Memory Networks

1 Citation

Sukhbaatar, Sainbayar, Jason Weston, and Rob Fergus. ”End-to-end memory networks.” Advances in neural information processing systems. 2015.

`https://arxiv.org/pdf/1410.3916.pdf`

2 Abstract

A memory network has a large memory (matrix) that it can read and write from. We evaluate our memory network on the text QA task.

3 Introduction

Suppose we want to tell a neural network a story and ask it questions afterwards. An Long Short-Term Memory (LSTM) model's hidden state is not powerful enough to store the information the model needs to solve this problem. Memory networks explictly model reading and writing to a memory.

4 Memory Networks

The network has an input feature map I that turns the raw input into a feature vector, a generalization piece G that updates memories given the input, an output feature map O that selects memories to use to answer the question, and a response piece R that creates the response. The equations for the network are:

$$\mathbf{m}_i = G(\mathbf{m}_i, I(x), \mathbf{m}) \forall i$$
$$o = O(I(x), \mathbf{m}) \text{ and } r = R(o)$$

5 A MemNN Implementation for Text

Our input will be sentences (we'll consider words later). I can be the identity function (or multiplication by an embedding matrix). Our G function just stores memories in the next available slot. Our O function will pick two sentences:

$$o_1 = O_1(x, \mathbf{m}) = \text{argmax}_{i=1 \ldots N} s_O(x, \mathbf{m}_i)$$
$$o_2 = O_2(x, \mathbf{m}) = \text{argmax}_{i=1 \ldots N} s_O([x, \mathbf{m}_{\mathbf{o_1}}], \mathbf{m}_i)$$

where s_O measures the similarity between the two argument sentences (higher means more similar). Our R just returns a single-word response based on a scoring function s_R.

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

Both s_O and s_R take this form (we let ϕ just map sentences to bag-of-words vectors, but you can also use an embedding):

$$s(x, y) = \phi_x(s)^T U^T U \phi_y(y)$$

Our training data indicates both the supporting sentences and the one-word response. Thus, we use margin ranking loss:

$$\begin{aligned}
&\sum_{\bar{f} \neq \mathbf{m}_{o_1}} \max\left(0, \gamma - s_O(x, \mathbf{m}_{o_1}) + s_O(x, \bar{f})\right) \\
&\qquad + \sum_{\bar{f}' \neq \mathbf{m}_{o_2}} \max\left(0, \gamma - s_O([x, \mathbf{m}_{o_1}], \mathbf{m}_{o_2}) + s_O([x, \mathbf{m}_{o_1}], \bar{f})\right) \\
&\qquad\qquad + \sum_{\bar{r} \neq r} \max\left(0, \gamma - s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], r) + s_R([x, \mathbf{m}_{o_1}, \mathbf{m}_{o_2}], \bar{r})\right) \qquad (1)
\end{aligned}$$

If you want to use a mult-word response, just use an RNN to make the output and replace the final term of the loss with log-likelihood of the generated response.

If we take words (instead of sentences) as input, we use a segmenter that operates on a bag-of-words word sequence (c). Whenever it produces an output greater than margin γ, we write to memory:

$$seg(c) = W_{seg}^T U_S \phi_{seg}(c)$$

To avoid making the memory too large, we do k means clustering on the word vectors. Then, a sentence is hashed into each of its words buckets. Then, we can look up memories in buckets that the input's words are in.

We can also store the time when an input was written to memory. We use relative time instead of absolute time with:

$$s_{O_t}(x, y, y') = \phi_x(x)^T U_{O_t}^T U_{O_t}(\phi_y(y) - \phi_y(y') + \phi_t(x, y, y'))$$

For all ϕ, we extend the embedding with three binary element indicating whether x is older than y, x is older than y' and y is older than y'. If $s_{O_t}(x, y, y') > 0$, the model prefers y (otherwise it prefers y'). So, instead of doing an argmax to pick o_1 and o_2, we loop over the memories and keep running track of which memory wins the s_{O_t}.

To ensure our model can handle new words, we modify our feature vector so that it also indicates the left and right context word that the input word occurs with. At train time, we drop the input word with probability $d\%$ so the model can learn how to use the context words if the input word is missing.

When using word embeddings, we can't check if two words are equal because dimensionality is low, so we sum the bag of words matching score and embedding score with mixing weight λ.

$$\phi_x(x)^T U_{O_t}^T U_{O_t} \phi_y(y) + \lambda \phi_x(x)^T \phi_y(y)$$

Another option is to use matching features to indicate if the word appears in both x and y.

6 Related Work

Classical QA answers questions by searching a knowledge base. For memory networks, we don't separate knowledge base building and question answering. We do both at once. We submitted our work a little before the Neural Turing Machine by Graves. Our storage is larger than in NTM. We also focus on QA tasks while Graves looks at copy, sort, etc. RNNSearch uses an attention mechanism to do neural machine translation. Graves also used RNNs for handwriting synthesis.

7 Experiments

We use a dataset mined from the ClueWeb09 corpus. It consists of triples like (milne, authored, winnie-the-pooh). We feed in a set of triples and then ask a question like "Who wrote winnie the pooh?"

We clustered embeddings into $k = 1000$ clusters and hash embeddings (this gives 80x speedup without losing much accuracy).

We also built a little simulator involving 4 characters, 3 object,s and 5 rooms where characters would pick up and put down objects in different rooms. We use a simple grammer to turn the actions into sentences.

For MemNN, we have 100-dimensional embeddings, learning rate 0.01, $\gamma = 0.1$, and 10 training epochs.

We compare against RNN and LSTM - we beat both of them. LSTM does better than RNN, but they struggle to remember sentences from long ago. MemNN only messes up when it uses its memory incorrectly. Also, using two stage (i.e. using top-two memories) approach is better than just one.

To measure how MemNN handles unseen words, we feed it character names from Lord of the Rings. MemNN figures out linguistic patterns and learns how to do well at the task even in this domain.

Our technique of combining synthetic data with real data might be a good way to build a generalizable system.

8 Conclusions and Future Work

MemNN is an end-to-end neural network that knows how to write and read from a large memory. Future work should look at better memory management, better sentence embeddings, and weakly supervised learning.