Seq2seq, NMT, Transformer

Milan Straka

Apr 17, 2023
Sequence-to-Sequence Architecture
Sequence-to-Sequence is a name for an architecture allowing to produce an arbitrary output sequence $y_1, \ldots, y_M$ from an input sequence $x_1, \ldots, x_N$.

Unlike CRF/CTC, no assumptions are necessary and we condition each output sequence element on all input sequence elements and all already generated output sequence elements:

$$P(y_i | x_1, \ldots, x_N, y_1, \ldots, y_{i-1}).$$
Sequence-to-Sequence Architecture

Figure 1 of "Sequence to Sequence Learning with Neural Networks", https://arxiv.org/abs/1409.0473
Sequence-to-Sequence Architecture

Decoder

Encoder

Figure 1 of "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", https://arxiv.org/abs/1406.1078
**Sequence-to-Sequence Architecture**

**Training**
The so-called **teacher forcing** is used during training – the gold outputs are used as inputs during training.

**Inference**
During inference, the network processes its own predictions – such an approach is called **autoregressive decoding**.

Usually, the generated logits are processed by an \( \text{arg max} \), the chosen word embedded and used as next input.
In the decoder, we both:

- embed the previous prediction, using a matrix of size $\mathbb{R}^{V \times D}$, where $V$ is the vocabulary size and $D$ is the embedding size;
- classify the hidden state into current prediction, using a matrix of size $\mathbb{R}^{D \times V}$.

Both these matrices have similar meaning – they represent words in the embedding space (the first explicitly represents words by the embeddings, the second produces logits by computing weighted cosine similarity of the inputs and columns of the weight matrix).

Therefore, it makes sense to tie these matrices, i.e., to represent one of them as a transposition of the other.

- However, while the embedding matrix should usually have constant variance per dimension, the output layer should keep the variance of the RNN output; therefore, the output layer matrix is usually the embedding matrix divided by $\sqrt{D}$. 
Bahdanau Attention

As another input during decoding, we add context vector $c_i$:

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

We compute the context vector as a weighted combination of source sentence encoded outputs:

$$c_i = \sum_j \alpha_{ij} h_j$$

The weights $\alpha_{ij}$ are softmax of $e_{ij}$ over $j$,

$$\alpha_i = \text{softmax}(e_i),$$

with $e_{ij}$ being

$$e_{ij} = v^\top \tanh(V h_j + W s_{i-1} + b).$$
Bahdanau Attention Implementation

\[ s_{i-1} \]
\[ h_1 \]
\[ s_{i-1} \]
\[ h_2 \]
\[ \vdots \]
\[ s_{i-1} \]
\[ h_N \]
\[ \text{tanh} \]
\[ \text{softmax} \]
\[ e_{i,1} \rightarrow \alpha_{i,1} \cdot h_1 \]
\[ e_{i,2} \rightarrow \alpha_{i,2} \cdot h_2 \]
\[ \vdots \]
\[ e_{i,N} \rightarrow \alpha_{i,N} \cdot h_N \]
\[ \sum \]
\[ c_i \]
Trained Attention Visualization

Figure 3 of "Neural Machine Translation by Jointly Learning to Align and Translate", https://arxiv.org/abs/1409.0473
Luong Attention

In the described Bahdanau (or additive) attention, we performed

\[ e_{ij} = \mathbf{v}^\top \tanh(\mathbf{V} \mathbf{h}_j + \mathbf{W} \mathbf{s}_{i-1} + \mathbf{b}). \]

There are however other methods how \( \mathbf{V} \mathbf{h}_j \) and \( \mathbf{W} \mathbf{s}_{i-1} \) can be combined, most notably the Luong (or dot-product) attention, which uses just a dot product:

\[ e_{ij} = (\mathbf{V} \mathbf{h}_j)^T (\mathbf{W} \mathbf{s}_{i-1}). \]

The latter is easier to implement, but may sometimes be more difficult to train (scaling helps a bit, wait for the Transformer self-attention description); both approaches are used in quite a few papers.
Subword Units

Translate **subword units** instead of words. The subword units can be generated in several ways, the most commonly used are:

- **BPE**: Using the *byte pair encoding* algorithm. Start with individual characters plus a special end-of-word symbol •. Then, merge the most occurring symbol pair $A, B$ by a new symbol $AB$, with the symbol pair never crossing word boundary (so that the end-of-word symbol cannot be inside a subword).

Considering text with words *low, lowest, newer, wider*, a possible sequence of merges:

\[
\begin{align*}
   & r \cdot \rightarrow r\cdot \\
   & l\ o \rightarrow lo \\
   & lo\ w \rightarrow low \\
   & e\ r\cdot \rightarrow er\cdot
\end{align*}
\]

The BPE algorithm is executed on the training data, and it generates the resulting dictionary, merging rules, and training data encoded using this dictionary.
Subword Units

- **Wordpieces**: Given a text divided into subwords, we can compute unigram probability of every subword, and then get the likelihood of the text under a unigram language model by multiplying the probabilities of the subwords in the text.

When we have only a text and a subword dictionary, we divide the text in a greedy fashion, iteratively choosing the longest existing subword.

When constructing the subwords, we again start with individual characters (compared to BPE, we have a *start-of-word* character instead of an *end-of-word* character), and then repeatedly join such a pair of subwords that increases the unigram language model likelihood the most.

  - In the original implementation, the input data were once in a while “reparsed” (retokenized) in a greedy fashion with the up-to-date dictionary. However, the recent implementations do not seem to do it — but they retokenize the training data with the final dictionary, contrary to the BPE approach.

For both approaches, usually quite little subword units are used (32k-64k), often generated on the union of the two vocabularies of the source and target languages (the so-called *joint BPE* or *shared wordpieces*).
Both the BPE and the WordPieces give very similar results; the biggest difference is that during the inference:

- for BPE, the sequence of merges must be performed in the same order as during the construction of the BPE (because we use the output of BPE as training data),
- for Wordpieces, it is enough to find longest matches from the subword dictionary (because we reprocessed the training data with the final dictionary);
- note that the above difference is mostly artificial – if we reparsed the training data in the BPE approach, we could also perform “greedy tokenization”.

Of course, the two algorithms also differ in the way how they choose the pair of subwords to merge.

Both algorithms are implemented in quite a few libraries, most notably the sentencepiece library and the Hugging Face tokenizers package.
Google NMT

Figure 1 of "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", https://arxiv.org/abs/1609.08144
Figure 5 of "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", https://arxiv.org/abs/1609.08144
Google NMT

Figure 6 of “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”, https://arxiv.org/abs/1609.08144
Beyond one Language Pair

Fig. 5. A selection of evaluation results, grouped by human rating.

Figure 5 of “Show and Tell: Lessons learned from the 2015 MSCOCO...”, https://arxiv.org/abs/1609.06647
Beyond one Language Pair

Figure 6 of "Multimodal Compact Bilinear Pooling for VQA and Visual Grounding", https://arxiv.org/abs/1606.01847

- What vegetable is the dog chewing on?
  - MCB: carrot
  - GT: carrot

- What kind of dog is this?
  - MCB: husky
  - GT: husky

- What kind of flooring does the room have?
  - MCB: carpet
  - GT: carpet

- What color is the traffic light?
  - MCB: green
  - GT: green

- Is this an urban area?
  - MCB: yes
  - GT: yes

- Where are the buildings?
  - MCB: in background
  - GT: on left

NPFL114, Lecture 10
Many attempts at multilingual translation.

- Individual encoders and decoders, shared attention.
- Shared encoders and decoders.

Surprisingly, even unsupervised translation is attempted lately. By unsupervised we understand settings where we have access to large monolingual corpora, but no parallel data.

In 2019, the best unsupervised systems were on par with the best 2014 supervised systems.

<table>
<thead>
<tr>
<th></th>
<th>WMT-14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsupervised</td>
<td></td>
</tr>
<tr>
<td>Proposed system</td>
<td>33.5 36.2 27.0 22.5</td>
</tr>
<tr>
<td>detok. SacreBLEU*</td>
<td>33.2 33.6 26.4 21.2</td>
</tr>
<tr>
<td>Supervised</td>
<td></td>
</tr>
<tr>
<td>WMT best*</td>
<td>35.0 35.8 29.0 20.6†</td>
</tr>
<tr>
<td>Vaswani et al. (2017)</td>
<td>- 41.0 - 28.4</td>
</tr>
<tr>
<td>Edunov et al. (2018)</td>
<td>- 45.6 - 35.0</td>
</tr>
</tbody>
</table>

Table 3: Results of the proposed method in comparison to different supervised systems (BLEU).

For some sequence processing tasks, *sequential* processing (as performed by recurrent neural networks) of its elements might be too restrictive.

Instead, we may want to be able to combine sequence elements independently on their distance.

Such processing is allowed in the **Transformer** architecture, originally proposed for neural machine translation in 2017 in *Attention is All You Need* paper.
Figure 1 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762
Transformer

http://jalammar.github.io/images/t/The_transformer_encoder_decoder_stack.png
Transformer

ENCODER #2

ENCODER #1

Feed Forward Neural Network

Self-Attention

x_1

x_2

z_1

z_2

r_1

r_2

http://jalammar.github.io/images/t/encoder_with_tensors_2.png
Transformer – Self-Attention

Assume that we have a sequence of \( n \) words represented using a matrix \( \mathbf{X} \in \mathbb{R}^{n \times d} \).

The attention module for queries \( \mathbf{Q} \in \mathbb{R}^{n \times d_k} \), keys \( \mathbf{K} \in \mathbb{R}^{n \times d_k} \) and values \( \mathbf{V} \in \mathbb{R}^{n \times d_v} \) is defined as:

\[
\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left( \frac{\mathbf{Q} \mathbf{K}^\top}{\sqrt{d_k}} \right) \mathbf{V}.
\]

The queries, keys and values are computed from the input word representations \( \mathbf{X} \) using a linear transformation as

\[
\mathbf{Q} = \mathbf{X} \mathbf{W}^Q
\]
\[
\mathbf{K} = \mathbf{X} \mathbf{W}^K
\]
\[
\mathbf{V} = \mathbf{X} \mathbf{W}^V
\]

for trainable weight matrices \( \mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d \times d_k} \) and \( \mathbf{W}^V \in \mathbb{R}^{d \times d_v} \).
Transformer – Self-Attention

Input

Embedding

Queries

Keys

Values

http://jalammar.github.io/images/t/transformer_self_attention_vectors.png
Transformer – Self-Attention

Input

Embedding

Queries

Keys

Values

Score

Divide by $8 \cdot \sqrt{d_k}$

Softmax

Softmax $X$

Value

Sum

Thinking

Machines

$x_1$  
$q_1$  
$k_1$  
$v_1$  
$q_1 \cdot k_1 = 112$

$x_2$  
$q_2$  
$k_2$  
$v_2$  
$q_1 \cdot k_2 = 96$

$14$  
$0.88$

$12$  
$0.12$

$z_1$  
$z_2$

http://jalammar.github.io/images/t/self-attention-output.png
Transformer – Self-Attention

Query q

Key k

q x k (elementwise)

q · k

Softmax

[CLS]

the

cat

sat

on

the

mat

[SEP]

the
dog

lay

on

the

floor

[SEP]

https://miro.medium.com/max/2000/1*5BslVNOOcJ-J3tsLVgni_w.png
Transformer – Self-Attention

\[
\begin{align*}
X \times W^Q &= Q \\
X \times W^K &= K \\
X \times W^V &= V \\
Q \times K^T &= Z \\
\text{softmax} \left( \frac{Q \times K^T}{\sqrt{d_k}} \right) &= Z
\end{align*}
\]

Multihead attention is used in practice. Instead of using one huge attention, we split queries, keys and values to several groups (similar to how ResNeXt works), compute the attention in each of the groups separately, concatenate the results and multiply them by a matrix $W^O$.

Figure 2 of “Attention Is All You Need”, https://arxiv.org/abs/1706.03762
Transformer – Multihead Attention

![Transformer Diagram](http://jalammar.github.io/images/t/transformer_attention_heads_qkv.png)

**ATTENTION HEAD #0**

- **Q₀**
- **K₀**
- **V₀**

<table>
<thead>
<tr>
<th>X</th>
<th>Thinking Machines</th>
</tr>
</thead>
</table>

**ATTENTION HEAD #1**

- **Q₁**
- **K₁**
- **V₁**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀^Q</td>
<td>W₀^K</td>
<td>W₀^V</td>
</tr>
<tr>
<td>W₁^Q</td>
<td>W₁^K</td>
<td>W₁^V</td>
</tr>
</tbody>
</table>

http://jalammar.github.io/images/t/transformer_attention_heads_qkv.png
Transformer – Multihead Attention

1) Concatenate all the attention heads

\[ Z_0 \quad Z_1 \quad Z_2 \quad Z_3 \quad Z_4 \quad Z_5 \quad Z_6 \quad Z_7 \]

2) Multiply with a weight matrix \( W^o \) that was trained jointly with the model

\[ W^o \]

3) The result would be the \( Z \) matrix that captures information from all the attention heads. We can send this forward to the FFNN

\[ Z \]

http://jalammar.github.io/images/t/transformer_attention_heads_z.png

http://jalammar.github.io/images/t/transformer_attention_heads_weight_matrix_o.png
Transformer – Multihead Attention

1. This is our input sentence
2. We embed each word
3. Split into 8 heads. We multiply X or R with weight matrices
4. Calculate attention using the resulting Q/K/V matrices
5. Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer

* In all encoders other than #0, we don’t need embedding. We start directly with the output of the encoder right below this one

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. $n$ is the sequence length, $d$ is the representation dimension, $k$ is the kernel size of convolutions and $r$ the size of the neighborhood in restricted self-attention.

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Complexity per Layer</th>
<th>Sequential Operations</th>
<th>Maximum Path Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Attention</td>
<td>$O(n^2 \cdot d)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Recurrent</td>
<td>$O(n \cdot d^2)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Convolutional</td>
<td>$O(k \cdot n \cdot d^2)$</td>
<td>$O(1)$</td>
<td>$O(\log_k(n))$</td>
</tr>
<tr>
<td>Self-Attention (restricted)</td>
<td>$O(r \cdot n \cdot d)$</td>
<td>$O(1)$</td>
<td>$O(n/r)$</td>
</tr>
</tbody>
</table>

Table 1 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762