How Self-Attention with Relative Position Representations works

___ Feb 2, 2019 · 8 min read ★

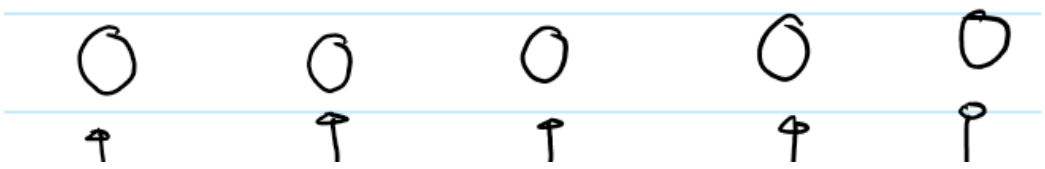
Introduction

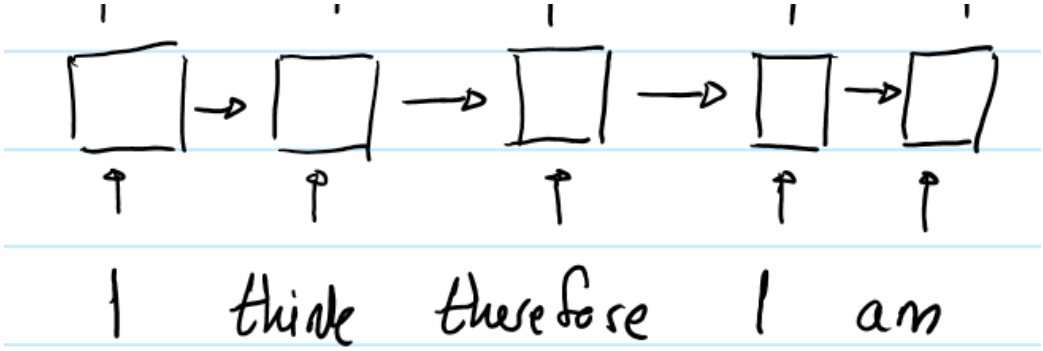
This article is based on the paper titled Self-Attention with Relative Position Representations by Shaw et al. The paper introduced an alternative means to encode positional information in an input sequence inside a Tranformer. In particular, it modified the Transformer's self-attention mechanism to efficiently consider the relative distances between sequence elements.

My goal is to explain the salient aspects of this paper in such a way people unaccustomed to reading academic papers can understand. I assume the reader has basic familiarity with Recurrent Neural Networks (RNNs) and the multi-head self-attention mechanism in Transformers.

Motivation

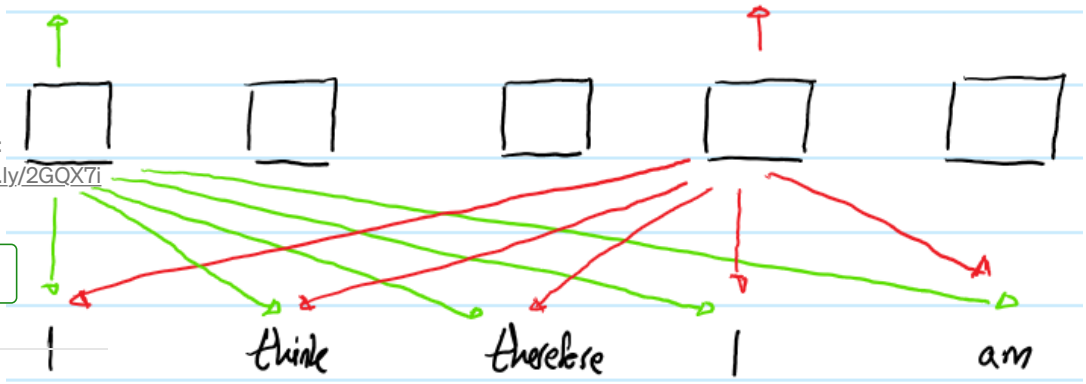
The architecture of an RNN allows it to implicitly encode sequential information using its hidden state. For example, the diagram below depicts an RNN that outputs a representation for each word in an input sequence where the input sequence is "I think therefore I am":





The output representation for the second “I” is not the same as the output representation for the first “I” because the hidden state that is inputted into these words are not the same: For the second “I”, the hidden state has passed though the words “I think therefore” while for the first “I” the hidden state is just initialized. Therefore, the RNN’s hidden state ensures that identical words that are in different positions in an input sequence will have distinct output representations.

In contrast, the self-attention layer of a Transformer (without any positional representation) causes identical words at different positions to have the same output representation. For example:



The diagram above shows the input sequence “I think therefore I am” being passed through just one Transformer. For readability reasons, only the inputs that make up the output representations of the “I”s are shown (in different colors). Notice that although the two “I” are located at different positions in the input sequence, the inputs for their respective output representations are identical.

Solution

Overview

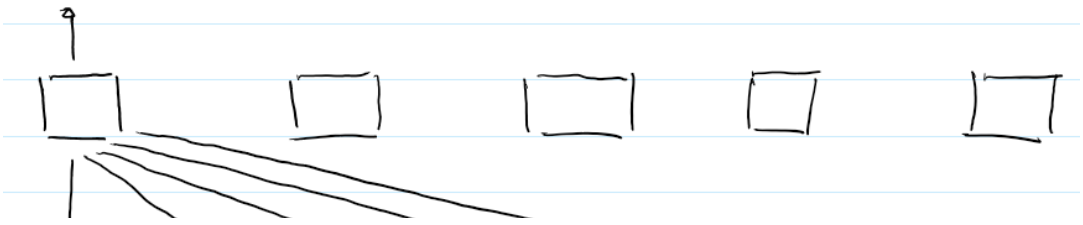
The authors proposed adding a set of trainable embeddings to the Transformer so as to make its output representations also represent the sequential information in its inputs. These embeddings are vectors that are used when computing the attention weight and value between word i and j in the input sequence. They represent the distance (number of words) between word i and j, hence the name Relative Position Representation (RPR).

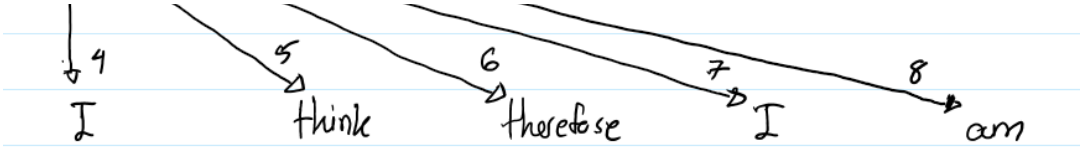
Top highlight

For example, a sequence of 5 words will have a total of 9 embeddings to be learned (1 embedding for the current word, 4 embeddings for the 4 words to the left of the current word and 4 embeddings for the 4 words to the right of the current word). The interpretation of these 9 embeddings are as follows:

Index	Interpretation
0	dist between word at position i and i-4
1	dist between word at position i and i-3
2	dist between word at position i and i-2
3	dist between word at position i and i-1
4	dist between word at position i and i
5	dist between word at position i and i+1
6	dist between word at position i and i+2
7	dist between word at position i and i+3
8	dist between word at position i and i+4

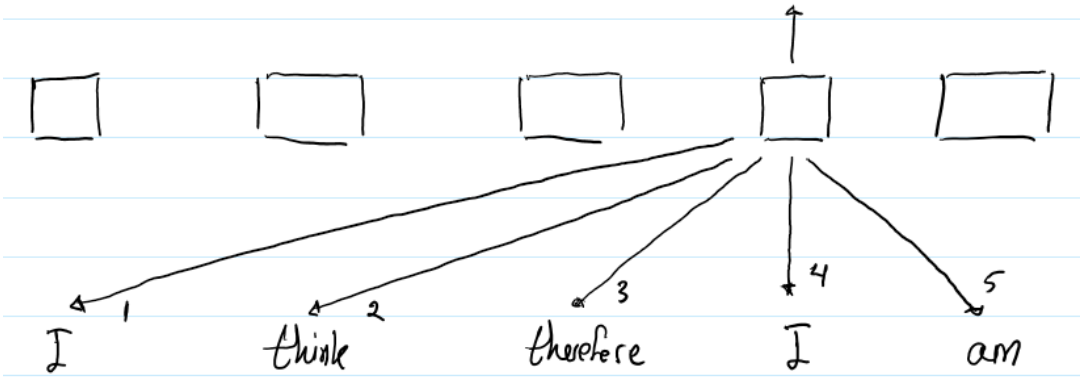
The following diagram will make it clear how these embeddings are use:





The above diagram depicts the process of computing the output representation of the first “I” in the sequence “I think therefore I am”. The numbers next to the arrows shows which of the RPRs are used when computing the attention. For example. When the Transformer is computing the attention to assign between “I” and the “therefore”, it will make use of the information contained in the 6th RPR, because “therefore” is 2 words away to the right of “I”.

The next diagram depicts the process of computing the representation of the second “I” in the same sequence:



As before, the second “I” will use the same input words as the first “I” to compute its output representation. However, the RPR associated with each word is different. For example, the 3rd RPR is used to compute the attention between “I” and “therefore” because “therefore” is one word away to the left of “I”. This is the mechanism by which RPRs help Transformers encode the sequential information in its inputs.

Notation

The following notation will apply for the remainder of this article:

z_i : The output representation for input word i

a_{ij} : The weight coefficient between input word i & word j

e_{ij} : The scaled dot product between input word i & word j

h : Number of attention heads in the Transformer
d_x : Embedding size of the words in the input sequence
d_z : Embedding size of z_i

W^Q, W^K, W^V : Projection matrices to create the query, key and value vectors respectively

a^V_{ij} : The RPR vector to use when computing z_i
a^K_{ij} : The RPR vector to use when computing e_{ij}

note: . $W^Q, W^K, W^V \in \mathbb{R}^{d_x \times d_z}$ and are unique per layer & attention head
. $a^V_{ij}, a^K_{ij} \in \mathbb{R}^{d_a}$
. $d_a = d_z$

Notice that there are actually two sets of RPR embeddings to learn: one for computing z_i and the other for computing e_{ij}. Unlike the projection matrices, these embeddings are shared across attention attention heads (they are still unique per layer though).

Another important point to note is that the maximum number of words considered is clipped to some absolute value of k. This means that the number of RPR embeddings to learn is 2k+1 (k words to the left, k worlds to the right and the current word). Words whose distance extend beyond k words to the right of word i are associated with the (k-1)-th RPR while those that extend beyond k words to the left of word i are associated with the 0-th RPR. For example, an input sequence that has 10 words and k=3 will have an RPR embedding lookup table as follows:

```
<tf.Tensor: id=1499, shape=(10, 10), dtype=int32, numpy=
array([[3, 4, 5, 6, 6, 6, 6, 6, 6, 6],
       [2, 3, 4, 5, 6, 6, 6, 6, 6, 6],
       [1, 2, 3, 4, 5, 6, 6, 6, 6, 6],
       [0, 1, 2, 3, 4, 5, 6, 6, 6, 6],
       [0, 0, 1, 2, 3, 4, 5, 6, 6, 6],
       [0, 0, 0, 1, 2, 3, 4, 5, 6, 6],
       [0, 0, 0, 0, 1, 2, 3, 4, 5, 6],
       [0, 0, 0, 0, 0, 1, 2, 3, 4, 5],
       [0, 0, 0, 0, 0, 0, 1, 2, 3, 4],
       [0, 0, 0, 0, 0, 0, 0, 1, 2, 3]])>
```

In this setup, row i corresponds to the i-th word and the columns represent word j in the input sequence. Index 3 corresponds to the RPR for word i (0-indexed), index 6 corresponds to the RPR for the 3rd word to the right of word i and index 0 corresponds to the 3rd word to the left of word i. The embedding lookup for the first word (row 0) is [3, 4, 5, 6, 6, 6, 6, 6, 6, 6]. Notice that the value of the lookup indices after the 3rd element are all 6. This means that even though the distance between the first and last word is 9, the RPR embedding associated with this pair is still the RPR embedding corresponding to the 3rd word to the right of the first word.

There are two reasons for doing this:

- The authors hypothesized that precise relative position information is not useful beyond a certain distance.
- Clipping the maximum distance enables the model to generalize to sequence lengths not seen during training.

Implementation

The following equations show the steps for computing z_i without RPR embeddings:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V) \qquad (1)$$

Each weight coefficient, α_{ij}, is computed using a softmax function:

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}}$$

And e_{ij} is computed using a compatibility function that compares two input elements:

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}} \qquad (2)$$

Introducing RPR embeddings modifies equation (1) as follows:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V + a_{ij}^V) \qquad (3)$$

and equation (2) as follows:

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}} \qquad (4)$$

In words, equation (3) means that when computing the output representation for word i, we modify the part where we weight the value vector for word j by modifying the value vector for word j by adding to it the RPR embedding between word i and j. Similarly, equation (4) tells us to modify the scaled dot product operation between word i and word j by adding to the key vector of word j the RPR embedding between word i and j. According to the authors, using addition as a means to incorporate the information in the RPR embeddings allows an efficient implementation, which will be described in the next section.

Efficient Implementation

The input to the Transformer is a tensor of size (batch_size, seq_length, embedding_dim).

Without RPR embeddings, the Transformer can compute $e_{i\,j}$ (equation 2) using batch_size * h parallel matrix multiplications. Each matrix multiplication will compute the $e_{i\,j}$ for all elements in a given input sequence and head. This is accomplished using the following expression:

$$\frac{(XW^Q)(XW^K)^T}{\sqrt{d_z}}, \quad X \in (seq\_length, d_x)$$

where X is the concatenation (row-wise) of the elements in the given input sequence.

In order to achieve similar efficiency (in terms of running time and space) with RPR embeddings, we first use the properties of matrix multiplication and transpose to rewrite equation (4) to:

$$e_{ij} = \frac{x_i W^Q (x_j W^K)^T + x_i W^Q (a_{ij}^K)^T}{\sqrt{d_z}} \quad (5)$$

The first term in the numerator is exactly the same as in equation (2), so it can be efficiently computed in a single matrix multiplication.

The second term is a bit tricky to compute efficiently. The code to do it is defined in the _relative_attention_inner function in the tensor2tensor repo so I will just briefly outline its logic in this article.

- The shape of the first term in the numerator is (batch_size, h, seq_length, seq_length). Row i and column j of this tensor is the dot product between the query vector of word i and the key vector word j. Therefore, our goal is to produce another tensor with the same shape as the first term but whose content is the dot product between the query vector of word i and the RPR embedding between word i and word j.
- First, we use the embedding lookup table like the one described in the Notation section to create the RPR embedding tensor for a given input sequence. Let us denote this tensor with A. This tensor will have shape (seq_length, seq_length, d_a). Then, we transpose A such that its new shape is (seq_length, d_a , seq_length) and call this A^T.
- Next, we compute the query vectors for all elements in the input sequence. This will give a tensor of shape (batch_size, h, seq_length, dz). We then transpose this tensor so that its shape is (seq_length, batch_size, h, dz) after which we reshape it to (seq_length, batch_size * h, dz). This tensor can now be multiplied with A^T. This multiplication can be cast as seq_length parallel matrix multiplications between (batch_size * h, dz) and (d_a, seq_length) matrices. Each seq_length matrix multiplication corresponds to a particular position in an input sequence. The matrix multiplication basically computes the dot product between that input position's query vector and its corresponding RPR embeddings across all input sequences in the heads and batch.
- The result of the above multiplication is a tensor with shape

(seq_length, batch_size * h, seq_length). We just need to reshape this to (seq_length, batch_size, h, seq_length) and transpose it to (batch_size, h, seq_length, seq_length) so that we can add it to the first term in the numerator.

The same logic can be applied to efficiently compute equation (3) too.

Results

The authors evaluated the impact of their modifications using the same machine translation task defined in the Attention is All You Need paper by Vaswani et al. Although the training steps per second dropped by 7%, their models improved the BLEU score by up to 1.3 for the English-to-German task and up to 0.5 in the English-to-French task.

Conclusion

In this article, I have explained why the self-attention mechanism in a Transformer does not encode an input sequence's positional information and how the Relative Position Representation embeddings proposed by Shaw et al. solves this problem. I hope this has improved your understanding of the paper. Let me know in the comments if you have any questions or feedback.

References

Self-Attention with Relative Position Representations; Shaw et al. 2018.

Attention Is All You Need; Vaswani et al., 2017.

Transformer: A Novel Neural Network Architecture for Language Understanding; Uszkoreit. 2017

The Unreasonable Effectiveness of Recurrent Neural Networks; Karpathy. 2015.

Machine Learning Deep Learning Neural Networks NLP Artificial Intelligence

About Help Legal