MEMORY NETWORKS

Jason Weston, Sumit Chopra & Antoine Bordes
Facebook AI Research
770 Broadway
New York, USA
{jase,spchopra,abordes}@fb.com

arXiv:1410.3916v11 [cs.AI] 29 Nov 2015

ABSTRACT

We describe a new class of learning models called *memory networks*. Memory networks reason with inference components combined with a long-term memory component; they learn how to use these jointly. The long-term memory can be read and written to, with the goal of using it for prediction. We investigate these models in the context of question answering (QA) where the long-term memory effectively acts as a (dynamic) knowledge base, and the output is a textual response. We evaluate them on a large-scale QA task, and a smaller, but more complex, toy task generated from a simulated world. In the latter, we show the reasoning power of such models by chaining multiple supporting sentences to answer questions that require understanding the intension of verbs.

1 INTRODUCTION

Most machine learning models lack an easy way to read and write to part of a (potentially very large) long-term memory component, and to combine this seamlessly with inference. Hence, they do not take advantage of one of the great assets of a modern day computer. For example, consider the task of being told a set of facts or a story, and then having to answer questions on that subject. In principle this could be achieved by a language modeler such as a recurrent neural network (RNN) (Mikolov et al., 2010; Hochreiter & Schmidhuber, 1997) as these models are trained to predict the next (set of) word(s) to output after having read a stream of words. However, their memory (encoded by hidden states and weights) is typically too small, and is not compartmentalized enough to accurately remember facts from the past (knowledge is compressed into dense vectors). RNNs are known to have difficulty in performing memorization, for example the simple copying task of outputting the same input sequence they have just read (Zaremba & Sutskever, 2014). The situation is similar for other tasks, e.g., in the vision and audio domains a long term memory is required to watch a movie and answer questions about it.

In this work, we introduce a class of models called memory networks that attempt to rectify this problem. The central idea is to combine the successful learning strategies developed in the machine learning literature for inference with a memory component that can be read and written to. The model is then trained to learn how to operate effectively with the memory component. We introduce the general framework in Section 2, and present a specific implementation in the text domain for the task of question answering in Section 3. We discuss related work in Section 4, describe our experiments in 5, and finally conclude in Section 6.

2 MEMORY NETWORKS

A memory network consists of a memory $\mathbf{m}$ (an array of objects[1] indexed by $\mathbf{m}_i$) and four (potentially learned) components I, G, O and R as follows:

> I: (input feature map) – converts the incoming input to the internal feature representation.

---

[1]For example an array of vectors or an array of strings.

G: (generalization) – updates old memories given the new input. We call this generalization as there is an opportunity for the network to compress and generalize its memories at this stage for some intended future use.

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.

Given an input x (e.g., an input character, word or sentence depending on the granularity chosen, an image or an audio signal) the flow of the model is as follows:

1. Convert x to an internal feature representation $I(x)$.
2. Update memories $\mathbf{m}_i$ given the new input: $\mathbf{m}_i = G(\mathbf{m}_i, I(x), \mathbf{m})$, $\forall i$.
3. Compute output features o given the new input and the memory: $o = O(I(x), \mathbf{m})$.
4. Finally, decode output features o to give the final response: $r = R(o)$.

This process is applied at both train and test time, if there is a distinction between such phases, that is, memories are also stored at test time, but the model parameters of I, G, O and R are not updated. Memory networks cover a wide class of possible implementations. The components I, G, O and R can potentially use any existing ideas from the machine learning literature, e.g., make use of your favorite models (SVMs, decision trees, etc.).

I **component:** Component I can make use of standard pre-processing, e.g., parsing, coreference and entity resolution for text inputs. It could also encode the input into an internal feature representation, e.g., convert from text to a sparse or dense feature vector.

G **component:** The simplest form of G is to store $I(x)$ in a "slot" in the memory:

$$\mathbf{m}_{H(x)} = I(x), \tag{1}$$

where $H(.)$ is a function selecting the slot. That is, G updates the index $H(x)$ of $\mathbf{m}$, but all other parts of the memory remain untouched. More sophisticated variants of G could go back and update earlier stored memories (potentially, all memories) based on the new evidence from the current input x. If the input is at the character or word level one could group inputs (i.e., by segmenting them into chunks) and store each chunk in a memory slot.

If the memory is huge (e.g., consider all of Freebase or Wikipedia) one needs to organize the memories. This can be achieved with the slot choosing function H just described: for example, it could be designed, or trained, to store memories by entity or topic. Consequently, for efficiency at scale, G (and O) need not operate on all memories: they can operate on only a retrieved subset of candidates (only operating on memories that are on the right topic). We explore a simple variant of this in our experiments.

If the memory becomes full, a procedure for "forgetting" could also be implemented by H as it chooses which memory is replaced, e.g., H could score the utility of each memory, and overwrite the least useful. We have not explored this experimentally yet.

O **and** R **components:** The O component is typically responsible for reading from memory and performing inference, e.g., calculating what are the relevant memories to perform a good response. The R component then produces the final response given O. For example in a question answering setup O finds relevant memories, and then R produces the actual wording of the answer, e.g., R could be an RNN that is conditioned on the output of O. Our hypothesis is that without conditioning on such memories, such an RNN will perform poorly.

Pretty similar to attentiion

3 A MemNN Implementation For Text

One particular instantiation of a memory network is where the components are neural networks. We refer to these as memory neural networks (MemNNs). In this section we describe a relatively simple implementation of a MemNN with textual input and output.

3.1 BASIC MODEL

In our basic architecture, the I module takes an input text. Let us first assume this to be a sentence: either the statement of a fact, or a question to be answered by the system (later we will consider word-based input sequences). The text is stored in the next available memory slot in its original form[2], i.e., $S(x)$ returns the next empty memory slot N: $\mathbf{m}_N = x$, $N = N + 1$. The G module is thus only used to store this new memory, so old memories are not updated. More sophisticated models are described in subsequent sections.

The core of inference lies in the O and R modules. The O module produces output features by finding k supporting memories given x. We use k up to 2, but the procedure is generalizable to larger k. For $k = 1$ the highest scoring supporting memory is retrieved with:

$$o_1 = O_1(x, \mathbf{m}) = \arg\max_{i=1,\ldots,N} s_O(x, \mathbf{m}_i) \tag{2}$$

where s_O is a function that scores the match between the pair of sentences x and $\mathbf{m}_i$. For the case $k = 2$ we then find a second supporting memory given the first found in the previous iteration:

$$o_2 = O_2(x, \mathbf{m}) = \arg\max_{i=1,\ldots,N} s_O([x, \mathbf{m}_{o_1}], \mathbf{m}_i) \tag{3}$$

where the candidate supporting memory $\mathbf{m}_i$ is now scored with respect to both the original input and the first supporting memory, where square brackets denote a list[3]. The final output o is $[x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}]$, which is input to the module R.

Finally, R needs to produce a textual response r. The simplest response is to return $\mathbf{m}_{o_k}$, i.e., to output the previously uttered sentence we retrieved. To perform true sentence generation, one can instead employ an RNN. In our experiments we also consider an easy to evaluate compromise approach where we limit textual responses to be a single word (out of all the words seen by the model) by ranking them:

$$r = \text{argmax}_{w \in W} \ s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], w) \tag{4}$$

where W is the set of all words in the dictionary, and s_R is a function that scores the match.

An example task is given in Figure 1. In order to answer the question $x =$ "Where is the milk now?", the O module first scores all memories, i.e., all previously seen sentences, against x to retrieve the most relevant fact, $\mathbf{m}_{o_1} =$ "Joe left the milk" in this case. Then, it would search the memory again to find the second relevant fact given $[x, \mathbf{m}_{o_1}]$, that is $\mathbf{m}_{o_2} =$ "Joe travelled to the office" (the last place Joe went before dropping the milk). Finally, the R module using eq. (4) would score words given $[x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}]$ to output $r =$ "office".

In our experiments, the scoring functions s_O and s_R have the same form, that of an embedding model:

$$s(x, y) = \Phi_x(x)^\top U^\top U \Phi_y(y). \tag{5}$$

where U is a $n \times D$ matrix where D is the number of features and n is the embedding dimension. The role of Φ_x and Φ_y is to map the original text to the D-dimensional feature space. The simplest feature space to choose is a bag of words representation, we choose $D = 3|W|$ for s_O, i.e., every word in the dictionary has three different representations: one for $\Phi_y(.)$ and two for $\Phi_x(.)$ depending on whether the words of the input arguments are from the actual input x or from the supporting memories so that they can be modeled differently.[4] Similarly, we used $D = 3|W|$ for s_R as well. s_O and s_R use different weight matrices U_O and U_R.

---

[2]Technically, we will be using an embedding model to represent text, so we could store the incoming input using its learned embedding vector in memory instead. The downside of such a choice is that during learning the embedding parameters are changing, and hence the stored vectors would go stale. However, at test time (where the parameters are not changing) storing as embedding vectors could make sense, as this is faster than reading the original words and then embedding them repeatedly.

[3]As we will use a bag-of-words model where both x and $\mathbf{m}_{o_1}$ are represented in the bag (but with two different dictionaries) this is equivalent to using the sum $s_O(x, \mathbf{m}_i) + s_O(\mathbf{m}_{o_1}, \mathbf{m}_i)$, however a more sophisticated modeling of the inputs (e.g., with nonlinearities) may not separate into a sum.

[4]Experiments with only a single dictionary and linear embeddings performed worse (not shown). In order to model with only a single dictionary, one could consider deeper networks that transform the words dependent on their context. We leave this to future work.

Figure 1: Example "story" statements, questions and answers generated by a simple simulation. Answering the question about the location of the milk requires comprehension of the actions "picked up" and "left". The questions also require comprehension of the time elements of the story, e.g., to answer "where was Joe before the office?".

> 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

Training We train in a fully supervised setting where we are given desired inputs and responses, and the supporting sentences are labeled as such in the training data (but not in the test data, where we are given only the inputs). That is, during training we know the best choice of both max functions in eq. (2) and (3)[5]. Training is then performed with a margin ranking loss and stochastic gradient descent (SGD). Specifically, for a given question x with true response r and supporting sentences $\mathbf{m}_{o_1}$ and $\mathbf{m}_{o_2}$ (when $k = 2$), we minimize over model parameters U_O and U_R:

$$\sum_{\bar{f} \neq \mathbf{m}_{o_1}} \max(0, \gamma - s_O(x, \mathbf{m}_{o_1}) + s_O(x, \bar{f})) + \tag{6}$$

$$\sum_{\bar{f}' \neq \mathbf{m}_{o_2}} \max(0, \gamma - s_O([x, \mathbf{m}_{o_1}], \mathbf{m}_{o_2}) + s_O([x, \mathbf{m}_{o_1}], \bar{f}')) + \tag{7}$$

$$\sum_{\bar{r} \neq r} \max(0, \gamma - s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], r) + s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], \bar{r})) \tag{8}$$

where $\bar{f}$, $\bar{f}'$ and $\bar{r}$ are all other choices than the correct labels, and γ is the margin. At every step of SGD we sample $\bar{f}, \bar{f}', \bar{r}$ rather than compute the whole sum for each training example, following e.g., Weston et al. (2011).

In the case of employing an RNN for the R component of our MemNN (instead of using a single word response as above) we replace the last term with the standard log likelihood used in a language modeling task, where the RNN is fed the sequence $[x, o_1, o_2, r]$. At test time we output its prediction r given $[x, o_1, o_2]$. In contrast the absolute simplest model, that of using $k = 1$ and outputting the located memory $\mathbf{m}_{o_1}$ as response r, would only use the first term to train.

In the following subsections we consider some extensions of our basic model.

3.2 WORD SEQUENCES AS INPUT

If input is at the word rather than sentence level, that is words arrive in a stream (as is often done, e.g., with RNNs) and not already segmented as statements and questions, we need to modify the approach we have so far described. We hence add a "segmentation" function, to be learned, which takes as input the last sequence of words that have so far not been segmented and looks for breakpoints. When the segmenter fires (indicates the current sequence is a segment) we write that sequence to memory, and can then proceed as before. The segmenter is modeled similarly to our other components, as an embedding model of the form:

$$seg(c) = W_{seg}^{\top} U_S \Phi_{seg}(c) \tag{9}$$

where W_{seg} is a vector (effectively the parameters of a linear classifier in embedding space), and c is the sequence of input words represented as bag of words using a separate dictionary. If $seg(c) > \gamma$, where γ is the margin, then this sequence is recognised as a segment. In this way, our MemNN has a learning component in its write operation. We consider this segmenter a first proof of concept: of course, one could design something much more sophisticated. Further details on the training mechanism are given in Appendix B.

---

[5] However, note that methods like RNNs and LSTMs cannot easily use this information.

3.3 EFFICIENT MEMORY VIA HASHING

If the set of stored memories is very large it is prohibitively expensive to score all of them as in equations (2) and (3). Instead we explore hashing tricks to speed up lookup: hash the input $I(x)$ into one or more buckets and then only score memories $\mathbf{m}_i$ that are in the same buckets. We investigated two ways of doing hashing: (i) via hashing words; and (ii) via clustering word embeddings. For (i) we construct as many buckets as there are words in the dictionary, then for a given sentence we hash it into all the buckets corresponding to its words. The problem with (i) is that a memory $\mathbf{m}_i$ will only be considered if it shares at least one word with the input $I(x)$. Method (ii) tries to solve this by clustering instead. After training the embedding matrix U_O, we run K-means to cluster word vectors $(U_O)_i$, thus giving K buckets. We then hash a given sentence into all the buckets that its individual words fall into. As word vectors tend to be close to their synonyms, they cluster together and we thus also will score those similar memories as well. Exact word matches between input and memory will still be scored by definition. Choosing K controls the speed-accuracy trade-off.

3.4 MODELING WRITE TIME

We can extend our model to take into account *when* a memory slot was written to. This is not important when answering questions about fixed facts ("What is the capital of France?") but is important when answering questions about a story, see e.g., Figure 1. One obvious way to implement this is to add extra features to the representations Φ_x and Φ_y that encode the index j of a given memory $\mathbf{m}_j$, assuming that j follows write time (i.e., no memory slot rewriting). However, that requires dealing with absolute rather than relative time. We had more success empirically with the following procedure: instead of scoring input, candidate pairs with s as above, learn a function on triples $s_{O_t}(x, y, y')$:

$$s_{O_t}(x, y, y') = \Phi_x(x)^\top U_{O_t}^{\ \top} U_{O_t} \Big(\Phi_y(y) - \Phi_y(y') + \Phi_t(x, y, y') \Big). \qquad (10)$$

$\Phi_t(x, y, y')$ uses three new features which take on the value 0 or 1: whether x is older than y, x is older than y', and y older than y'. (That is, we extended the dimensionality of all the Φ embeddings by 3, and set these three dimensions to zero when not used.) Now, if $s_{O_t}(x, y, y') > 0$ the model prefers y over y', and if $s_{O_t}(x, y, y') < 0$ it prefers y'. The argmax of eq. (2) and (3) are replaced by a loop over memories $i = 1, \ldots, N$, keeping the winning memory (y or y') at each step, and always comparing the current winner to the next memory $\mathbf{m}_i$. This procedure is equivalent to the argmax before if the time features are removed. More details are given in Appendix C.

3.5 MODELING PREVIOUSLY UNSEEN WORDS

Even for humans who have read a lot of text, new words are continuously introduced. For example, the first time the word "Boromir" appears in Lord of The Rings (Tolkien, 1954). How should a machine learning model deal with this? Ideally it should work having seen only one example. A possible way would be to use a language model: given the neighboring words, predict what the word should be, and assume the new word is similar to that. Our proposed approach takes this idea, but incorporates it into our networks s_O and s_R, rather than as a separate step.

Concretely, for each word we see, we store a bag of words it has co-occurred with, one bag for the left context, and one for the right. Any unknown word can be represented with such features. Hence, we increase our feature representation D from $3|W|$ to $5|W|$ to model these contexts ($|W|$ features for each bag). Our model learns to deal with new words during training using a kind of "dropout" technique: $d\%$ of the time we pretend we have not seen a word before, and hence do not have a n-dimensional embedding for that word, and represent it with the context instead.

3.6 EXACT MATCHES AND UNSEEN WORDS

Embedding models cannot efficiently use exact word matches due to the low dimensionality n. One solution is to score a pair x, y with

$$\Phi_x(x)^\top U^\top U \Phi_y(y) + \lambda \Phi_x(x)^\top \Phi_y(y) \qquad (11)$$

instead. That is, add the "bag of words" matching score to the learned embedding score (with a mixing parameter λ). Another, related way, that we propose is to stay in the n-dimensional embedding space, but to extend the feature representation D with *matching features*, e.g., one per

word. A matching feature indicates if a word occurs in both x and y. That is, we score with $\Phi_x(x)^\top U^\top U \Phi_y(y, x)$ where Φ_y is actually built conditionally on x: if some of the words in y match the words in x we set those matching features to 1. Unseen words can be modeled similarly by using matching features on their context words. This then gives a feature space of $D = 8|W|$.

4 RELATED WORK

Classical QA methods use a set of documents as a kind of memory, and information retrieval methods to find answers, see e.g., (Kolomiyets & Moens, 2011) and references therein. More recent methods try instead to create a graph of facts – a knowledge base (KB) – as their memory, and map questions to logical queries (Berant et al., 2013; 2014). Neural network and embedding approaches have also been recently explored (Bordes et al., 2014a; Iyyer et al., 2014; Yih et al., 2014). Compared to recent knowledge base approaches, memory networks differ in that they do not apply a two-stage strategy: (i) apply information extraction principles first to build the KB; followed by (ii) inference over the KB. Instead, extraction of useful information to answer a question is performed on-the-fly over the memory which can be stored as raw text, as well as other choices such as embedding vectors. This is potentially less brittle as the first stage of building the KB may have already thrown away the relevant part of the original data.

Classical neural network memory models such as associative memory networks aim to provide content-addressable memory, i.e., given a key vector to output a value vector, see e.g., Haykin (1994) and references therein. Typically this type of memory is distributed across the whole network of weights of the model rather than being compartmentalized into memory locations. Memory-based learning such as nearest neighbor, on the other hand, does seek to store all (typically labeled) examples in compartments in memory, but only uses them for finding closest labels. Memory networks combine compartmentalized memory with neural network modules that can learn how to (potentially successively) read and write to that memory, e.g., to perform reasoning they can iteratively read salient facts from the memory.

However, there are some notable models that have attempted to include memory read and write operations from the 90s. In particular (Das et al., 1992) designed differentiable push and pop actions called a neural network pushdown automaton. The work of Schmidhuber (1992) incorporated the concept of two neural networks where one has very fast changing weights which can potentially be used as memory. Schmidhuber (1993) proposed to allow a network to modify its own weights "self-referentially" which can also be seen as a kind of memory addressing. Finally two other relevant works are the DISCERN model of script processing and memory (Miikkulainen, 1990) and the NARX recurrent networks for modeling long term dependencies (Lin et al., 1996).

Our work was submitted to arxiv just before the Neural Turing Machine work of Graves et al. (2014), which is one of the most relevant related methods. Their method also proposes to perform (sequence) prediction using a "large, addressable memory" which can be read and written to. In their experiments, the memory size was limited to 128 locations, whereas we consider much larger storage (up to 14M sentences). The experimental setups are notably quite different also: whereas we focus on language and reasoning tasks, their paper focuses on problems of sorting, copying and recall. On the one hand their problems require considerably more complex models than the memory network described in Section 3. On the other hand, their problems have known algorithmic solutions, whereas (non-toy) language problems do not.

There are other recent related works. RNNSearch (Bahdanau et al., 2014) is a method of machine translation that uses a learned alignment mechanism over the input sentence representation while predicting an output in order to overcome poor performance on long sentences. The work of (Graves, 2013) performs handwriting recognition by dynamically determining "an alignment between the text and the pen locations" so that "it learns to decide which character to write next". One can view these as particular variants of memory networks where in that case the memory only extends back a single sentence or character sequence.

Table 1: Results on the large-scale QA task of (Fader et al., 2013).

Method	F1
(Fader et al., 2013)	0.54
(Bordes et al., 2014b)	0.73
MemNN (embedding only)	0.72
MemNN (with BoW features)	0.82

Table 2: Memory hashing results on the large-scale QA task of (Fader et al., 2013).

Method	Embedding F1	Embedding + BoW F1	Candidates (speedup)
MemNN (no hashing)	0.72	0.82	14M (0x)
MemNN (word hash)	0.63	0.68	13k (1000x)
MemNN (cluster hash)	0.71	0.80	177k (80x)

5 EXPERIMENTS

5.1 LARGE-SCALE QA

We perform experiments on the QA dataset introduced in Fader et al. (2013). It consists of 14M statements, stored as (subject, relation, object) triples, which are stored as memories in the MemNN model. The triples are *REVERB* extractions mined from the ClueWeb09 corpus and cover diverse topics such as *(milne, authored, winnie-the-pooh)* and *(sheep, be-afraid-of, wolf)*. Following Fader et al. (2013) and Bordes et al. (2014b), training combines pseudo-labeled QA pairs made of a question and an associated triple, and 35M pairs of paraphrased questions from WikiAnswers like *"Who wrote the Winnie the Pooh books?"* and *"Who is poohs creator?"*.

We performed experiments in the framework of re-ranking the top returned candidate answers by several systems measuring F1 score over the test set, following Bordes et al. (2014b). These answers have been annotated as right or wrong by humans, whereas other answers are ignored at test time as we do not know their label. We used a MemNN model of Section 3 with a $k = 1$ supporting memory, which ends up being similar to the approach of Bordes et al. (2014b).[6] We also tried adding the bag of words features of Section 3.6 as well. Time and unseen word modeling were not used. Results are given in Table 1. The results show that MemNNs are a viable approach for large scale QA in terms of performance. However, lookup is linear in the size of the memory, which with 14M facts is slow. We therefore implemented the memory hashing techniques of Section 3.3 using both hashing of words and clustered embeddings. For the latter we tried $K = 1000$ clusters. The results given in Table 2 show that one can get significant speedups (∼80x) while maintaining similar performance using the cluster-based hash. The string hash on the other hand loses performance (whilst being a lot faster) because answers which share no words are now no longer matched.

5.2 SIMULATED WORLD QA

Similar to the approach of Bordes et al. (2010) we also built a simple simulation of 4 characters, 3 objects and 5 rooms – with characters moving around, picking up and dropping objects. The actions are transcribed into text using a simple automated grammar, and labeled questions are generated in a similar way. This gives a QA task on simple "stories" such as in Figure 1. The overall difficulty of the task is that multiple statements have to be used to do inference when asking where an object is, e.g. to answer where is the milk in Figure 1 one has to understand the meaning of the actions "picked up" and "left" and the influence of their relative order. We generated 7k statements and 3k questions from the simulator for training,[7] and an identical number for testing and compare MemNNs to RNNs and LSTMs (long short term memory RNNs (Hochreiter & Schmidhuber, 1997)) on this task. To

---

[6]We use a larger 128 dimension for embeddings, and no fine tuning, hence the result of MemNN slightly differs from those reported in Bordes et al. (2014b).

[7]Learning curves with different numbers of training examples are given in Appendix D.

Table 3: Test accuracy on the simulation QA task.

Method	Difficulty 1			Difficulty 5	
	actor w/o before	actor	actor+object	actor	actor+object
RNN	100%	60.9%	27.9%	23.8%	17.8%
LSTM	100%	64.8%	49.1%	35.2%	29.0%
MemNN $k = 1$	97.8%	31.0%	24.0%	21.9%	18.5%
MemNN $k = 1$ (+time)	99.9%	60.2%	42.5%	60.8%	44.4%
MemNN $k = 2$ (+time)	100%	100%	100%	100%	99.9%

test with sequences of words as input (Section 3.2) the statements are joined together again with a simple grammar[8], to produce sentences that may contain multiple statements, see e.g., Figure 2.

We control the complexity of the task by setting a limit on the number of time steps in the past the entity we ask the question about was last mentioned. We try two experiments: using a limit of 1, and of 5, i.e., if the limit is 5 then we pick a random sentence between 1-5 time steps in the past. If this chosen sentence only mentions an actor, e.g., "Bill is in the kitchen" then we generate the question "where is Bill?" or "where was Bill before the kitchen?". If the sentence mentions an object, e.g., "Bill dropped the football" then we ask the question "where is the football?". For the answers we consider two options: (i) single word answers; and (ii) a simple grammar for generating true answers in sentence form, e.g., "kitchen" for (i) and "He is in the kitchen I believe" (and other variants) for (ii). More details on the dataset generation are given in Appendix A. Note that in the object case the supporting statements necessary to deduce the answer may not lie in the last 5 sentences, e.g., in this example the answer depends on other sentences to find out where Bill actually was when he dropped the football. In fact, in the dataset we generated necessary supporting statements can be up to 65 sentences before (but are usually closer). For that reason, we also conducted two further types of experiments: where we only ask questions about actors (easier) and about actors and objects (harder). We also consider the actor-based questions without the "before" questions for the simplest possible task (i.e. "where is Bill?" but not "where was Bill before the kitchen?" questions).

Figure 2: Sample test set predictions (in red) for the simulation in the setting of word-based input and where answers are sentences and an LSTM is used as the R component of the MemNN.

> Joe went to the garden then Fred picked up the milk; Joe moved to the bathroom and Fred dropped the milk, and then Dan moved to the living_room.
> Where is Dan? A: living room I believe
> Where is Joe? A: the bathroom
> Fred moved to the bedroom and Joe went to the kitchen then Joe took the milk there and Dan journeyed to the bedroom; Joe discarded the milk.
> Where is the milk now ? A: the milk is in the kitchen
> Where is Dan now? A: I think he is in the bedroom
> Joe took the milk there, after that Mike travelled to the office, then Joe went to the living_room, next Dan went back to the kitchen and Joe travelled to the office.
> Where is Joe now? A: I think Joe is in the office

For the baseline RNN and LSTM systems we perform language modeling with backpropagation through time (Mikolov et al., 2010), but where we backprop only on answer words[9]. We optimized the hyperparameters: size of the hidden layer, bptt steps, and learning rate for each dataset. For MemNNs we fixed the embedding dimension to 100, learning rate to 0.01 and margin γ to 0.1 and 10 epochs of training in all experiments.

Results The results for the single word answer setting (i) are given in Table 3. For the actor-only tasks, RNN and LSTMs solve the simpler difficulty level 1 task *without* before questions ("w/o

---

[8]We also tried the same kind of experiments with sentence-level rather than word-sequence input, without joining sentences, giving results with similar overall conclusions, see Appendix E.

[9]We tried using standard language modeling on the questions as well, with slightly worse results.

Figure 3: An example story with questions correctly answered by a MemNN. The MemNN was trained on the simulation described in Section 5.2 and had never seen many of these words before, e.g., Bilbo, Frodo and Gollum.

> Bilbo travelled to the cave. Gollum dropped the ring there. Bilbo took the ring.
> Bilbo went back to the Shire. Bilbo left the ring there. Frodo got the ring.
> Frodo journeyed to Mount-Doom. Frodo dropped the ring there. Sauron died.
> Frodo went back to the Shire. Bilbo travelled to the Grey-havens. The End.
> Where is the ring? A: Mount-Doom
> Where is Bilbo now? A: Grey-havens
> Where is Frodo now? A: Shire

Figure 4: An example dialogue with a MemNN system trained on both the simulation data and the large-scale QA data. The system is able to (attempt to) answer questions about general world knowledge and about specific story-based statements in the dialogue.

> Fred went to the kitchen. Fred picked up the milk. Fred travelled to the office.
> Where is the milk ? A: office
> Where does milk come from ? A: milk come from cow
> What is a cow a type of ? A: cow be female of cattle
> Where are cattle found ? A: cattle farm become widespread in brazil
> What does milk taste like ? A: milk taste like milk
> What does milk go well with ? A: milk go with coffee
> Where was Fred before the office ? A: kitchen

before"), but perform worse *with* before questions, and even worse on the difficulty 5 tasks. This demonstrates that the poor performance of the RNN is due to its failure to encode long(er)-term memory. This would likely deteriorate even further with higher difficulty levels (distances). LSTMs are however better than RNNs, as expected, as they are designed with a more sophisticated memory model, but still have trouble remembering sentences too far in the past. MemNNs do not have this memory limitation and its mistakes are instead due to incorrect usage of its memory, when the wrong statement is picked by s_O. Time features are necessary for good performance on before questions or difficulty > 1 (i.e., when the answer is not in the last statement), otherwise s_O can pick a statement about a person's whereabouts but they have since moved. Finally, results on the harder actor+object task indicate that MemNN also successfully perform 2-stage inference using $k = 2$, whereas MemNNs without such inference (with $k = 1$) and RNNs and LSTMs fail.

We also tested MemNNs in the multi-word answer setting (ii) with similar results, whereby MemNNs outperform RNNs and LSTMs, which are detailed in Appendix F. Example test prediction output demonstrating the model in that setting is given in Figure 2.

5.2.1 QA WITH PREVIOUSLY UNSEEN WORDS

We then tested the ability of MemNNs to deal with previously unseen words at test time using the unseen word modeling approach of Sections 3.5 and 3.6. We trained the MemNN on the same simulated dataset as before and test on the story given in Figure 3. This story is generated using similar structures as in the simulation data, except that the nouns are unknowns to the system at training time. Despite never seeing any of the Lord of The Rings specific words before (e.g., Bilbo, Frodo, Sauron, Gollum, Shire and Mount-Doom), MemNNs are able to correctly answer the questions.

MemNNs can discover simple linguistic patterns based on verbal forms such as (X, dropped, Y), (X, took, Y) or (X, journeyed to, Y) and can successfully generalize the meaning of their instantiations using unknown words to perform 2-stage inference. Without the unseen word modeling described in Section 3.5, they completely fail on this task.

5.3 COMBINING SIMULATED DATA AND LARGE-SCALE QA

Combining simulated world learning with real-world data might be one way to show the power and generality of the models we design. We implemented a naive setup towards that goal: we took the two models from Sections 5.1 and 5.2, trained on large-scale QA and simulated data respectively, and built an ensemble of the two. We present the input to both systems and then for each question simply output the response of the two choices with the highest score. This allows us to perform simple dialogues with our combined MemNN system. The system is then capable of answering both general knowledge questions and specific statements relating to the previous dialogue. An example dialogue trace is given in Fig. 4. Some answers appear fine, whereas others are nonsensical. Future work should combine these models more effectively, for example by multitasking directly the tasks with a single model.

6 CONCLUSIONS AND FUTURE WORK

In this paper we introduced a powerful class of models, memory networks, and showed one instantiation for QA. Future work should develop MemNNs for text further, evaluating them on harder QA and open-domain machine comprehension tasks (Richardson et al., 2013). For example, large scale QA tasks that require multi-hop inference such as WebQuestions should also be tried Berant et al. (2013). More complex simulation data could also be constructed in order to bridge that gap, e.g., requiring coreference, involving more verbs and nouns, sentences with more structure and requiring more temporal and causal understanding. More sophisticated architectures should also be explored in order to deal with these tasks, e.g., using more sophisticated memory management via G and more sophisticated sentence representations. Weakly supervised settings are also very important, and should be explored, as many datasets only have supervision in the form of question answer pairs, and not supporting facts as well as we used here. Finally, we believe this class of models is much richer than the one specific variant we detail here, and that we have currently only explored one specific variant of memory networks. Memory networks should be applied to other text tasks, and other domains, such as vision, as well.

REFERENCES

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Berant, Jonathan, Chou, Andrew, Frostig, Roy, and Liang, Percy. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, pp. 1533–1544, 2013.

Berant, Jonathan, Srikumar, Vivek, Chen, Pei-Chun, Huang, Brad, Manning, Christopher D, Van-der Linden, Abby, Harding, Brittany, and Clark, Peter. Modeling biological processes for reading comprehension. In *Proc. EMNLP*, 2014.

Bordes, Antoine, Usunier, Nicolas, Collobert, Ronan, and Weston, Jason. Towards understanding situated natural language. In *AISTATS*, 2010.

Bordes, Antoine, Chopra, Sumit, and Weston, Jason. Question answering with subgraph embeddings. In *Proc. EMNLP*, 2014a.

Bordes, Antoine, Weston, Jason, and Usunier, Nicolas. Open question answering with weakly supervised embedding models. *ECML-PKDD*, 2014b.

Das, Sreerupa, Giles, C Lee, and Sun, Guo-Zheng. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*, 1992.

Fader, Anthony, Zettlemoyer, Luke, and Etzioni, Oren. Paraphrase-driven learning for open question answering. In *ACL*, pp. 1608–1618, 2013.

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Haykin, Simon. Neural networks: A comprehensive foundation. 1994.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Iyyer, Mohit, Boyd-Graber, Jordan, Claudino, Leonardo, Socher, Richard, and III, Hal Daumé. A neural network for factoid question answering over paragraphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 633–644, 2014.

Kolomiyets, Oleksandr and Moens, Marie-Francine. A survey on question answering technology from an information retrieval perspective. *Information Sciences*, 181(24):5412–5434, 2011.

Lin, Tsungnam, Horne, Bil G, Tiňo, Peter, and Giles, C Lee. Learning long-term dependencies in narx recurrent neural networks. *Neural Networks, IEEE Transactions on*, 7(6):1329–1338, 1996.

Miikkulainen, Risto. {DISCERN}:{A} distributed artificial neural network model of script processing and memory. 1990.

Mikolov, Tomas, Karafiát, Martin, Burget, Lukas, Cernockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *Interspeech*, pp. 1045–1048, 2010.

Richardson, Matthew, Burges, Christopher JC, and Renshaw, Erin. Mctest: A challenge dataset for the open-domain machine comprehension of text. In *EMNLP*, pp. 193–203, 2013.

Schmidhuber, Jürgen. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Schmidhuber, Jürgen. A self-referentialweight matrix. In *ICANN93*, pp. 446–450. Springer, 1993.

Tolkien, John Ronald Reuel. *The Fellowship of the Ring*. George Allen & Unwin, 1954.

Weston, Jason, Bengio, Samy, and Usunier, Nicolas. Wsabie: Scaling up to large vocabulary image annotation. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pp. 2764–2770. AAAI Press, 2011.

Yih, Wen-Tau, He, Xiaodong, and Meek, Christopher. Semantic parsing for single-relation question answering. In *Proceedings of ACL*. Association for Computational Linguistics, June 2014. URL http://research.microsoft.com/apps/pubs/default.aspx?id=214353.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

A   SIMULATION DATA GENERATION

Aim   We have built a simple simulation which behaves much like a classic text adventure game. The idea is that generating text within this simulation allows us to ground the language used.

Some comments about our intent:

- Firstly, while this currently only encompasses a very small part of the kind of language and understanding we want a model to learn to move towards full language understanding, we believe it is a prerequisite that models should perform well on this kind of task for them to work on real-world environments.
- Secondly, our aim is to make this simulation more complex and to release improved versions over time. Hopefully it can then scale up to evaluate more and more useful properties.

Currently, tasks within the simulation are restricted to question answering tasks about the location of people and objects. However, we envisage other tasks should be possible, including asking the learner to perform actions within the simulation ("Please pick up the milk", "Please find John and give him the milk") and asking the learner to describe actions ("What did John just do?").

Actions   The underlying actions in the simulation consist of the following:

go <location>,   *get <object>*,   *get <object1> from <object2>*,
put <object1> in/on <object2>,   *give <object> to <actor>*,
drop <object>,   *look*,   *inventory*,   *examine <object>*.

There are a set of constraints on those actions. For example an actor cannot get something that they or someone else already has, they cannot go to a place they are already at, cannot drop something they do not already have, and so on.

Executing Actions and Asking Questions   Using the underlying actions and their constraints, there is then a (hand-built) model that defines how actors act. Currently this is very simple: they try to make a random valid action, at the moment restricted to go *or* go, get and drop depending on the which of two types of experiments we are running: (i) actor; or (ii) actor + object.

If we write these actions down in text form this gives us a very simple "story" which is executable by the simulation, e.g., *joe go kitchen; fred go kitchen; joe get milk; joe go office; joe drop milk; joe go bathroom*. This example corresponds to the story given in Figure 1. The system can then ask questions about the state of the simulation e.g., *where milk?*, *where joe?*, *where joe before office?* It is easy to calculate the true answers for these questions as we have access to the underlying world. What remains is to convert both the statements and the questions to look more like natural language.

Simple Grammar For Generating Language   In order to produce more natural looking text with lexical variety we built a simple automated grammar. Each verb is assigned a set of synonyms, e.g., the simulation command *get* is replaced with either *picked up*, *got*, *grabbed* or *took*, and *drop* is replace with either *dropped*, *left*, *discarded* or *put down*. Similarly, each object and actor can have a set of replacement synonyms as well, although currently there is no ambiguity there in our experiments, we simply add articles or not. We do add lexical variation to questions, e.g., "Where is John ?" or "Where is John now ?".

Joining Statements   Finally, for the word sequence training setting, we join the statements above into compound sentences. To do this we simply take the set of statements and then join them randomly with one of the following: ".", "*and*", "*then*", ", *then*", ";", ", *later*", ", *after that*", ", *and then*", or ", next". Example output can be seen in Figure 2.

Issues   There are a great many aspects of language not yet modeled. For example, currently coreference is not modeled (e.g., "He picked up the milk") and similarly there are no compound noun phrases ("John and Fred went to the kitchen"). Some of these seem easy to add to the simulation. The hope is that adding these complexities will help evaluate models in a controlled way, within the simulated environment, which is hard to do with real data. Of course, this is not a substitute for real data which our models should be applied to as well, but does serve as a useful testbed.

B WORD SEQUENCE TRAINING

For segmenting an input word stream as generated in Appendix A we use a segmenter of the form:

$$seg(c) = W_{seg}^\top U_S \Phi_{seg}(c)$$

where W_{seg} is a vector (effectively the parameters of a linear classifier in embedding space). As we are already in the fully supervised setting, where for each question in the training set we are given the answer and the supporting facts from the input stream, we can also use that supervision for the segmenter as well. That is, for any known supporting fact, such as "Bill is in the Kitchen" for the question "Where is Bill?" we wish the segmenter to fire for such a statement, but not for unfinished statements such as "Bill is in the". We can thus write our training criterion for segmentation as the minimization of:

$$\sum_{f \in \mathcal{F}} \max(0, \gamma - seg(f)) + \sum_{\bar{f} \in \bar{\mathcal{F}}} \max(0, \gamma + seg(\bar{f})) \tag{12}$$

where $\mathcal{F}$ are all known supporting segments in the labeled training set, and $\bar{\mathcal{F}}$ are all other segments in the training set.

C WRITE TIME FEATURE TRAINING

The training procedure to take into account modeling write time is slightly different to that described in Section 3.1. Write time features are important so that the MemNN knows when each memory was written, and hence knows the ordering of statements that comprise a story or dialogue. Note that this is different to time information described in the text of a statement, such as the tense of a statement, or statements containing time expressions, e.g., "He went to the office yesterday". For such cases, write time features are not directly necessary, and they could (potentially) be modeled directly from the text.

As was described in Section 3.4 we add three write time features to the model and score triples using:

$$s_{O_t}(x, y, y') = \Phi_x(x)^\top U_{O_t}{}^\top U_{O_t}\Big(\Phi_y(y) - \Phi_y(y') + \Phi_t(x, y, y')\Big). \tag{13}$$

If $s_O(x, y, y') > 0$ the model prefers y over y', and if $s_O(x, y, y') < 0$ it prefers y'. The argmax of eq. (2) and (3) are replaced by a loop over memories $i = 1, \ldots, N$, keeping the winning memory (y or y') at each step, and always comparing the current winner to the next memory $\mathbf{m}_i$. That is, at inference time, for a $k = 2$ model the $\arg\max$ functions of eq. (2) and (3) are replaced with $o_1 = O_t(x, \mathbf{m})$ and $o_2 = O_t([x, \mathbf{m}_{o_1}], \mathbf{m})$ where O_t is defined in Algorithm 1 below.

Algorithm 1 O_t replacement to $\arg\max$ when using write time features

> **function** $O_t(q, \mathbf{m})$
> $t \leftarrow 1$
>   **for** $i = 2, \ldots, N$ **do**
>   **if** $s_{O_t}(q, \mathbf{m}_i, \mathbf{m}_t) > 0$ **then**
> $t \leftarrow i$
>   **end if**
>   **end for**
>   **return** t
> **end function**

$\Phi_t(x, y, y')$ uses three new features which take on the value 0 or 1: whether x is older than y, x is older than y', and y older than y'. When finding the second supporting memory (computing $O_t([x, \mathbf{m}_{o_1}], \mathbf{m})$) we encode whether $\mathbf{m}_{o_1}$ is older than y, $\mathbf{m}_{o_1}$ is older than y', and y older than y' to capture the relative age of the first supporting memory w.r.t. the second one in the first two features. Note that when finding the first supporting memory (i.e., for $O_t(x, \mathbf{m})$) the first two features are useless as x is the last thing in the memory and hence y and y' are always older.

To train our model with write time features we need to replace the hinge loss in eqs. (6)-(7) with a loss that matches Algorithm 1. To do this, we instead minimize:

$$\sum_{\bar{f} \neq \mathbf{m}_{o_1}} \max(0, \gamma - s_{O_t}(x, \mathbf{m}_{o_1}, \bar{f})) + \sum_{\bar{f} \neq \mathbf{m}_{o_1}} \max(0, \gamma + s_{O_t}(x, \bar{f}, \mathbf{m}_{o_1})) +$$

$$\sum_{\bar{f}' \neq \mathbf{m}_{o_2}} \max(0, \gamma - s_{O_t}([x, \mathbf{m}_{o_1}], \mathbf{m}_{o_2}, \bar{f}')) + \sum_{\bar{f}' \neq \mathbf{m}_{o_2}} \max(0, \gamma + s_{O_t}([x, \mathbf{m}_{o_1}], \bar{f}', \mathbf{m}_{o_2}) +$$

$$\sum_{\bar{r} \neq r} \max(0, \gamma - s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], r) + s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], \bar{r}]))$$

The last term is the same as in eq. (8) and is for the final ranking of words to return a response, which remains unchanged (as usual, this can also be replaced by an RNN for a more sophisticated model). Terms 1-4 replace eqs. (6)-(7) by considering triples directly. For both $\mathbf{m}_{o_1}$ and $\mathbf{m}_{o_2}$ we need to have two terms considering them as the second or third argument to S_{O_t} as they may appear on either side during inference (via Algorithm 1). As before, at every step of SGD we sample $\bar{f}, \bar{f}', \bar{r}$ rather than compute the whole sum for each training example.

D WORD-SEQUENCE LEARNING CURVE EXPERIMENTS

We computed the test accuracy of MemNNs $k = 2$ (+ time) for varying amounts of training data: 100, 500, 1000 and 3000 training questions. The results are given in Table 4. These results can be compared with RNNs and LSTMs on the full data (3000 examples) by comparing with Figure 3. For example, on the difficulty 5 actor and actor + object tasks MemNNs outperform LSTMs even using 30 times less training examples.

Table 4: Test accuracy of MemNNs $k = 2$ (+time) on the word-sequence simulation QA task for differing numbers of training examples (number of questions).

| Num. training | Difficulty 1 | | Difficulty 5 | |
questions	actor	actor + object	actor	actor + object
100	73.8%	64.9%	74.4%	49.8%
500	99.9%	99.2%	99.8%	95.1%
1000	99.9%	100%	100%	98.4%
3000	100%	100%	100%	99.9%

E SENTENCE-LEVEL EXPERIMENTS

We conducted experiments where input was at the sentence-level, that is the data was already pre-segemented into statements and questions as input to the MemNN (as opposed to being input as a stream of words). Results comparing RNNs with MemNNs are given in Table 5. The conclusions are similar to those at the word level from Section 5.2. That is, MemNNs outperform RNNs, and that inference that finds $k = 2$ supporting statements and time features are necessary for the actor w/o before + object task.

Table 5: Test accuracy on the sentence-level simulation QA task.

| Method | Difficulty 1 | | Difficulty 5 | |
	actor w/o before	actor w/o before + object	actor w/o before	actor w/o before + object
RNN	100%	58%	29%	17%
MemNN $k = 1$	90%	9%	46%	21%
MemNN $k = 1$ (+time)	100%	73%	100%	73%
MemNN $k = 2$ (+time)	100%	99.95%	100%	99.4%

F   MULTI-WORD ANSWER SETTING EXPERIMENTS

We conducted experiments for the simulation data in the case where the answers are sentences (see Appendix A and Figure 2). As the single word answer model can no longer be used, we simply compare MemNNs using either RNNs or LSTMs for the response module R. As baselines we can still use RNNs and LSTMs in the standard setting of being fed words only including the statements and the question as a word stream. In contrast, the MemNN RNN and LSTMs are effectively fed the output of the O module (see Section 3.1). In these experiments we only consider the difficulty 5 actor+object setting in the case of MemNNs with $k = 2$ iterations (eq. (3)), which means the module R is fed the features $[x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}]$ after the modules I, G and O have run.

The sentence generation is performed on the test data, and the evaluation we chose is as follows. A correct generation has to contain the correct location answer, and can optionally contain the subject or a correct pronoun referring to it. For example the question "Where is Bill?" allows the correct answers "Kitchen", "In the kitchen", "Bill is in the kitchen", "He is in the kitchen" and "I think Bill is in the kitchen". However incorrect answers contain an incorrect location or subject reference, for example "Joe is in the kitchen", "It is in the kitchen" or "Bill is in the bathroom I believe". We can then measure the percentage of text examples that are correct using this metric.

The numerical results are given in Table 6, and example output is given in Figure 2. The results indicate that MemNNs with LSTMs perform quite strongly, outperforming MemNNs using RNNs. However, both MemNN variant outperform both RNNs and LSTMs by some distance.

Table 6: Test accuracy on the multi-word answer simulation QA task. We compare conventional RNN and LSTMs with MemNNs using an RNN or LSTM module R (i.e., where R is fed features $[x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}]$ after the modules I, G and O have run).

Model	MemNN: IGO features $[x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}]$	Word features
RNN	68.83%	13.97%
LSTM	90.98%	14.01%