neural Network (RNN) is very similar to a classic neural network, but differ by having an internal vector named the hidden state, called h_t. It is time depend and the output is computed using not only the input, but the hidden state as well.
Recurrent neural network are being used to encode the input sequence of the words embedding of S_I. Different types of recurrent network exist, such as the standard tanh RNN or the complex LSTM, but the one used here is the Gated Recurrent Unit (GRU). Indeed, GRU have been shown to be faster to train and to perform better (Kumar et al. (2016)). Assume each time step t has an input x_t and a hidden state h_t. Then the internal mechanics of the GRU are defined as:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1} + b^{(z)})$$

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1} + b^{(r)})$$

$$p_t = tanh(W^{(p)}x_t * U^{(p)}h_{t-1} + b^{(p)})$$

$$h_t = z_t * h_{t-1} + (1 - z_t) * p_t$$

Where $*$ is the element-wise product, $W^{(z)}$, $W^{(r)}$, $W^{(p)} \in \mathbb{R}^{n_h * n_I}$, $U^{(z)}$, $U^{(r)}$, $U^{(p)} \in \mathbb{R}^{n_h * n_h}$ and $b^{(z)}$, $b^{(r)}$, $b^{(p)} \in \mathbb{R}^{n_I}$ are the network weights and bias, they are all trainable matrix. The dimensions n are hyper-parameters. All of this parameters are learned - as explained later on - by using the back propagation through time algorithm.
The sigmoid function σ is defined here by $\sigma(x) = \frac{1}{1+e^{-x}}$. Sigmoid functions have domain of all real numbers, with return value monotonically increasing from 0 to 1. They are often used as activation function of artificial neurons, as they are differentiable and known to be easy to train and simple. The vector z_t is called the update gate, r_t the reset gate, p_t the potential state and h_t the hidden state, or updated state. The above computation is abbreviated with $h_t = GRU(x_t, h_{t-1})$.

Having that, for each sentence, a representation c_t is produced by iterating on the GRU and by using word embedding to represent the words of the current sentence. More precisely, the input module will use a GRU and the input words to compute $c_t = GRU(L[w_t^I], c_{t-1})$, where L represents the word embedding matrix and w_t^I the word of the t-th word in the input. At the end of each sentence, the corresponding fact representation c is outputted by this module. It is important to note that a number T_C of fact representation is produced, where T_C is equal to the number of sentences.

B. Question Module

For this module, the same idea is used for the question. Indeed, the question is also a sequence of T_Q words, called $S_Q = w_1^Q, w_2^Q, ..., w_{T_Q}^Q$. As before, the model starts by using word embedding to translate the words into vector.
Then, by iterating on a different GRU, the question module produces a sequence of hidden state $q_t = GRU(L[w_t^Q], q_{t-1})$, where L represents the word embedding matrix and w_t^Q the word index of the t-th word in the question. Unlike the input module, the question module then outputs only the final hidden state as the question representation $q = q_{T_Q}$

C. Episodic Memory Module

The role of the episodic memory module is to iterate over the fact representation outputted by the input module using the question representation. It also has an internal memory, which is updated at each iteration. It is divided in two parts: the attention mechanism and the memory update mechanism.
At each iteration, the attention mechanism attends over the fact representations c while taking into consideration the question representation q and the previous memory m_{t-1} to produce what is called an episode e_t. Then, this episode e_t is used, alongside the previous memory m_{t-1} to update the episodic memory and produce a new memory m_t.
After T_M passes, the final memory m_{T_M} is outputted by the episodic memory module and passed to the answer module.

1) Attention Mechanism: The attention mechanism must produce an episode e at each iteration. For that, it will first calculate a gate g_i for each fact c_i: $g_i = G(c_i, m, q)$. It uses the previous memory m and the question q to compute the scoring function G, which is a two-layer feed forward neural network. G itself takes as input the feature set $z(c, m, q)$, defined as a large vector that captures a variety of similarities between input, memory and question vectors.

$$G(c, m, q) =$$
$$\sigma(W^{(g_2)} tanh(W^{(g_1)} z(c, m, q) + b^{(g_1)}) + b^{(g_2)})$$
$$z(c, m, q) =$$
$$[c, m, q, c*q, c*m, |c-q|, |c-m|, c^T W^{(b)} q, c^T W^{(b)} m]$$

where $*$ is the element-wise product and the W and b are weights and bias. The function σ is once again the sigmoid function, $|x|$ is the absolute value of x and c^T represents the transpose of the vector c.
To compute the episode for pass t, a modified GRU is employed over the sequence of the inputs $c_1, ...c_{T_C}$, weighted by the gates g. The equation to update the hidden states of the GRU at time t and the equation to compute the episode e_t are respectively

$$h_{i,t} = g_{i,t} * GRU(c_i, h_{i-1,t}) + (1 - g_{i,t}) * h_{i-1,t}$$
$$e_t = h_{T_C,t}$$

2) Memory Update Mechanism: The role of the memory update mechanism is to update the memory vector m at each episode. For this, it uses a simple GRU $m_t = GRU(e_t, m_{t-1})$. The memory vector is initialized by the question vector $m_0 = q$. The output of the episodic memory update module is the final memory m_{T_M}, produced after T_M passes over the facts. The memory update mechanism iterates multiple times, which allows this module to attend to different inputs during each pass. It is useful for a type of transitive inference and may help for sentiment analysis (Kumar et al. (2016)).

D. Answer Module

The answer module is a simple one which uses the last memory produced by the previous module to generate an answer to the question. Depending on the type of task, the answer module is either triggered once or multiple times.
The answer vector is initialized to the last memory vector $a_0 = m_{T_M}$. Then, at each time step t, a GRU is used to produce an other answer vector.

$$y_t = softmax(W^{(a)} a_t)$$
$$a_t = GRU([y_{t-1}, q], a_{t-1})$$

where q is the question vector outputted by the question module. It is concatenated with the previously predicted output y_{t-1}.

III. Modifications

In this part, multiple modifications to the DMN are proposed in order to generate long answers, i.e. sentences or group of words. The original DMN are not able to produce them, but it looks like the architecture could work for sentence generation.
Two major changes are described below, the first one comes from the encoder-decoder architecture, and the other one from pointer nets.

A. Encoder-Decoder Architecture

Encoder-Decoder models are used in multiple task which needs a generative model, such as auto-completion or query reformulation as shown in Sordoni et al. (2015). In this tasks, context-awareness and the word order are important, as in the problem of multiple word answer generation. The main idea is to have two RNN working together. The first one, called $encoder$ takes as input the word embeddings of the context and its final hidden state can be considered as a compact order-sensitive and context-aware encoding of the initial input. Then, the second one, the $decoder$, is used to generate a sentence given an initial query encoding. Each of the hidden state is used to estimate the probabilities of the next word in the sequence.

1) Using the basic framework: The basic answer module can be already considered as a decoder RNN. Indeed, it uses a compact vector as input, outputted by a context-aware and order-sensitive module. Below is the actual computation done in the original answer module, where $a_0 = m_{T_M}$

$$y_t = softmax(W^{(a)} a_t)$$

$$a_t = GRU([y_{t-1}, q], a_{t-1})$$

The idea here is to simply train the model with sentences, in order to make it learn how to produce multiple word given a simple vector as input. In theory, a conditional ending to the iteration process can be put in place. Indeed, it only needs the model to be able to output an end of sentence token (<eos>), and it could then stop itself.

2) Adding the original memory vector as input: This modification is really similar to the previous one. The experiment shows that a lot of information is lost at each time step and that the model learns as a priority to produce the easiest part of the sentences.
In order to save information and to reduce the number of examples needed to train the network, the answer module is modified as below

$$a_t = GRU([y_{t-1}, q, a_0], a_{t-1})$$

By adding the last memory vector $a_0 = m_{T_M}$ concatenated in the input, the model can focus on modifying its hidden state without losing information, about the context, to produce the words for the answer.

B. Pointer Nets

Pointer nets are a different kind of models, where the goal is to generate an output sequence whose token must come from the input sequence. They use an attention mechanism as a pointer to select a position from the input sequence as an output symbol. Here, and similarly to the boundary model of Zhang et al. (2017), the model produces only two indices a_s and a_e, respectively for the start and for then end of the answer.

More precisely, this modified answer module takes obviously m_{T_M} as input, and compute two pairs of values by using a two-directional prediction module. The first pair is calculated by the following

$$P(s+) = softmax(W^{(s+)} m_{T_M})$$

$$P(e+) = softmax(W^{(e+)} m_{T_M} + W^{(c+)} c_{s+})$$

where all $W \in \mathbb{R}^{T_I * n_h}$ are trainable matrices, c_{s+} is the hidden state from the input module at index $s+$. Indeed, even if only one fact representation c per sentence is used by the episodic memory module, the input module actually produces one representation c for each word.
Theses operations are also performed by predicting the end position first and then the starting position.

$$P(e-) = softmax(W^{(e-)} m_{T_M})$$

$$P(s-) = softmax(W^{(s-)} m_{T_M} + W^{(h-)} c_{e-})$$

Finally, the span of the answer is identified with the following equations

$$P(s) = mean(P(s+), P(s-))$$

$$P(e) = mean(P(e+), P(e-))$$

The two-direction prediction idea has been added in a second time, and the results of this additional modification will be presented later.

IV. Implementation details

In this section, the different data set which have been used and the modifications applied to them will be presented, but also the different loss calculations and some of the minor architecture modifications.
All the models tested below employ L_2 regularization, and dropout on the word embedding, which means that if the word is not in the embedding matrix, a random

vector will be created. More precisely, it uses a uniform distribution between 0.0 and 1.0. Word vector are pre-trained using GloVe and their embedding size is 50. All the models are all trained via back-propagation through time (Mozer (1989), Werbos (1988)) with the Adaptive Learning Rate Method (AdaDelta, D. Zeiler (2012)), on the Adaptive Moment Estimation algorithm, or Adam (P. Kingma and Ba (2014)). The use of a test set, created by cutting at least 10% of the training set, guarantee that if over-fitting occurs, it will be seen in the results.
All the GRU have a total of 40 hidden units, and the iterative process in the episodic memory module, or the number of memory hops done, is 5.

A. *Classic Dynamic Memory Network*

The original DMN has been run mainly on the bAbI data set from Facebook (Weston et al. (2015)). It's a synthetic data set for testing a model's ability to retrieve facts and to reason over them. There is a total of 20 tasks, each of them testing a different skill that a QA model ought to have, such as deduction or inference.

Context:
John travelled to the hallway.
Mary journeyed to the bathroom.
Daniel went back to the bathroom.
John moved to the bedroom.
Question: Where is Mary?
Answer: bathroom

Figure 2: Example of an input-question-answer triplet.

As the data set is synthetic, passing a task does not mean that the model would also exhibit the respective ability on real world text data. However, it is a necessary condition.
It has been shown (Kumar et al. (2016)) that DMN is a powerful model for this bAbI tasks. Indeed, it passes (with accuracy$> 95\%$) 18 of the tasks. These results are detailed in the fig 3.

The size of the training set here is usually 1000 examples, sometimes 10000. The size of the testing set is always 1000 examples.

Task	MemNN	DMN
1: Single Supporting Fact	100	100
2: Two Supporting Facts	100	98.2
3: Three Supporting Facts	100	95.2
4: Two Argument Relations	100	100
5: Three Argument Relations	98	99.3
6: Yes/No Questions	100	100
7: Counting	85	96.9
8: Lists/Sets	91	96.5
9: Simple Negation	100	100
10: Indefinite Knowledge	98	97.5
11: Basic Coreference	100	99.9
12: Conjunction	100	100
13: Compound Coreference	100	99.8
14: Time Reasoning	99	100
15: Basic Deduction	100	100
16: Basic Induction	100	99.4
17: Positional Reasoning	65	59.6
18: Size Reasoning	95	95.3
19: Path Finding	36	34.5
20: Agent's Motivations	100	100
Mean Accuracy (%)	93.3	**93.6**

Figure 3: Results of the test accuracies on the bAbI data set. MemNN numbers taken from Weston et al. (2015). The DMN passes 18 tasks with an accuracy higher than 95%.

B. *Encoder-Decoder Architecture*

The two models proposed which follow the encoder-decoder logic have only been trained on the first and third bAbi tasks, which are $Single\ Supporting\ Fact$ and $Three\ Supporting\ Facts$. Indeed, both of this tasks can be modified to have sentences as answer. However, these data sets have been modified as the answer must be a sentence and not a single word. The modifications goes as in fig 4.

Obviously, these transformations are quite simple, as they always follow the same logic for both of the tasks. For the first one for example, they all look like <$Name$> $went\ to\ the$ <$Place$>, as shown in fig 4a. However, the idea is to simply test if the answer module success to endorse the role of a decoder. If

it's working, then the data set could be modified to try to generate more complex structure.
The third bAbI data set has not only longer sentences as answers, but requires also more complex reasoning from the model, see fig 4b. Indeed, the basic DMN performed a bit worse on it than on the first bAbI task, as shown in fig 3. By testing these implementations on this data set, it will be easier to see how much the sentence generative ability is hard to train.
The loss is calculated by summing each individual loss between the words. This insure that the model won't be biased to learn specific words, particularly the hard ones, such as <$Place$> or <$Name$>, which is necessary to test if the model is still able to show reasoning ability despite this changes.

Question: Where is Mary?
Original Answer: bathroom
Sentence Answer: Mary went to the bathroom.

(a) Example of the modification applied to the first bAbI data set

Question: Where was the apple before the bathroom?
Original Answer: garden
Sentence Answer: The apple was in the garden before being in the bathroom.

(b) Example of the modification applied to the third bAbI data set

Figure 4: The modification applied to the first and third bAbI data sets in order to have sentences as answer instead of single word answer.

The implementations presented here are the same for the two different approaches of the Encoder-Decoder Architecture, that is, using the basic framework or adding the original memory vector as input to the decoder part.
In a second step, it could be interesting to allow the model to answer with multiple forms of sentences. Indeed, saying <$Name$> $is\ in\ the$ <$Place$> could be considered as a valid answer for the first modified bAbI task. However, it is important to test at first if the model is able to answer the questions with simple and formatted sentences, and if this works, allowing the use of more complex answers should definitely be implemented.

C. Pointer Nets Architecture

For this model, the SQuAD QA (Rajpurkar et al. (2016)) data set fits particularly to this problem. It consists of questions posed by crowd-workers on a set of Wikipedia articles, where the answer is always a segment of text, or span, from the corresponding reading passage. An example of this data set can be seen in fig 5
The data set originally contains around 87000 examples.

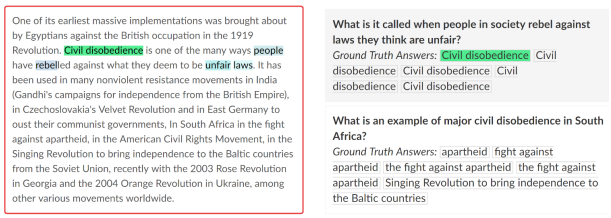


Figure 5: An example of the SQuAD QA Data set, with the context, the questions and answers associated.

An important thing to note, is that the size of the input must be fixed. Indeed, the computation $W^{(c+)}c_{s+}$ requires a matrix of size T_I times T_I, i.e. the number of words in the input. For this, the input has been sorted by removing from the data set all the contexts which had more than T_I words, where T_I is an hyper-parameter. The rest have been padded with end-of-sentence <eos> tokens to insure that all the input has the same size.
However, this is an important trade-off: if T_I is too big, the model may not be trainable, but if T_I is too small, the data set may not be good enough. The average value of T_I over the data set is 136.

V. Results

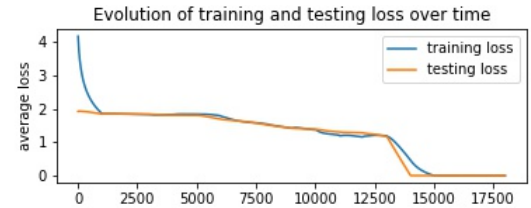
The evaluation consists of observing three measures, which are the loss, the so called soft precision and the hard precision. The loss is outputted by the model by comparing the expected answer and the prediction, using Adam (P. Kingma and Ba (2014)). The soft precision is the mean of the number of words that the model got right. The hard precision is a Boolean, set to one if the model output the right sequence of words, or zero if it fails. For example, if the model outputs "$Marie\ went\ to\ the\ kitchen$." but the right answer was "$Marie\ went\ to\ the\ hallway$.", the soft accuracy

is 80% but the hard accuracy is 0.
A hard precision per word can also be useful to see where the model is having trouble to learn, and what it masters quickly. This is presented below as accuracy per word, at index i.

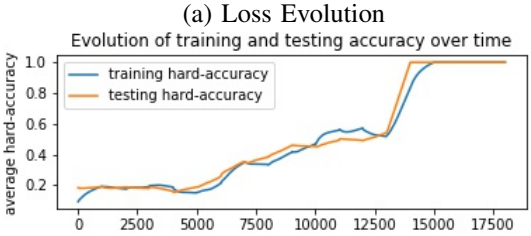
A. Encoder-Decoder Model

The encoder-decoder model gives really good results on task 1 as seen in fig 6, but it seems that the model needs a lot more data for task 3. Indeed, with relatively small amount of data, over-fitting occurs rapidly, as it is clearly shown in fig 7.

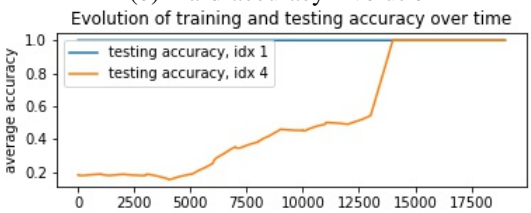
1) Basic Framework: Task 1 results are good, as the model achieves an hard-accuracy of 1 on the test set. This is important, as it means that the sentence generating ability did not impact the fact retrieval and reasoning parts. Indeed, the original DMN, which produced only one word answer, achieved an accuracy of 100% for the first bAbI task.



(a) Loss Evolution



(b) Hard-accuracy Evolution

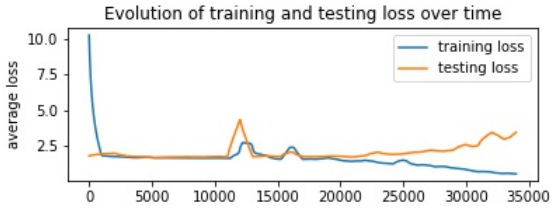


(c) Evolution of the words $went$, at index 1, and <$Place$>, at index 4

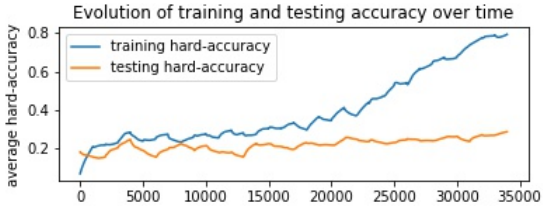
Figure 6: Evolution of the performance of the basic framework on the modified first bAbI task

Without surprise, the model learns really well and quick the static part of the sentence. Indeed, on fig 6c, the hard-accuracy for the word $went$ is 1 for almost all the training session, as it did not pose any problem to the model to guess it. However, it takes some iterations to get the first and last word (<$Name$> and <$Place$>), i.e. the parts of the sentence which need reasoning.

Task 3, however, seems to be a lot harder. Indeed, with 1000 examples, the model over-fit the data. It is really clear in fig 7a: when the training loss goes up but the testing loss goes down, this means that the model learned the training set, i.e. it over-fits.



(a) Loss Evolution

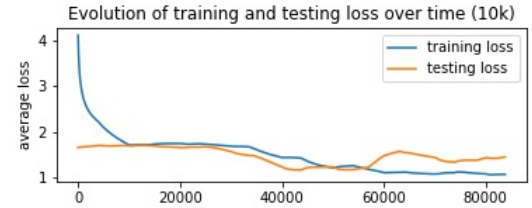


(b) Hard-accuracy Evolution

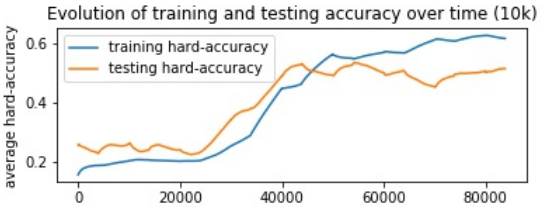
Figure 7: Evolution of the performance of the basic framework on the modified third bAbI task

Considering this results on a small data set, a bigger training set has been used. It contains 10000 examples, which should prevent the model from over-fitting.

As seen in the fig 8, it's difficult and it does not work as well as hoped. In fact, it began to over-fit as well which was unexpected, given the size of the training set. Indeed, the model still performed better (around 50% hard-accuracy, while with only 1000 example it achieves around 20%), but it looks like the model could use even more data.
The original DMN achieved an accuracy of 95.2%, as shown earlier in fig 3, which is obviously a better result. This means that the sentence generation ability of the model costed performance to the reasoning ability of the model.

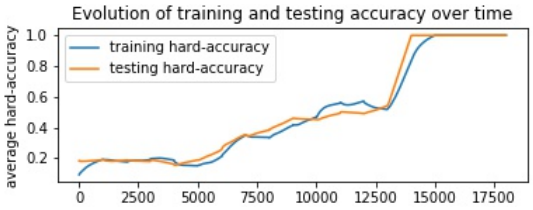


(a) Loss Evolution

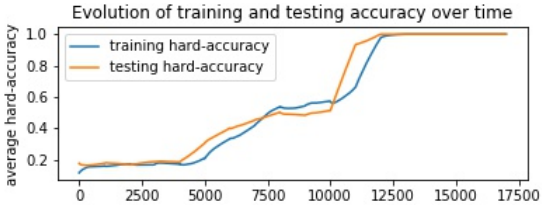


(b) Hard-accuracy Evolution

Figure 8: Evolution of the performance of the basic framework on the modified third bAbI task, with 10000 examples



(a) Hard-accuracy Evolution, with using only the basic framework



(b) Hard-accuracy Evolution, with adding the original memory vector as input

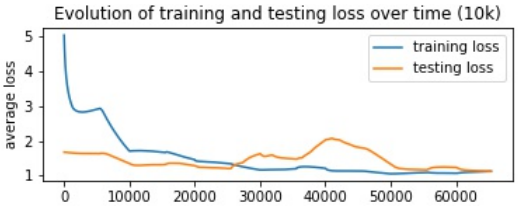
Figure 9: Results in term of hard-accuracy of the influence of adding the original memory vector in the answer module as input

Considering these results, a conclusion can be made: training the answer module's GRU as a decoder is hard and need both a lot of time and data. The goal of the next modification is to make the life simpler for the model, by allowing it to reuse the memory vector. It should improve the performance, make the training session quicker and make the answer module less data consuming.

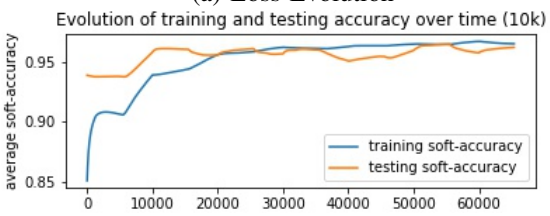
2) Adding the original memory vector as input: Without surprise, this modification did not impact the performance for the first task. However, it is important to notice that this new model learned quicker how to answer the question. Indeed, after only 11 epochs, the hard-accuracy is already one: comparing to the previous model, it required 2-3 epochs less.

Concerning the 3th task, as small amount of data (1000 examples) make the model over-fit again. This issue led to use the 10000 examples data set, with which the model achieves an accuracy of 60% after many epochs as seen if fig 10.

This confirms that the Decoder-Encoder architecture is a good direction, but also that it does not suffice on its own, as it needs a lot of data and a huge amount of time to train. Indeed, it is well known that using GRU as decoder can be difficult.



(a) Loss Evolution



(b) Soft-accuracy

Figure 10: Evolution of the performance of the second framework on the modified third bAbI task, with 10000 examples

For this reason, a different approach has been taken, in the form of the pointer net model

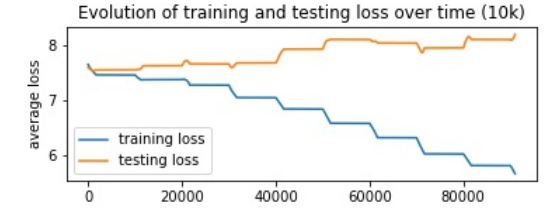
B. Pointer Net Model

The soft-accuracy and the hard-accuracy measures have been modified, as they don't fit the problem. The position of the pointers are here compared, and not the words themselves.

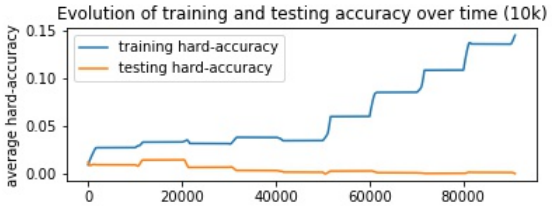
The hard-accuracy is still a Boolean, which is set to 0 if one of the two predicted pointers is not the right one. The soft accuracy is computed as below: $soft - accuracy(p_s, p_e, a_s, a_e) =$

$$\frac{1}{2}(\frac{1}{1+|p_s - a_s|} + \frac{1}{1+|p_e - a_e|})$$

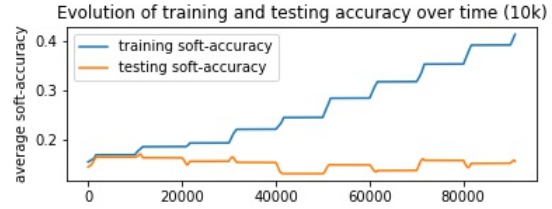
where p_s and p_e are the predictions for the start and the end of the answer, and a_s and a_e are the right start and the end of the answer.
The loss calculation is still outputted directly and as before by the model.
The pointer net model did not performed well. It fails to improve its accuracy and reduce its loss, which implies that the model on its own does not work coupled with DMN.



(a) Loss Evolution



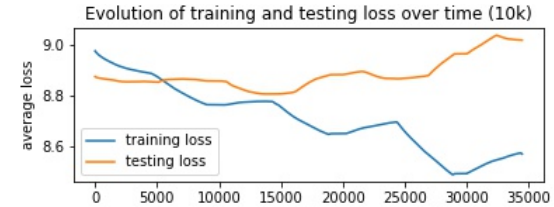
(b) Hard-accuracy Evolution



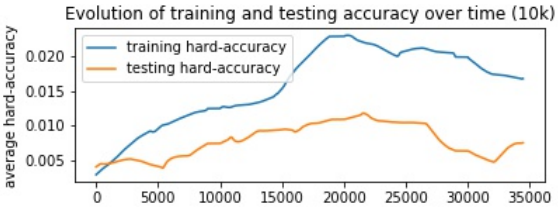
(c) Soft-accuracy Evolution

Figure 11: Evolution of the performance of the pointer net framework on the SQuAD data set, with $T_I = 50$, $n_h = 40$, $GloVe\ Dim = 50$

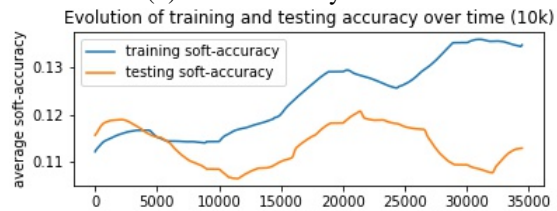
With small amount of data, like in fig 11, the model quickly over-fits. However, even with the hyper parameter T_I set to 100, the model doesn't produce good results, as seen in fig 12.



(a) Loss Evolution



(b) Hard-accuracy Evolution



(c) Soft-accuracy Evolution

Figure 12: Evolution of the performance of the pointer net framework on the SQuAD data set, with $T_I = 100$, $n_h = 100$, $GloVe\ Dim = 50$

This means that the model can be trained, but it also confirms that the pointer net architecture on its own fails to solve this problem.
Multiple hyper parameters have been tested, like T_I, n_h, i.e. the number of hidden units, but also the number of memory hops.

VI. Conclusions

Modifying the structure of the Dynamic Memory Network to produce sentence and multiple-words answer is a complicated task. In this paper, it has been shown that the encoder-decoder architecture is a great way to start, but it needs to be combined with other techniques. Indeed, on its own, it often ends up by costing to the model its information retrieval and reasoning ability. Moreover, this modification made the model highly data consuming, making it over-fit on relatively big data set, where the original DMN used to achieve high results with small amount of data.
Pointers nets are another potential direction to look, as they are used in other artificial network based framework to retrieve multiple-words answer. However, on

its own, this architecture seems to fail to solve this problem coupled with DMN.
A possible next modification should probably be a mixture of these two solutions, as each one could compensate the difficulty of the other.
Moreover, a major modification have been made to the DMN recently to make it handle images (Xiong et al. (2016)), and maybe this new DMN+ could handle the pointer network. Finally, an other network, the Dynamic Coattention Network (Xiong et al. (2017)) is also a new architecture to look at.

A. Acknowledgement

This research has been supported by the Artificial Intelligence Lab. at the Ecole Polytechnique Federale de Lausanne (EPFL).
I would like to thanks my supervisor Mi Fei and director Prof. Boi Faltings for their help and wisdom.

REFERENCES

Matthew D. Zeiler. Adadelta: An adaptive learning rate method. 2012.

Ankit Kumar, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. 2016.

Michael Mozer. *Complex System: A Focused Backpropagation Algorithm for Temporal Pattern Recognition*. 1989.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ question for machine comprehension of text. 2016.

Alessandro Sordoni, Yoshua Bengio, Hossein Vahabi, Christina Lioma, Jakob G. Simonsen, and Jian-Yun Nie. A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. 2015.

Paul Werbos. *Neural Network: Generalization of backpropagation with application to a recurrent gas market model*, volume 1. 1988.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander Rush, Bart van Merrienboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. 2015.

Caiming Xiong, Stephen Merity, and Richard Socher. Dynamic memory networks for visual and textual question answering. 2016.

Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. 2017.

Junbei Zhang, Xiaodan Zhu, Qian Chen, Lirong Dai, Si Wei, and Hui Jiang. Exploring question understanding and adaptation in neural-network-based question answering. 2017.