CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 15

1 Reading

End-to-End Memory Networks

The contribution of this paper is an RNN with the ability to repeatedly read from (make multiple "hops" over) a large external memory before outputting a symbol. It can further be trained end-to-end, and so exhibits a versatility in the actual task it solves. In the paper, they test it on question-answering tasks.

The model takes a discrete set of inputs $x_1, ..., x_n$ which will be stored in aforementioned "external memory," along with a query q. It then outputs an answer a. Each quantity involves symbols from a vocabulary of V words. After writing all of the x, the model determines a continuous representation for the x and q. At last this continuous representation is processed via multiple hops to output a.

Single Layer Description

A single layer implements a single memory hop op. We embed the input x_i's into memory vectors $m_i \in \mathbb{R}^d$ in continuous space (in the simplest case with an embedding matrix A). We also embed the query q, e.g. with another matrix B, to obtain an internal state u. Then, in the embedding space, we compute matches between u and each memory vector m_i by taking the softmax of an inner product:

$$p_i = \text{softmax}(u^T m_i)$$

In this way p_i becomes the probability of input x_i being similar, i.e. relevant, to the query q. By extension, p is a probability vector over the inputs.

Meanwhile, each x_i has a second embedding c_i (which can be given by applying another matrix C to each x_i). Hence the memory response vector o becomes a sum of the c_i's weighted by their probabilities of being relevant:

$$o = \sum_i p_i c_i$$

If there is only one layer, the final prediction can be generated by passing the sum of the output vector o and the input embedding u through a final weight matrix W and a softmax.

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

In training, we can therefore minimize the cross-entropy between $\hat{a}$ and the true label a.

Multiple Layer Description

These layers can be stacked in order to handle K hop operations. The input to layers after the first (say layer $k + 1$) can simply be the sum of the output o^k and the input u^k from layer k:

$$u^{k+1} = u^k + o^k$$

Each layer should have its own embedding matrices A^k and C^k for embedding the inputs $\{x_i\}$. However, there are options for tying weights in order to reduce the number of parameters and thus ease training. For

instance, we can have the output embedding matrix for one layer be the input embedding matrix for the next, i.e. $A^{k+1} = C^k$. We can also have the input and output embedding matrices be the same across all layers, i.e. $A^1 = ... = A^K$ and $C^1 = ... = C^K$.

For the final prediction, the input W can again combine the input and output of the final memory layer:

$$\hat{a} = \text{softmax}(Wu^K + o^K)$$

2 Lecture

Recap

Last time, we argued that depth helps with sequence-to-sequence translation. Reversing the input sequence also helps deal with dependencies. One significant improvement in performance came from adding attention, i.e. as soft attention from each member of the output sequence to each member of the input sequence. This helps even more with dependencies.

We further argued that translation can be used to encode other kinds of structure as sequences. Even if the structure is not obviously linear, such as a parse tree, it can typically be linearized somehow. *Parsing reduces to translation.*

Finally, we looked at the Transformer – a recent approach to attention-based translation that eliminates the sequence models on both the encoder and decoder sides and replaces them with information propagating laterally using self-attention in a non-recurrent way. (The recurrence is fixed by the levels of nesting of the basic building block of the Transformer.) Nesting allows individual words to be influenced by other words (e.g. grammatical dependencies can be recognized in the self-attention) and also allows for resolution of hierarchical dependencies. There are multiple heads in the self-attention, each implementing a different function.

Memory Networks

In CNNs and sequence models, everything is sort of feedforward. Activations are very predictable given the inputs. With attention models, we have a new kind of activations which are activation weights, and these behave like pointers in that they extract or enhance some part of the input. These are often computed dynamically (even for static inputs).

With memory networks, we're adding even more complexity and internal state in the form of something like a memory. It's dynamic and open-ended; it's no longer the case that we can determine what it does from a finite set of inputs. These networks provide general-purpose memory, pointers via attention, and read/write capability. This is critical for dynamic memory in conversational agents.

In a bit more detail, memory networks model short and long-term memory. *Short-term* (or *working*) memory is dynamic and ephemeral; things go away quite rapidly. *Long-term* memory stores events in a write-once, read-many fashion; this is "accumulated experience memory." *Memory networks are supposed to model the event-driven, persistent memory that humans have.*

People use memory networks for classes of tasks such as *text Q&A*, *dialogue* (each round of dialogue adds some additional facts to the context, and we should remember these facts in order to have a coherent conversation), *learning from dialogue*, and *reading with attention over memory* (RAM).

Framework

Memory networks generally have a similar architecture. There are four components:

1. *Input feature map*: convert input data to a fixed vector representation.

2. *Generalization*: extend the long-term memory with new facts (given this new input).
3. *Output*: produce new output (in feature representation space) given the memories.
4. *Response*: convert the output into a response seen by the outside world.

Memory network interactions are typically iterative. We start with a query vector, i.e. an embedding of a query, and use it as a content-based address (for an attention model), meaning we take the query vector and try to match it to the embeddings of existing facts. When we find a match, we will take that match and turn it into a new context vector that we combine with the original query.

Q&A Memory Network

Let x_i be a piece of information that we're going to store permanently into our memory. It has two kinds of embeddings, $m_i = Ax_i$ and $c_i = Cx_i$. The first embedding (with map A) is used to match input queries, and the second embedding (with map C) is used to create output vectors that will either derive the output directly or get reused in an iterative process of querying.

Effectively, m_i is a key (because it's going to get matched to the query embedding) and c_i is a value (because it's what's going to go out of the key-value store).

Then the content-based addressing happens. A query q comes in, perhaps as a bag of embedded words, and gets re-embedded using a matrix B as $u = Bq$. We take this embedded version u and compute its inner product with all of the embedded memories. This will give us some scores, of which we can take the softmax to get a probability distribution p across the possible memory locations. (This represents attention.)

Finally we take these attention weights across all of the memories and compute a sum with the embedded values c_i. This is o, and is computed as $o = \sum_i p_i c_i$. And the return value is $\hat{a} = \text{softmax}(W(o + u))$, i.e. a softmax which combines a final linear map with the original embedded query.

Note: $o+u$ is a sum, but it's really more of a union because feature vectors are so sparse. If we want multiple layers, we can simply feed $o + u$ back in (to the next layer) for another iteration.

Each query-value retrieval is called a **hop**. Accuracy generally improves with the number of hops.

Position Encoding

Many models, such as memory networks and the Transformer, use position encoding so that embedded words carry information about their location in the input. (This encoding consistently outperformed bag-of-words in the paper from the beginning of these notes.) Memory networks, for example, multiply input words by a linear function of positions.

In a information retrieval task it's often important to know *when* things happen.

Key-Value Store

A key is used to find something; the value is the thing we retrieve. A **window-level encoding** encodes a window of words in bag-of-words as the key, and uses the center word as the value. A **KB-triple** encodes **k**nowledge-**b**ased facts, which have the form "subject relation object" – in this setup, the subject-relation pair is the key and the object is the value.

Knowledge-based encoding performs the best, but can't answer as many questions because it requires a particular structure.

Takeaways

Memory networks include the ability to make multiple retrievals from an episodic (WORM) memory. A memory network is like an associative key-value store which allows us to take a humanlike long-term memory

and retrieve facts from it in order to make interesting inferences on them – in a way that's learned entirely from examples. It uses *multiple hops* to follow inference chains or get up-to-date results.

Dialog

People are very interested in developing goal-directed dialog systems, which can perform a task as well as engage a user in chat. Traditional dialog systems use *slot-filling*; they convert from (unstructured) free text to a set of structured assertions (i.e. they bind actionable items to particular slots, e.g. for times or numbers of people). They then require the information for these slots in order to take the right action.

The main duties that an agent typically performs can be broken down into five tasks:

1. *Issuing API calls.* This is just basic slot-filling, whether prompted by the agent or automatically defined by the user. The agent then issues an API call saying "I need a restaurant with this time, location, etc."
2. *Updating API calls.* In case of clarification or lack of availability, the second task is to update requests for the API calls.
3. *Displaying options.* Given a user request, the agent should be able to list the user's available options.
4. *Providing extra information.* Maybe the agent needs a bit more information to fulfill the request.
5. *Conducting full dialogs.* This is just combining everything together.

The side effect of dialog with the agent is to cause the slots to get filled. We train a dialog-based agent using memory networks on top of the ground task, which is filling slots.

Memory networks are very relevant for dialog, because we follow a similar process of posting statements gleaned from assertions during the dialog; these effectively become facts about the world that we'll want to reference later. We can train agents to achieve our five tasks using memory networks.

Dynamic Memory Networks

Dynamic memory networks (DMNs) are similar to memory networks in that they also have input, scoring, attention, and response mechanisms. One difference is that they're a bit more flexible in their output (they include an RNN on the output instead of just a simple softmax). They also have RNNs on the input, and use these to perform a sentence-level (or question-level) embedding on top of the base words coming in. (The original memory networks simply assume a bag-of-words embedding for input queries.)

The biggest difference is that a DMN has an RNN managing the hopping within the memory. DMNs want to support repeated key-value retrieval and inference and therefore use a separate RNN to perform the functions of recursive hopping.

Essentially they add three RNNs and remove the ad hoc stacking of layers. (This means they lose the different embeddings that the original memory networks have at different levels. However, the same architecture can now be used for both text Q&A and visual Q&A.)

NLP Question Answering

> "All NLP/AI tasks can be reduced to question answering." – Richard Socher

Q&A is the basic model for making requests and getting things done with information systems. We either want to get information by asking questions, or we want the agent to do something for us by asking questions (which are really requests).

Nevertheless, we need robots if we want to do things in the physical world.

References

[1] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, Rob Fergus. End-To-End Memory Networks. In NIPS, 2015.