Question Answering with Maximum Inner Product Search

Minjoon Seo*[1], Tom Kwiatkowski[2], and Ankur Parikh[2]

[1]University of Washington
[2]Google Research, NY

September 10, 2017

1 Introduction

All current work using deep learning for extractive question answering relies on attention mechanisms that align question words with related words in the evidence document [8, 4] inter alia. While effective, this approach requires that the evidence document be re-encoded for each new question. This will not scale to real world question-answering tasks, since we cannot re-encode every document in the world every time we see a new question.

This project focuses on learning a question-independent representation of each possible answer in a document that supports fast lookup. In this setting, all document processing can happen offline, and only once. Then, the task of finding an answer for any given question is reduced to one of encoding the question and performing a nearest neighbor search among the answer candidates in the document. The complexity of this scales with the number of candidate answers logarithmically.

2 Inner Product Model

In extractive question answering, we take as input a question $\mathbf{q} = q_1, \ldots q_N$ and a evidence document $\mathbf{x} = x_1, \ldots, x_M$ that contains a single answer span. Our task is to correctly identify the start and end indices of this answer span.

Existing models predict answer boundaries by co-encoding $\mathbf{q}$ and $\mathbf{x}$ and using this encoding as input to classifiers used to predict the start and end indices. In contrast, the inner product models that we present here first encode $\mathbf{q}$ into a fixed length vector $\mathbf{u}$, and $\mathbf{x}$ into a sequence of fixed length vectors

*Work done while interning at Google.

$\mathbf{H} = \mathbf{h_1}, \ldots, \mathbf{h_M}$. Then an answer boundary is predicted as $t = \arg\max_t \mathbf{u}^\top \mathbf{h}_t$. The following sub-sections describe in detail the methods used to generate the representations $\mathbf{u}$ and $\mathbf{H}$. In the presentation of this work we focus on the prediction of a single answer boundary. The models are duplicated for the prediction of both start and end boundaries.

2.1 Word Embedding

All of the words in question and document are embedded into a continuous vector space using pretrained word vectors, GLoVe [6]. We also model character-level word embedding by learning an embedding for each character and then taking the maximum value in each of the embedding dimensions from the characters contained in each word. We concatenate the two vectors, GloVe vector and character-aware vector, for each word and pass it to a two-layer perceptron to enable interaction between the two vectors.

2.2 Question Encoding

We use a bidirectional LSTM to encode the question. We run the forward and backward LSTMs over the embedded question words and then use a weighted average of the concatenated output states to represent the question in a single fixed length vector. More formally, let $\mathbf{q_1}, \ldots, \mathbf{q_N} \in \mathbb{R}^d$ represent the sequence of LSTM output vectors. Then the final representation for the question, $\mathbf{u} \in \mathbb{R}^d$, is obtained by $\mathbf{u} = \sum_i p_i \mathbf{q}_i$ where $p_i = \text{softmax}(\mathbf{w}^\top \mathbf{q}_i)$ for learnable weight vector $\mathbf{w} \in \mathbb{R}^d$. Note that this also often considered as a self-attention mechanism, although we do not use the term to avoid confusion with the self-attention we use for the document side.

Multi-head. Instead of having single weight vector (and single weighted sum) for the question vector, we can have multiple different weight vectors to focus on different parts of the question. Then we can concatenate the heads into a single vector for the final representation. For instance, for double-head, we have two different weight vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, which lead to two different representations $\mathbf{u}_1$ and $\mathbf{u}_2$. Then the final representation is $\mathbf{u} = [\mathbf{u}_1; \mathbf{u}_2]$, where $[;]$ is columnwise vector concatenation. The motivation for this two head attention is to allow one head to focus on representing the answer-type information contained encoded in the question word while the other focuses on the relationship between the question word and other words in the question.

2.3 Answer Candidate Encoding

We are working in the extractive question answering setting defined by [7] where the answer to the question is always a substring of the evidence document. There are two predominant ways to represent each answer candidate, either through an explicit span representation [4], or by separately predicting the start and end index. In this project, we adopt the latter approach. Then the task becomes learning two vector representations for each word in the document, one for the start boundary and one for the end. Once these are learned and stored in hash table, one can use nearest neighbor search (where the query is the question encoding) to obtain the most similar word vector, whose position will be the start of the answer (or the end). Here we describe how we obtain the start representation of each word. Note that we use the same model architecture to obtian the end representation.

LSTM states. As with the question encoding, we apply a bidirectional LSTM top of the embedded document words. Let $\mathbf{x}_1, \ldots, \mathbf{x}_M \in \mathbb{R}^d$ represent the output sequence of the LSTM. As a baseline model, these representations can be directly used for nearest neighbor lookup. That is, $\mathbf{h}_t = \mathbf{x}_t$ for all t.

Self-attention. While the LSTM allows each index representation to be aware of its surrounding context, which is crucial, the LSTM still suffers from learning long-term dependencies. To counter this, we add an attention mechanism through which each index representation has direct access to all of the other LSTM outputs in the surrounding context[1]. Following [5], we avoid using the popular attention mechanism by [1], which requires explicit representation of all of the state pairs being compared—leading to huge memory comsuption. Rather, we entirely rely on dot product between state pairs, which can be efficiently computed using matrix multiplication in most GPUs.

More formally, let $\mathbf{X} = [\mathbf{x}_1^\top; \ldots; \mathbf{x}_M^\top] \in \mathbb{R}^{M \times d}$ be the matrix form of the LSTM outputs of the document, where $[;]$ indicates rowwise concatenation. Then we obtain query and key matrices, $\mathbf{K}, \mathbf{Q} \in \mathbb{R}^{M \times d}$, where each is a function of $\mathbf{X}$. There are several options for what the function could be, and we describe it later in this subsection. We use the obtained query and key matrices to compute the attention weights for each word, $\mathbf{A} \in \mathbb{R}^{M \times M}$:

$$\mathbf{A} = \text{softmax}(\mathbf{Q}^\top \mathbf{K}) \tag{1}$$

where the softmax function is applied at the second dimension (column dimension). Finally, the self-attended vectors are obtained by $\mathbf{V} = \mathbf{A}\mathbf{X} \in \mathbb{R}^{M \times d}$. Conceptually, each t-th row vector of $\mathbf{A}$ represents the attention distribution for where the t-th word should attend on. Hence, the matrix multiplication between $\mathbf{A}$ and $\mathbf{X}$ obtain those attended vectors for each t-th word.

For the self-attention model, we concatenate the LSTM output with the attended vector. That is, the representation for each t-th document word will be $\mathbf{h}_t = [\mathbf{x}_t; \mathbf{v}_t]$, where $\mathbf{v}_t$ is the t-th row vector of $\mathbf{V}$.

Obtaining query and key. We consider two functions for obtaining the query $\mathbf{Q}$ and key $\mathbf{K}$ representations used to calculate our attention weights. The first, also used by [9, 5], is a two-layer feed-forward neural network, without

[1] It is important to note that this attention is different from cross attention between question and context, which is not feasible in our case as the document must be processed without knowing what the question will be.

activation at the second layer. That is, $\mathbf{Q} = \sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$ where σ is a nonlinearity (e.g. ReLU). The second is using a single LSTM over the columns of $\mathbf{X}$ layer for each of the query and key, where the LSTM weights are not shared.

2.4 Answer Probability

After encoding question and document, we obtain a single vector for question $\mathbf{u}$, and M vectors for the document, $\mathbf{h}_1, \ldots, \mathbf{h}_M$. During training, we obtain probability distribution for the M answer candidates by $p_t = \text{softmax}(\mathbf{u}^\top \mathbf{h}_t)$. Then the loss function to be minimized is simply the negative log probability of the true answer, $L = -\log(p_s)$, where s is the true start index of the answer.

At inference time, we simply find the index $t = \arg\max_t \mathbf{u}^\top \mathbf{h}_t$. This has the benefit that we could compute all document vectors $\mathbf{h}_t$ offline and store them in a hastable for fast lookup using an off-the-shelf maximum inner product search (MIPS).

3 Regularization through Question Reconstruction.

We observed that it is very difficult to train the model described in Section 2 due to the very limited interaction between question and document. We subsequently propose an auxiliary task to help the training process, which can be considered as a regularization trick. The auxiliary task is to reconstruct the question with access to the true answer but without looking at the question. That is, we attempt to build an RNN decoder that generates the question given the true answer representation at the document side, $\mathbf{h}_s$ where s is the true answer (start position). $\mathbf{h}_s$ can be considered as the initial state of the RNN decoder, and we follow [2] to generate the question word by word. Let R be the sequence loss for generating the question, averaged over the (question) sequence. Then the final loss is computed by

$$L' = L + C \exp(-n \log(2)/\lambda) R \tag{2}$$

where n is training step, λ is half-life of the decay rate for R, and C is a constant coefficient. That is, R has a large influence to the training initially, and it has very little influence later.

4 Benchmark Model

As mentioned in Section 1, all state of the art use attention mechanisms between question and document words, which this project removes. However, as well as this interaction, the top performing models have also been very heavily engineered in other ways to maximize performance on the SQuAD benchmark. To facilitate a clear comparison between the MIPS models presented here and a model that only differs in terms of the question-dependent attention mechanism, we present a benchmark model that follows [8], but significantly simplifies it.

As in Section 2, the document and question words are encoded with a shared-weight LSTM layer. This gives question encoding $\mathbf{Q} \in \mathbb{R}^{N \times d}$ and document encoding $\mathbf{X} \in \mathbb{R}^{M \times d}$. We compute attention weights between each of pair question and document words:

$$\mathbf{A} = \mathbf{X} \cdot \mathrm{diag}(\mathbf{z}) \cdot \mathbf{Q}^\top \in \mathbb{R}^{M \times N} \tag{3}$$

where $\mathbf{z} \in \mathbb{R}^d$ is a learnable weight vector. These attention weights are used to obtain a weighted representation of the question for each index in the document, $\mathbf{V} = \mathbf{AQ} \in \mathbb{R}^{M \times d}$. We concatenate this index dependent question representation to the original document representation, as well as the elementwise multiplication of these two representations, $[\mathbf{X}, \mathbf{V}, \mathbf{X} \circ \mathbf{V}] \in \mathbb{R}^{M \times 3d}$, and use this as input to a final LSTM layer to obtain $\mathbf{H} \in \mathbb{R}^{M \times d}$. The probability of each answer index is calculated as $\mathrm{softmax}(\mathbf{Hw}) \in \mathbb{R}^M$, where $\mathbf{w}$ is a learnable weight vector. Similarly to Section 2, the loss to be optimized is the negative log probability of the true answer.

5 Experimental Setup

We use $d = 200$ and Adam optimizer [3] for all models, and train for 20,000 steps with early stopping based on dev-set performance. GloVe embeddings are obtained from 6B-token Wikipedia dump. To modulate the effect of the question reconstruction loss in 3 we use $C = 3.0, \lambda = 6000$. We evaluate all models on the Stanford Question Answering Dataset [7], which has 90k training examples and 10k dev examples.

6 Results and Discussions

Model	F1	Exact Match
LSTM states	57	46
With self attention (MLP)	56	44
With self attention (LSTM)	60	49
With query generation	63	52
Benchmark	74	62
R-Net	83	76

Table 1: Results on SQuAD.

Table 1 show the results of our inner product model, benchmark model, and the state-of-the-art model on SQuAD leaderboard. We first observe a significant benefit in using separate LSTMs to create the key and query representations for our self attention mechanism, over the MLP alternative. We believe that the

LSTMs allow the model to learn more complex interactions among the words in the document. We also see that co-training the model to generate queries helps a lot, giving additional 3% boost. However, the best inner product model included in this project is still more than 10% behind the benchmarked model, and 20% behind the state of the art.

7 Conclusion

In this project, we proposed a new problem, question answering with maximum inner product search, which can be easily scaled up to a large document or even a collection of many documents, such as Wikipedia. We report an LSTM-based baseline model, whose F1 score is 26% behind the state of the art. We use self-attention to reduce the gap by 3%, and we propose a novel regularization technique to further reduce the gap by 6%. We however also note that there still exists a large gap between the models presented here and the state of the art in extractive question answering, and this work requires significcant further improvement for practical usage.

References

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

[2] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[3] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] Kenton Lee, Shimi Salant, Tom Kwiatkowski, Ankur Parikh, Dipanjan Das, and Jonathan Berant. Learning recurrent span representations for extractive question answering. *arXiv preprint arXiv:1611.01436*, 2016.

[5] Ankur P Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*, 2016.

[6] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[7] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[8] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.