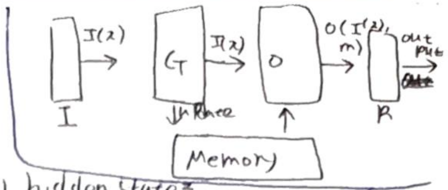
∘ 사전에 알아야 할 내용

∘ memory Network : LSTM과 같이 hidden state를 사용하는 모델은 입력의 시퀀스가 길어진다면 앞부분의 입력들을 적절히 반영하기 어렵 → 이를 해결하기 위해 메모리에 저장할 수 있는 만큼 최대한 각 단계에서의 hidden state를 저장하고 이를 활용 ⇒ Memory Network 필요한 부분



<김성현님 블로그 참고> & <renew 블로그 참고>

1. 임베딩 : $I(x)$
2. 임베딩된 x로 memory update : $m_i = G(m_i, I(x), m)$,
3. input과 memory를 이용해 output o를 계산 : $o = O(I(x), m)$
4. 마지막으로 output을 decode해서 최종 response 만듦 : $r = R(o)$

End-To-End Memory Networks

<<LM, QA와 같은 생성모델에 적합.>>

Sainbayar Sukhbaatar
Dept. of Computer Science
Courant Institute, New York University
sainbar@cs.nyu.edu

Arthur Szlam **Jason Weston** **Rob Fergus**
Facebook AI Research
New York
{aszlam,jase,robfergus}@fb.com

Abstract

We introduce a neural network with a recurrent attention model over a possibly large external memory. The architecture is a form of Memory Network [23] but unlike the model in that work, it is trained end-to-end, and hence requires significantly less supervision during training, making it more generally applicable in realistic settings. → Memory Network (basemodel)과 다른점 (장점) It can also be seen as an extension of RNNsearch [2] to the case where multiple computational steps (hops) are performed per output symbol. The flexibility of the model allows us to apply it to tasks as diverse as (synthetic) question answering [22] and to language modeling. For the former our approach is competitive with Memory Networks, but with less supervision. For the latter, on the Penn TreeBank and Text8 datasets our approach demonstrates comparable performance to RNNs and LSTMs. In both cases we show that the key concept of multiple computational hops yields improved results.

1 Introduction

Two grand challenges in artificial intelligence research have been to build models that can make multiple computational steps in the service of answering a question or completing a task, and models that can describe long term dependencies in sequential data.

Recently there has been a resurgence in models of computation using explicit storage and a notion of attention [23, 8, 2]; manipulating such a storage offers an approach to both of these challenges. In [23, 8, 2], the storage is endowed with a continuous representation; reads from and writes to the storage, as well as other processing steps, are modeled by the actions of neural networks.

In this work, we present a novel recurrent neural network (RNN) architecture where the recurrence reads from a possibly large external memory multiple times before outputting a symbol. Our model can be considered a continuous form of the Memory Network implemented in [23]. The model in → 기존 모델의 문제점 that work was not easy to train via backpropagation, and required supervision at each layer of the network. The continuity of the model we present here means that it can be trained end-to-end from input-output pairs, and so is applicable to more tasks, i.e. tasks where such supervision is not available, such as in language modeling or realistically supervised question answering tasks. Our model → model의 특징 (장점) can also be seen as a version of RNNsearch [2] with multiple computational steps (which we term "hops") per output symbol. We will show experimentally that the multiple hops over the long-term memory are crucial to good performance of our model on these tasks and that training the memory representation can be integrated in a scalable manner into our end-to-end neural network model.

2 Approach

Our model takes a discrete set of inputs $x_1, ..., x_n$ that are to be stored in the memory, a query q, and outputs an answer a. Each of the x_i, q, and a contains symbols coming from a dictionary with V → vocab size 의미하는 듯. words. The model writes all x to the memory up to a fixed buffer size, and then finds a continuous representation for the x and q. The continuous representation is then processed via multiple hops to output a. This allows backpropagation of the error signal through multiple memory accesses back to the input during training.

2.1 Single Layer

We start by describing our model in the single layer case, which implements a single memory hop operation. We then show it can be stacked to give multiple hops in memory.

Input memory representation: Suppose we are given an input set $x_1, ..., x_i$ to be stored in memory. The entire set of $\{x_i\}$ are converted into memory vectors $\{m_i\}$ of dimension d computed by embedding each x_i in a continuous space, in the simplest case, using an embedding matrix A (of size $d \times V$). The query q is also embedded (again, in the simplest case via another embedding matrix B with the same dimensions as A) to obtain an internal state u. In the embedding space, we compute the match between u and each memory m_i by taking the inner product followed by a softmax:

$$p_i = \text{Softmax}(u^T m_i). \tag{1}$$

즉, 문장별 주목할 정도를 측정 (q = query (question))

where $\text{Softmax}(z_i) = e^{z_i} / \sum_j e^{z_j}$. Defined in this way p is a probability vector over the inputs.

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

$$o = \sum_i p_i c_i. \tag{2}$$

Because the function from input to output is smooth, we can easily compute gradients and back-propagate through it. Other recently proposed forms of memory or attention take this approach, notably Bahdanau *et al.* [2] and Graves *et al.* [8], see also [9].

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

$(V \times d)(d \times 1) \to (V \times 1)$

$$\hat{a} = \text{Softmax}(W(o + u)) \tag{3}$$

The overall model is shown in Fig. 1(a). During training, all three embedding matrices A, B and C, as well as W are jointly learned by minimizing a standard cross-entropy loss between $\hat{a}$ and the true label a. Training is performed using stochastic gradient descent (see Section 4.2 for more details).

sentence representation을 만드는 방법은 chap 4 참고하기!
정확히는 4.1

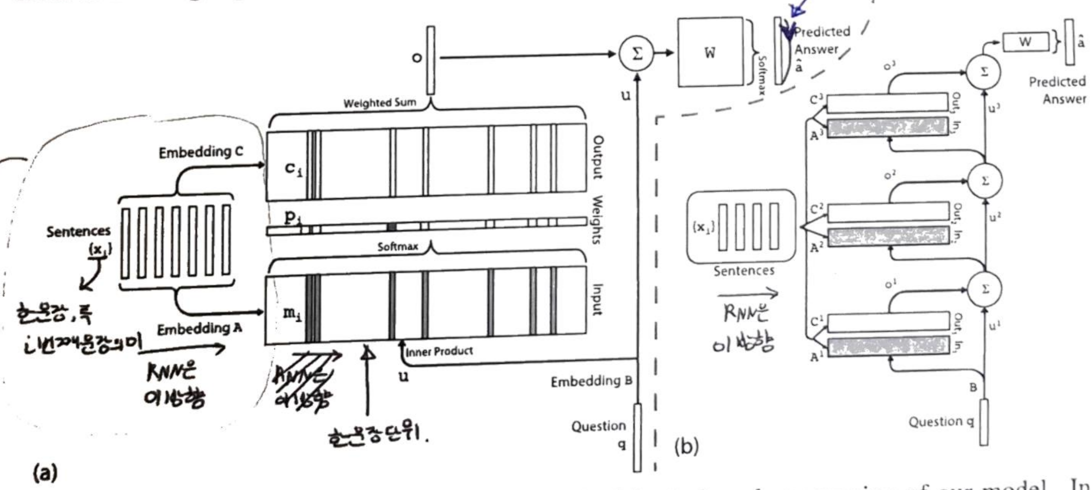


Figure 1: (a): A single layer version of our model. (b): A three layer version of our model. In practice, we can constrain several of the embedding matrices to be the same (see Section 2.2).

2.2 Multiple Layers

We now extend our model to handle K hop operations. The memory layers are stacked in the following way:

- The input to layers above the first is the sum of the output o^k and the input u^k from layer k (different ways to combine o^k and u^k are proposed later):

$$u^{k+1} = u^k + o^k. \tag{4}$$

- Each layer has its own embedding matrices A^k, C^k, used to embed the inputs $\{x_i\}$. However, as discussed below, they are constrained to ease training and reduce the number of parameters.
- At the top of the network, the input to W also combines the input and the output of the top memory layer: $\hat{a} = \text{Softmax}(Wu^{K+1}) = \text{Softmax}(W(o^K + u^K))$.

We explore two types of weight tying within the model:

1. **Adjacent**: the output embedding for one layer is the input embedding for the one above, i.e. $A^{k+1} = C^k$. We also constrain (a) the answer prediction matrix to be the same as the final output embedding, i.e $W^T = C^K$, and (b) the question embedding to match the input embedding of the first layer, i.e. $B = A^1$.
2. **Layer-wise (RNN-like)**: the input and output embeddings are the same across different layers, i.e. $A^1 = A^2 = ... = A^K$ and $C^1 = C^2 = ... = C^K$. We have found it useful to add a linear mapping H to the update of u between hops; that is, $u^{k+1} = Hu^k + o^k$. This mapping is learnt along with the rest of the parameters and used throughout our experiments for layer-wise weight tying.

A three-layer version of our memory model is shown in Fig. 1(b). Overall, it is similar to the Memory Network model in [23], except that the hard max operations within each layer have been replaced with a continuous weighting from the softmax. ㄴ> 기존 모델 대비 장점

~~RNN모델과~~ RNN 모델에 빗대어 자신의 모델 특징 나열

Note that if we use the layer-wise weight tying scheme, our model can be cast as a traditional RNN where we divide the outputs of the RNN into *internal* and *external* outputs. Emitting an internal output corresponds to considering a memory, and emitting an external output corresponds to predicting a label. From the RNN point of view, u in Fig. 1(b) and Eqn. 4 is a hidden state, and the model generates an internal output p (attention weights in Fig. 1(a)) using A. The model then ingests p using C, updates the hidden state, and so on[1]. Here, unlike a standard RNN, we explicitly condition on the outputs stored in memory during the K hops, and we keep these outputs soft, rather than sampling them. Thus our model makes several computational steps before producing an output meant to be seen by the "outside world".

3 Related Work

A number of recent efforts have explored ways to capture long-term structure within sequences using RNNs or LSTM-based models [4, 7, 12, 15, 10, 1]. The memory in these models is the state of the network, which is latent and inherently unstable over long timescales. The LSTM-based models address this through local memory cells which lock in the network state from the past. In practice, the performance gains over carefully trained RNNs are modest (see Mikolov *et al.* [15]). Our model differs from these in that it uses a global memory, with shared read and write functions. However, with layer-wise weight tying our model can be viewed as a form of RNN which only produces an output after a fixed number of time steps (corresponding to the number of hops), with the intermediary steps involving memory input/output operations that update the internal state.

Some of the very early work on neural networks by Steinbuch and Piske[19] and Taylor [21] considered a memory that performed nearest-neighbor operations on stored input vectors and then fit parametric models to the retrieved sets. This has similarities to a single layer version of our model.

Subsequent work in the 1990's explored other types of memory [18, 5, 16]. For example, Das *et al.* [5] and Mozer *et al.* [16] introduced an explicit stack with push and pop operations which has been revisited recently by [11] in the context of an RNN model.

Closely related to our model is the Neural Turing Machine of Graves *et al.* [8], which also uses a continuous memory representation. The NTM memory uses both content and address-based access, unlike ours which only explicitly allows the former, although the temporal features that we will introduce in Section 4.1 allow a kind of address-based access. However, in part because we always write each memory sequentially, our model is somewhat simpler, not requiring operations like sharpening. Furthermore, we apply our memory model to textual reasoning tasks, which qualitatively differ from the more abstract operations of sorting and recall tackled by the NTM.

[1]Note that in this view, the terminology of input and output from Fig. 1 is flipped - when viewed as a traditional RNN with this special conditioning of outputs, A becomes part of the output embedding of the RNN and C becomes the input embedding.
ㄴ> C 아님?!
ㄴ> A 아님?!