Transformer and Syntax in NMT

Ondřej Bojar

April 16, 2020
Overview

• Reminder: Seq2seq with Attention.
• Transformer Architecture.
  • Focus on Self-Attention.
• Explicit Syntax in NMT.
  • In Network Structure.
  • At Each Token.
  • In Attention.

Some images due to Jindřich Helcl and/or Jindřich Libovický.
Reminder: Seq2seq with Attention
Attention – Formal Notation

Inputs:
- decoder state $s_i$
- encoder states $h_j = [\text{\vec{h}_j}; \text{\vec{h}_j}]$ \quad \forall i = 1 \ldots T_x
where $\vec{h}_j = \text{RNN}_{\text{enc}}(h_{j-1}, x_j) = \tanh(U_e h_{j-1} + W_e E_e x_j + b_e)$

Attention energies: $e_{ij} = \nu_a^\top \tanh(W_a s_{i-1} + U_a h_j + b_a)$

Attention distribution: $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

Context vector: $c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$
Attention Mechanism in Equations (2)

Decoder state:

\[ s_i = \tanh(U_d s_{i-1} + W_d E_d \hat{y}_{i-1} + C c_i + b_d) \]

Output projection:

\[ t_i = \tanh(U_o s_i + W_o E_d \hat{y}_{i-1} + C_o c_i + b_o) \]

...context vector is mixed with the hidden state

Output distribution:

\[ p(y_i = k \mid s_i, y_{i-1}, c_i) \propto \exp(W_o t_i)_k + b_k \]
Attention is All You Need (Vaswani et al., 2017)
Transformer Detailed Walkthroughs

Transformer Illustrated:

• http://jalammar.github.io/illustrated-transformer/
  Amazingly simple description! (I am reusing the pictures.)

Transformer paper annotated with PyTorch code:

• http://nlp.seas.harvard.edu/2018/04/03/attention.html

• PyTorch by examples:
  https://github.com/jcjohnson/pytorch-examples

Summary at Medium:

• https://medium.com/@adityathiruvengadam/
  transformer-architecture-attention-is-all-you-need-aecc
Transformer = 6 Layers Enc + 6 Dec

INPUT: Je suis étudiant

OUTPUT: I am a student
Composition of One Layer

ENCODER

- Feed Forward
- Self-Attention

DECODER

- Feed Forward
- Encoder-Decoder Attention
- Self-Attention
Word Vectors in Encoder

ENCODER

Feed Forward

Self-Attention

\( \mathbf{z}_1 \), \( \mathbf{z}_2 \), \( \mathbf{z}_3 \)

\( \mathbf{x}_1 \), \( \mathbf{x}_2 \), \( \mathbf{x}_3 \)

Je, suis, étudiant
FF Is Actually Position-Independent
Positional Encoding

• Encode token position directly in the word vector:

<table>
<thead>
<tr>
<th>POSITIONAL ENCODING</th>
<th>EMBEDDINGS</th>
<th>INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1</td>
<td>x1</td>
<td>Je</td>
</tr>
<tr>
<td>0.84 0.0001 0.54 1</td>
<td>x2</td>
<td>suis</td>
</tr>
<tr>
<td>0.91 0.0002 -0.42 1</td>
<td>x3</td>
<td>étudiant</td>
</tr>
</tbody>
</table>

• Positional embedding can be random, or “frequency-like”:
Self-Attention
Self-Attention Motivation (1/2)

- Sequences of arbitrary length $n$ need to be processed.
- RNNs make the (time-unrolled) network as deep as $n$.

- CNNs allow to trade kernel size $k$ and depth for a target “receptive field”:

\[
x_0 = \bar{0} \quad \text{embeddings} \quad \mathbf{x} = (x_1, \ldots, x_N) \quad x_n = \bar{0}
\]
SANs (Self-Attentive Networks) can access any position in constant time.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Sequential Steps</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recurrent</td>
<td>$O(n \cdot d^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Convolutional</td>
<td>$O(k \cdot n \cdot d^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Self-attentive</td>
<td>$O(n^2 \cdot d)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- Sequence length $n$, state dimensionality $d$, kernel size $k$.
- Assuming infinitely many GPU cores (or rather ALU), operations can be run in parallel, but may depend on each other, needing some Sequential Steps.
Self-Attention

- **Goal:** Aggregate arbitrary-length input to fixed-size vector. Allow data-driven, trainable aggregation.

Given the sequence of inputs $x_1, \ldots, x_n$: 

- Create three "views" of them: queries, keys, values.


Self-Attention

• Goal: **Aggregate** arbitrary-length input to fixed-size vector. **Allow data-driven, trainable** aggregation.

Given the sequence of inputs $x_1, \ldots, x_n$:

• Create three “views” of them: queries, keys, values.

• Using trained matrices $W^Q$, $W^K$, $W^V$. 

![Diagram of self-attention](image)
Match All Queries with All Keys

Input

Embedding

Queries

Keys

Values

Score

Thinking

x₁

q₁

k₁

v₁

q₁ • k₁ = 112

x₂

q₂

k₂

v₂

q₁ • k₂ = 96

Machines
Normalize Scores

Input

Embedding

Values

Score

Divide by $8 (\sqrt{d_k})$

Softmax

Thinking

$x_1$

$v_1$

$q_1 \cdot k_1 = 112$

14

0.88

Machines

$x_2$

$v_2$

$q_1 \cdot k_2 = 96$

12

0.12
Aggregate Values Accordingly

Input

Embedding

Values

Softmax

Softmax X Value

Sum

Thinking

x₁

v₁

0.88

v₂

Machines

x₂

v₂

0.12

v₁

z₁

z₂
Self-Attention as Matrix Calculation

$X \times W^Q = Q$

$X \times W^K = K$

$X \times W^V = V$

$\text{softmax} \left( \frac{Q \times K^T}{\sqrt{d_k}} \right) = Z$

$V$
Multi-Head Attention

**ATTENTION HEAD #0**

- \( Q_0 \)
- \( K_0 \)
- \( V_0 \)

- \( W_{0Q} \)
- \( W_{0K} \)
- \( W_{0V} \)

**ATTENTION HEAD #1**

- \( Q_1 \)
- \( K_1 \)
- \( V_1 \)

- \( W_{1Q} \)
- \( W_{1K} \)
- \( W_{1V} \)
Multi-Head Attention

Thinking Machines

Calculating attention separately in eight different attention heads

ATTENTION HEAD #0

ATTENTION HEAD #1

... 

ATTENTION HEAD #7
Self-Attention Summary

X

Thinking Machines

W_0^Q W_0^K W_0^V

W_1^Q W_1^K W_1^V

... W_7^Q W_7^K W_7^V

Q_0 K_0 V_0 Z_0

Q_1 K_1 V_1 Z_1

... ... ...

Q_7 K_7 V_7 Z_7

W^o

Z

Z_0 Z_1 ...

Z_7
Self-Attention in Transformer

Three uses of multi-head attention in Transformer

- **Encoder-Decoder Attention:**
  - Q: previous decoder layers; K = V: outputs of encoder
  - Decoder positions attend to all positions of the input.

- **Encoder Self-Attention:**
  - Q = K = V: outputs of the previous layer of the encoder
  - Encoder positions attend to all positions of previous layer.

- **Decoder Self-Attention:**
  - Q = K = V: outputs of the previous decoder layer.
  - Masking used to prevent depending on future outputs.
  - Decoder attends to all its previous outputs.
Self-Attention at Enc Layer #5: 1 Head
Self-Attention at Enc Layer #5: 2 Heads
Self-Attention at Enc Layer #5: 8 Heads
Explicit Linguistic Information in NMT
Ways of Adding Linguistic Annotation

• Construct network structure along linguistic structure.
  ~ What we discussed in Syntax in SMT.
  • Tree-LSTMs.
  • Graph-Convolutional Networks.
  ... Source information only.

• Enrich information at each token.
  ~ What we discussed in Morphology in SMT.
  • Factors on the source side.
  • Multi-Task on the target side.

• Improve attention using linguistic annotation.
  • Attention calculation respecting syntax.
  • Attention forced to reflect syntax in multi-task.
Linguistics in NN Structure
Tree-LSTMs (Tai et al., 2015)

Memory comes from:

the single predecessor

• Two flavors:
  • Dependency trees: Sum over all children.
  • Constituency trees: Up to N children, respecting order.
Tree-GRU Encoder:
- Constituency syntax of the tree provides additional states.
Bidirectional tree encoder.

- Can be seen as many RNNs running from each word up to the root and back to the word.
Graph-Convolutional Networks (Bastings et al., 2017)

Figure 2: A 2-layer syntactic GCN on top of a convolutional encoder. Loop connections are depicted with dashed edges, syntactic ones with solid (dependents to heads) and dotted (heads to dependents) edges. Gates and some labels are omitted for clarity.
Linguistics at Each Token
Syntax reflects long-distance dependencies.

- What city is the Taj Mahal in?
- Where is the Taj Mahal ∅?

The need to produce in depends on the What/Where.

- CCG tags for is differ ⇒ dependency highlighted.
- Following CCG tags, the decoder can know if in is needed.
- CCG tags are denser than words ⇒ better generalization.
CCGs to Encode Syntax at Each Token

Syntax reflects long-distance dependencies.

- **What** city is the Taj Mahal in?
- **Where** is the Taj Mahal ∅?

The need to produce *in* depends on the *What/Where*.

- CCG tags for *is* differ ⇒ dependency highlighted.
- Following CCG tags, the decoder can know if *in* is needed.
- CCG tags are denser than words ⇒ better generalization.

- **What**_{(S[wq]/(S[q]/NP))}/N city *is*_{(S[q]/P P)/NP} the Taj Mahal in?
- **Where**_{S[wq]/(S[q]/NP)} *is*_{(S[q]/NP)/NP} the Taj Mahal?
Factors in NMT

- **Source word factors easy to incorporate:**
  - Concatenate embeddings of the various factors.
  - POS tags, morph. features, source dependency labels help en↔de and en→ro (Sennrich and Haddow, 2016).

- **Target word factors:**
  - Interleave for morphology: (Tamchyna et al., 2017)
    
    Source (BPE) there are a million different kinds of pizza.
    Baseline (BPE) existují miliony druhů piz@ zy.
    Interleave VB3P existovat NNIP1 milion NNIP2 druh NNFS2 pizza Z:

  - Interleave for syntax: (Nadejde et al., 2017)
    
    Source BPE Obama receives Net+ an+ yahu in the capital of USA
    Target NP Obama ((S[dcl]\NP)/PP)/NP receives NP Net+ an+ yahu PP/NP in NP/N the N cap
Predicting Target Syntax

My students Dan Kondratyuk and Ronald Cardenas retried Nadejde et al. (2017) with:

- sequence-to-sequence model,
- Transformer model.

Predicting target syntax using:

- a secondary decoder
- interleaving.

As tags, they used:

- correct CCG tags, • random tags, • a single dummy tag.

(Kondratyuk et al. Replacing Linguists with Dummies. PBML 2019.)
Predicting Target Syntax (S2S)

**Seq2seq**

- Training steps (millions)
- Interleaved
- Multi-Decoder

- Baseline
- CCG
- Random
- Same
Predicting Target Syntax (Transformer)

- Transformer

- Multi-Decoder

- Baseline
- CCG
- Random
- Same

Training steps (millions)

Interleaved

Transformer

Multi-Decoder

Transformer

Baseline  CCG  Random  Same

Baseline  CCG  Random  Same

0  2  4  6  8  10  12  14  16  18  20  22  24  26  28

0  5  10  15  20  25  30
• CNN-derived embeddings of nodes’ syntactic neighbourhood included: (parent, siblings).
• Two mechanisms:
  • Concatenated to standard embeddings.
  • Separate attention over these word-level annotations
Linguistics in Attention
Tree coverage model:

- Attention coverage depends on source syntax.
- Without it (left), output is repeated.
Pham et al. (2019) noticed that attention head could be interpreted as dependency parse.

Add secondary objective to require head #1 to match source dependency tree.
- Czech-to-English translation (BLEU).
- Czech dependency parse from head #1 (UAS).

<table>
<thead>
<tr>
<th></th>
<th>BLEU Dev</th>
<th>BLEU Test</th>
<th>UAS Dev</th>
<th>UAS Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer Baseline</td>
<td>37.28</td>
<td>36.66</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Parse from layer 0</td>
<td>36.95</td>
<td>36.60</td>
<td>81.39</td>
<td>82.85</td>
</tr>
<tr>
<td>Parse from layer 1</td>
<td>38.51</td>
<td>38.01</td>
<td>90.17</td>
<td>90.78</td>
</tr>
<tr>
<td>Parse from layer 2</td>
<td>38.50</td>
<td>37.87</td>
<td>91.31</td>
<td>91.18</td>
</tr>
<tr>
<td>Parse from layer 3</td>
<td>38.37</td>
<td>37.67</td>
<td>91.43</td>
<td>91.43</td>
</tr>
<tr>
<td>Parse from layer 4</td>
<td>37.86</td>
<td>37.60</td>
<td>91.65</td>
<td>91.56</td>
</tr>
<tr>
<td>Parse from layer 5</td>
<td>37.63</td>
<td>37.67</td>
<td>91.44</td>
<td>91.46</td>
</tr>
</tbody>
</table>
I shot an elephant in my pajamas
<table>
<thead>
<tr>
<th></th>
<th>BLEU</th>
<th></th>
<th>Precision</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dev</td>
<td>Test</td>
<td>Dev</td>
<td>Test</td>
</tr>
<tr>
<td>Transformer Baseline</td>
<td>37.28</td>
<td>36.66</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Dummy Parse from layer 0</td>
<td>38.68</td>
<td><strong>38.14</strong></td>
<td>99.97</td>
<td>99.96</td>
</tr>
<tr>
<td>Dummy Parse from layer 1</td>
<td><strong>39.11</strong></td>
<td>38.06</td>
<td><strong>99.99</strong></td>
<td><strong>99.99</strong></td>
</tr>
<tr>
<td>Dummy Parse from layer 2</td>
<td>37.85</td>
<td>37.85</td>
<td>99.98</td>
<td>99.98</td>
</tr>
<tr>
<td>Dummy Parse from layer 3</td>
<td>37.93</td>
<td>37.70</td>
<td>99.97</td>
<td>99.98</td>
</tr>
<tr>
<td>Dummy Parse from layer 4</td>
<td>37.68</td>
<td>37.47</td>
<td>99.98</td>
<td>99.96</td>
</tr>
<tr>
<td>Dummy Parse from layer 5</td>
<td>37.53</td>
<td>37.54</td>
<td>99.96</td>
<td>99.95</td>
</tr>
<tr>
<td>True Parse from layer 1</td>
<td><strong>38.51</strong></td>
<td><strong>38.01</strong></td>
<td>90.17</td>
<td>90.78</td>
</tr>
</tbody>
</table>
Summary

- Transformer is a great replacement for RNN.
  - Constant-time processing.
  - (CNNs can be comparable, but Gehring et al. (2016) was kind of missed.)
- Explicit syntax can be useful.
  - Many options how to include it.
  - Some gains hard to reproduce.
  - Dummy information can be equally useful.
  - Transformer seems to learn syntax for free.
References


Thuong-Hai Pham, Dominik Macháček, and Ondřej Bojar. 2019. Promoting the knowledge of source syntax in transformer nmt is not needed. Computación y Sistemas, 23(3):923–934.
