Machine Translation

Transformation

Cheskidova E - cheskidova@phystech.edu
Deep Learning Lab MIPT

Outline

1. Seq2seq architectures with rnn
    a. Simple encoder, simple decoder
    b. Attention types
    c. Decoder with attention

2. Transformer
    a. Encoder
        i. Multi-head self-attention
        ii. LayerNorm & residual connections
        iii. Position-wise feed-forward
        iv. Positional Encoding
    b. Decoder
        i. Multi-head attention with encoder outputs
        ii. Masking

3. Optional Part

Outline

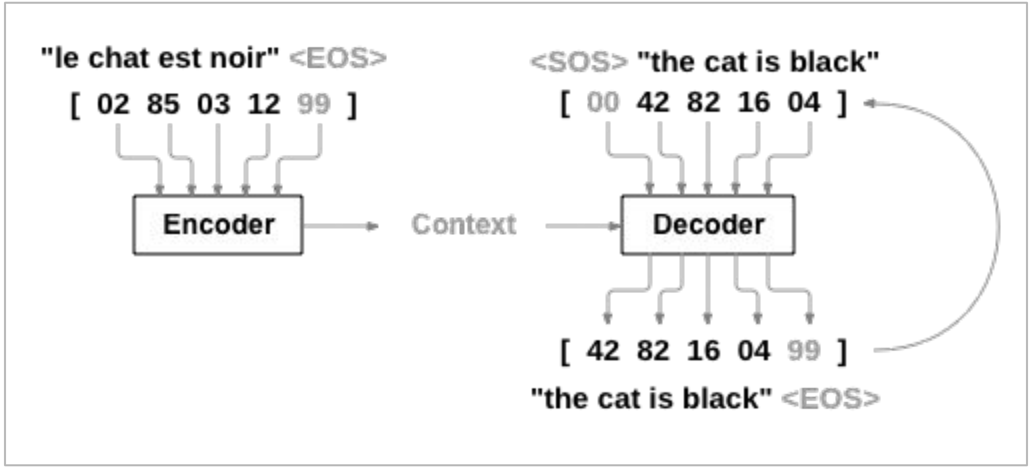
1. Seq2seq architectures with rnn
    a. Simple encoder, simple decoder
    b. Attention types
    c. Decoder with attention

2. Transformer
    a. Encoder
        i. Multi-head self-attention
        ii. LayerNorm & residual connections
        iii. Position-wise feed-forward
        iv. Positional Encoding
    b. Decoder
        i. Multi-head attention with encoder outputs
        ii. Masking

Seq2seq architectures with rnn

Simple encoder, simple decoder

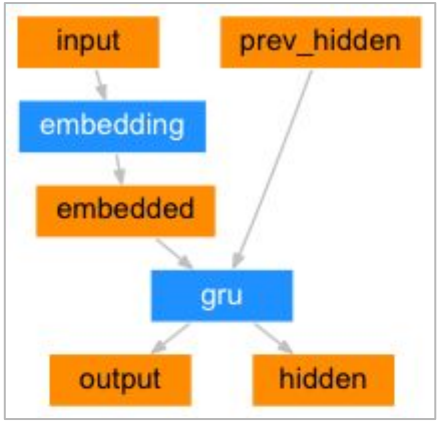
- How to transmit information from encoder to decoder?
  - Initialize hidden state of **decoder** with last hidden state of **encoder**
  - Concatenate last hidden state of **encoder** to each input of **decoder**



pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Seq2seq architectures with rnn

Simple encoder

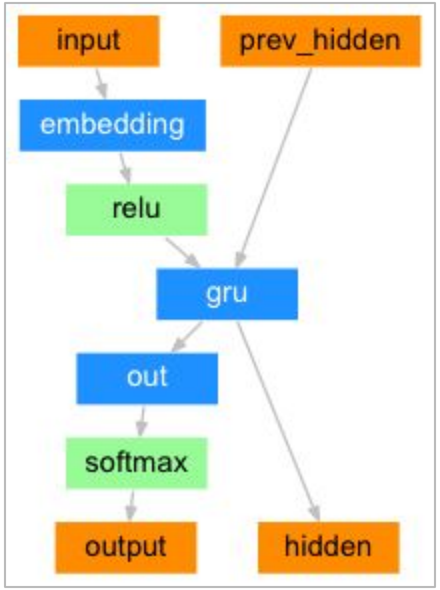



```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden
```

pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Seq2seq architectures with rnn

Simple decoder




```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden
```

pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Training

1. **X** := last **hidden state of encoder**
2. Init **decoder hidden state** with **X**
3. Input <SOS> token to decoder to start decoding
4. Calculate loss for the decoder output and right output (e.g. cross-entropy)
5. If (**teacher_forcing**):

        feeding ***right_prev_token*** into decoder

    else:

        feeding ***decoder_last_output*** into decoder

Outline

1. Seq2seq architectures with rnn
    a. Simple encoder, simple decoder
    b. Attention types
    c. Decoder with attention

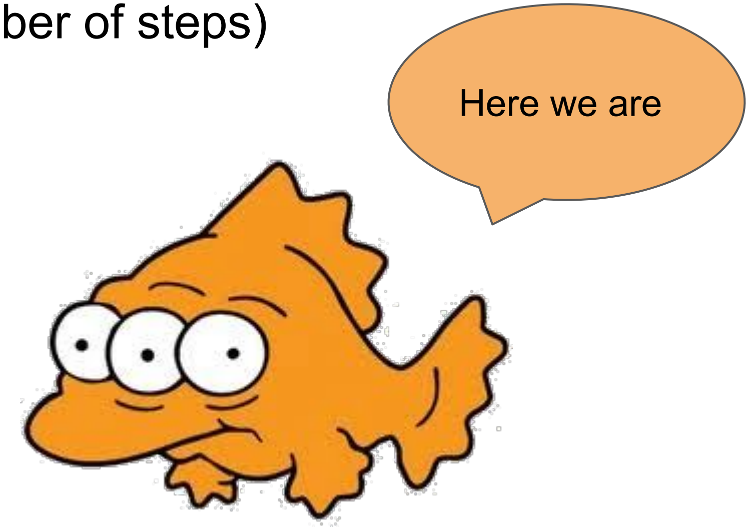
2. Transformer
    a. Encoder
        i. Multi-head self-attention
        ii. LayerNorm & residual connections
        iii. Position-wise feed-forward
        iv. Positional Encoding
    b. Decoder
        i. Multi-head attention with encoder outputs
        ii. Masking

Hard & soft attention

- **Hard**
  - Attention as stochastic process; sampling
  - Supports discrete decisions (number of steps)
  - Training with REINFORCE
- **Soft**
  - Attention as differentiable layer
  - No sampling
  - Training with backprop

Hard & soft attention

- **Hard**
  - Attention as stochastic process; sampling
  - Supports discrete decisions (number of steps)
  - Training with REINFORCE
- **Soft**
  - Attention as differentiable layer
  - No sampling
  - Training with backprop



Slide by: Michael Figurnov, DeepBayes school

Many ways to compute the score

$$\mathrm{score}(h, f) = \mathrm{NN}(h, f) = w_3^T \tanh(W_1 h + W_2 f)$$ [1]

additive attention
dim(*h*) != dim(*f*)

$$\mathrm{score}(h, f) = h^T f$$ [2]

$$\mathrm{score}(h, f) = h^T f / \sqrt{d},\ d = \dim(h)$$ [3]

multiplicative attention
dim(*h*) = dim(*f*)

[1] Bahdanau et al. "Neural Machine Translation by Jointly Learning to Align and Translate", 2014
[2] Luong et al. "Effective approaches to attention-based neural machine translation", 2015
[3] Vaswani et al. "Attention Is All You Need", 2017

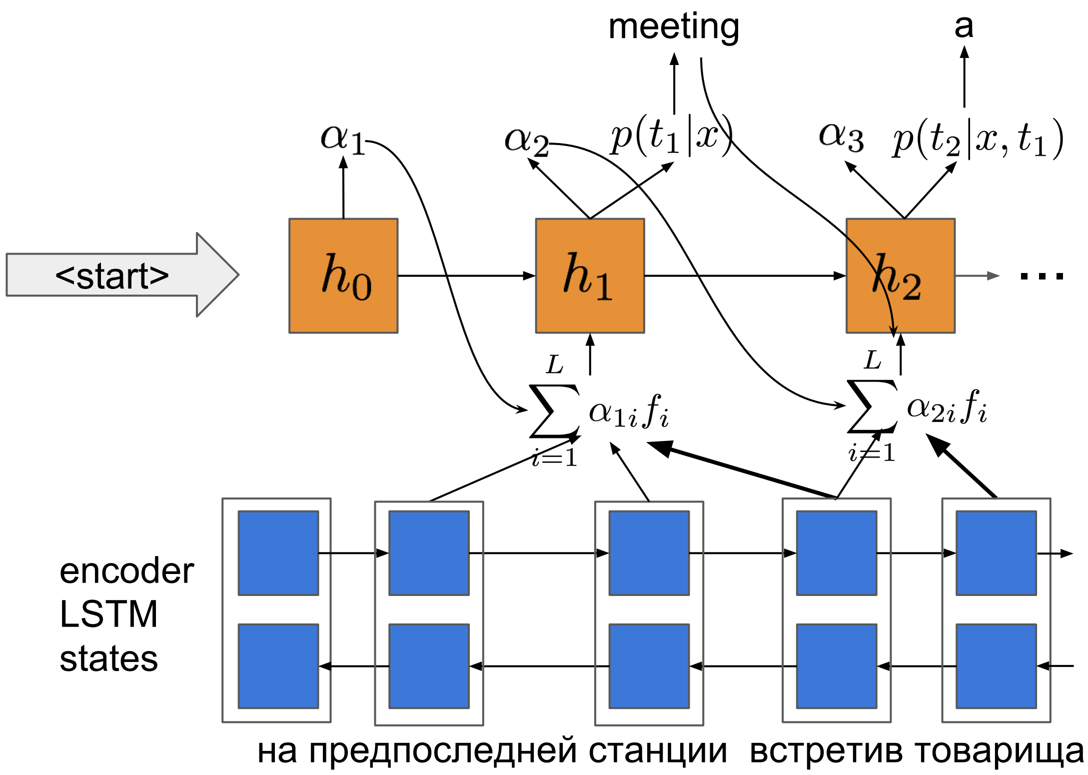
Slide credit: Michael Figurnov, DeepBayes school

Outline

1. Seq2seq architectures with rnn
    a. Simple encoder, simple decoder
    b. Attention types
    c. Decoder with attention

2. Transformer
    a. Encoder
        i. Multi-head self-attention
        ii. LayerNorm & residual connections
        iii. Position-wise feed-forward
        iv. Positional Encoding
    b. Decoder
        i. Multi-head attention with encoder outputs
        ii. Masking

Soft attention for neural machine translation



Bahdanau et al. "Neural Machine Translation by Jointly Learning to Align and Translate", 2014

Slide credit: Michael Figurnov, DeepBayes school

Transformer

NIPS 2017

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

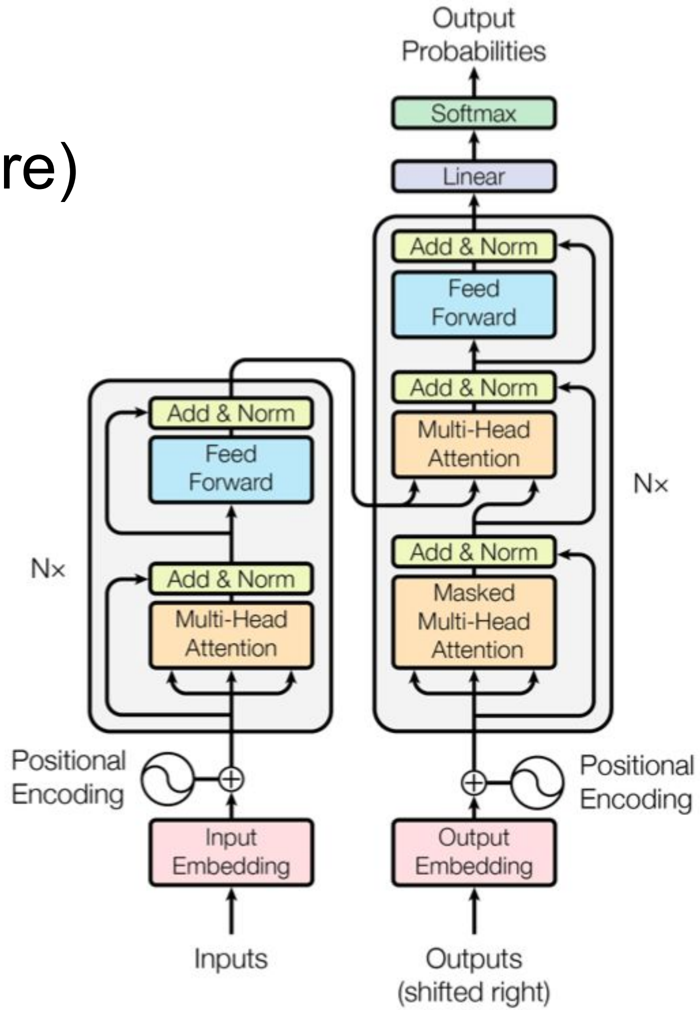
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

link: https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

Transformer
(model architecture)

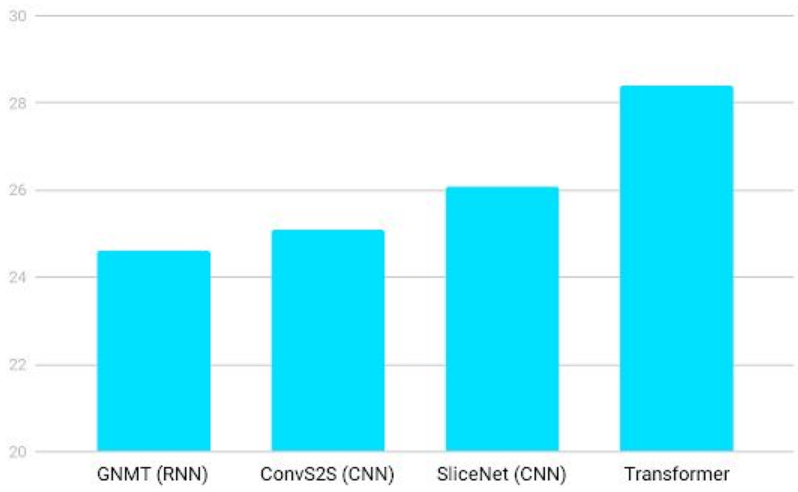
N = 6



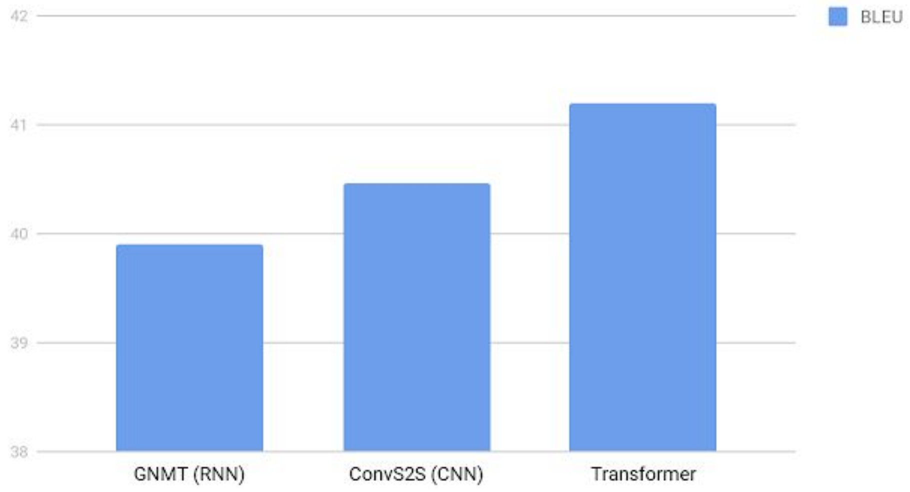
Transformer

- Better BLUE score

English German Translation quality

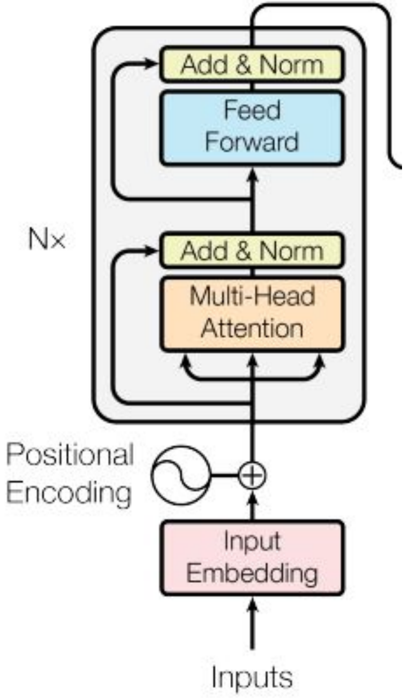


English French Translation Quality



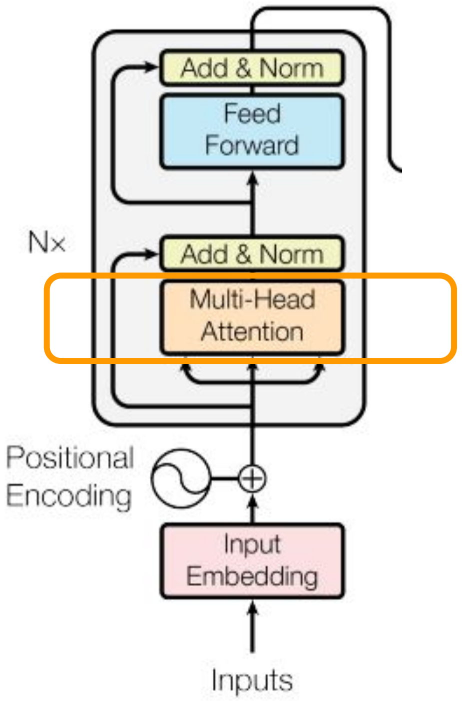
https://research.googleblog.com/2017/08/transformer-novel-neural-network.html

Encoder



- Multi-head self-attention
- LayerNorm & Residual connections
- Position-wise feed-forward
- Positional Encoding

Encoder



- Multi-head self-attention
- LayerNorm & Residual connections
- Position-wise feed-forward
- Positional Encoding

Multi-head self-attention

How does one head work?

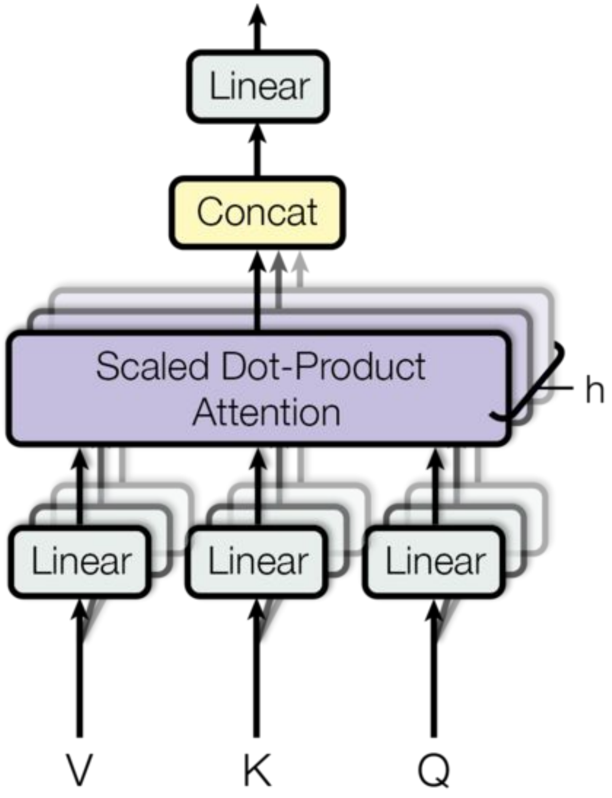
1. Value (V), Key (K), Query (Q) -- наборы векторов слов.
   (Value & Key одни и те же входные вектора, но подаются они на разные входы независимо)

2. Преобразуем входные V, K, Q каждый своим отдельным обучаемым линейным преобразованием.

3. Считаем скалярное произведение каждого вектора из Q с каждым вектором из K *(т.е. величину схожести каждого слова с каждым)*. Полученные произведения делим на корень из размерности вектора.

4. Для конкретного вектора q из Q мы получим преобразованный вектор, путем сложения всех векторов из V, с нормализованными весами, полученными в п. 3 с этим самым вектором q.

5. Возвращаем набор преобразованных векторов q.



Multi-head self-attention

Intuition

```
Keys = Values = Queries =
 embeddings([“In”, “my”, “humble”, “opinion”])

q := In;
q_out := my * (In, my) + humble * (In, humble) + opinion * (In, opinion)
```

Multi-head self-attention

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

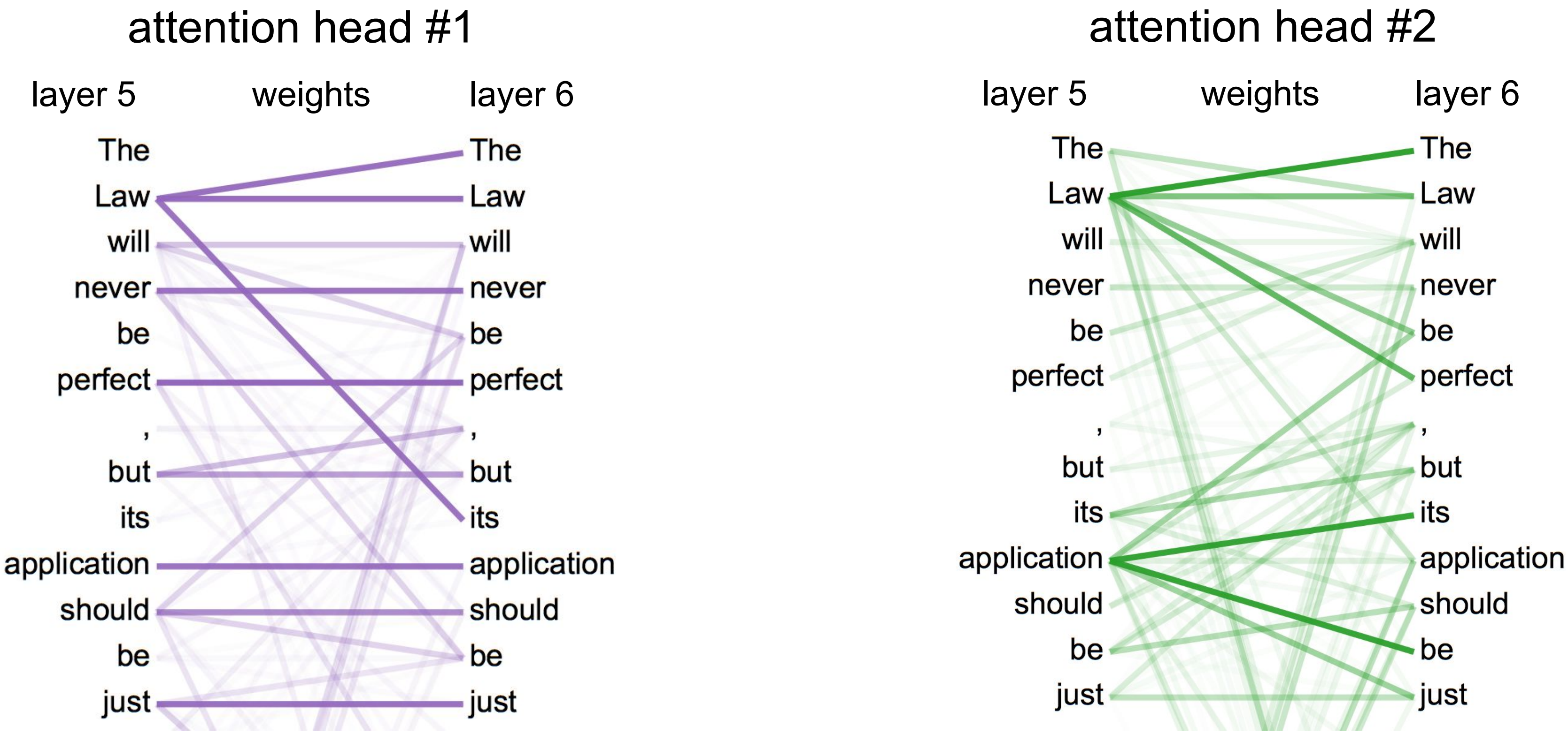
$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, \ldots, \mathrm{head_h})W^O$$

$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

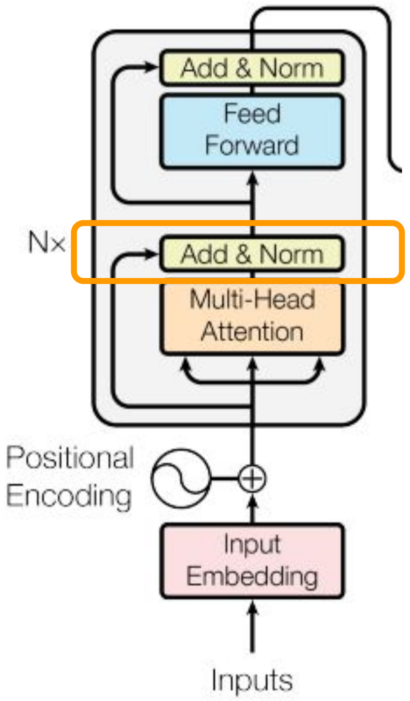
Dimension of the returned single matrix after applying Multi-head sublayer is the same as the dimension of any its input.

What do the self-attention heads look at?



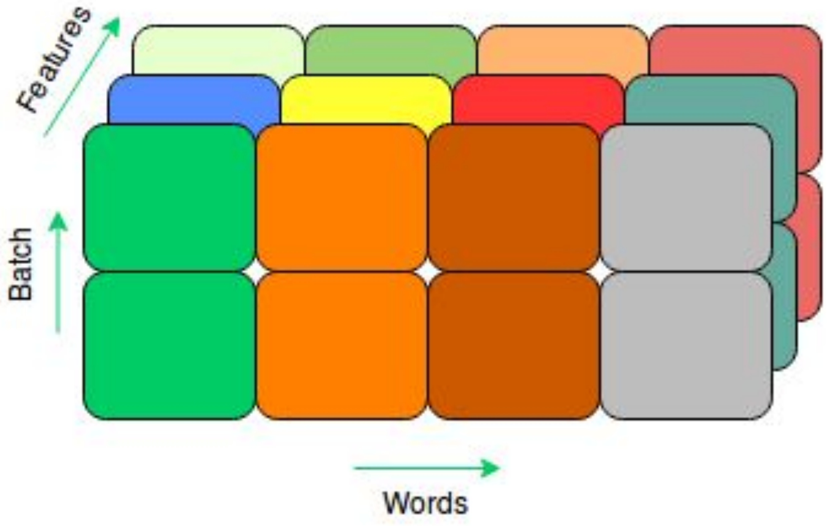
Slide credit: Michael Figurnov, DeepBayes school

Encoder

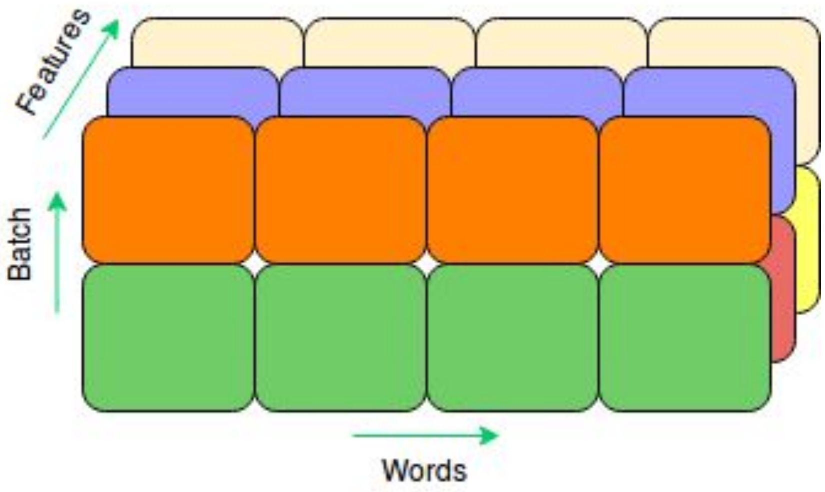


- Multi-head self-attention
- LayerNorm & Residual connections
- Position-wise feed-forward
- Positional Encoding

LayerNorm & Residual connections



Batch Norm



Layer Norm

LayerNorm & Residual connections

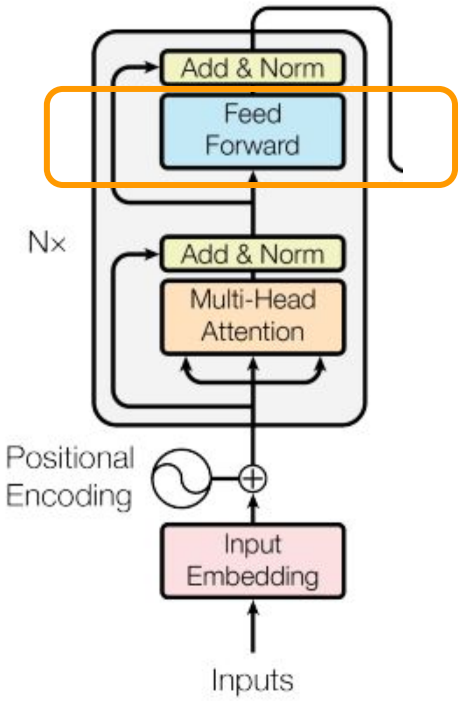
```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

Residual connection

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

Encoder



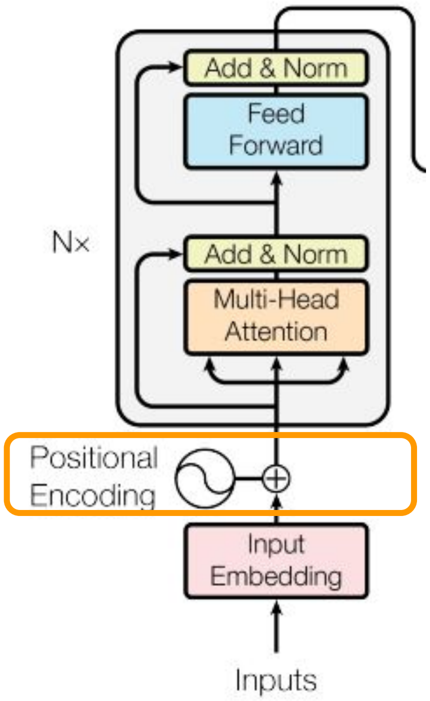
- Multi-head self-attention
- LayerNorm & Residual connections
- Position-wise feed-forward
- Positional Encoding

Position-wise feed-forward

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- ReLu + two dense layers
- dim(input) = dim(output)

Encoder



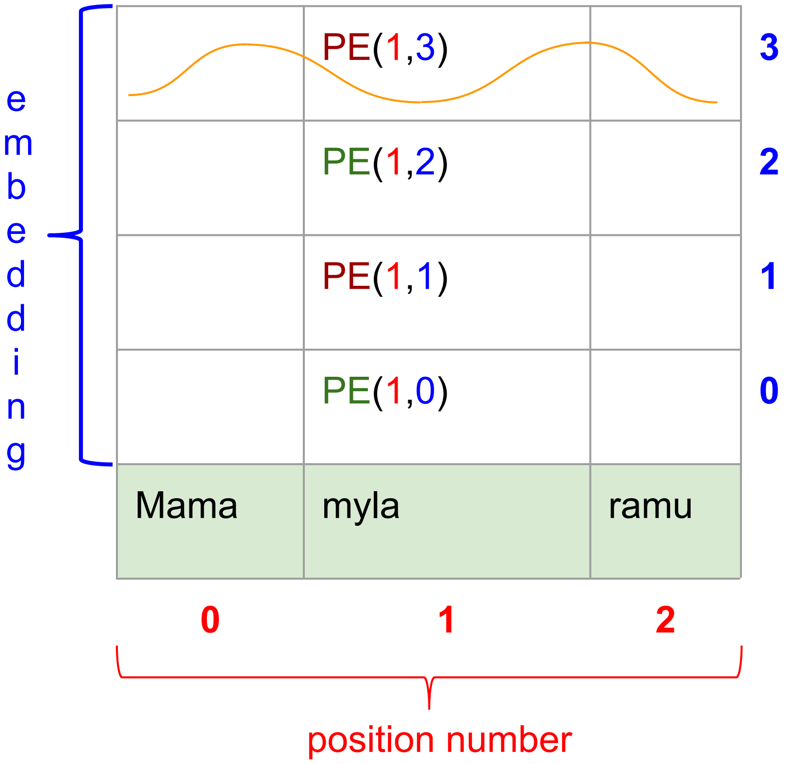
- Multi-head self-attention
- LayerNorm & Residual connections
- Position-wise feed-forward
- Positional Encoding

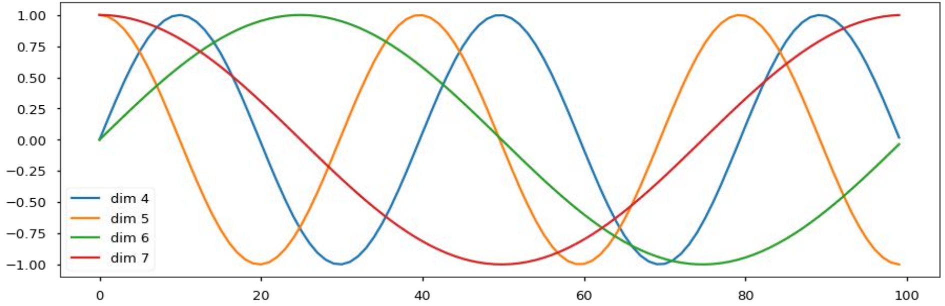
Positional encoding

- Learning embeddings for each position
  - (-) Bad embeddings for high-numbered positions
  - (-) More parameters needed
  - (-) Length restrictions
- Using sinusoids to encode position
  - **adding position vector to embedding vector**
  - concatenating position vector to embedding vector

Same scores with these two approaches

Positional encoding

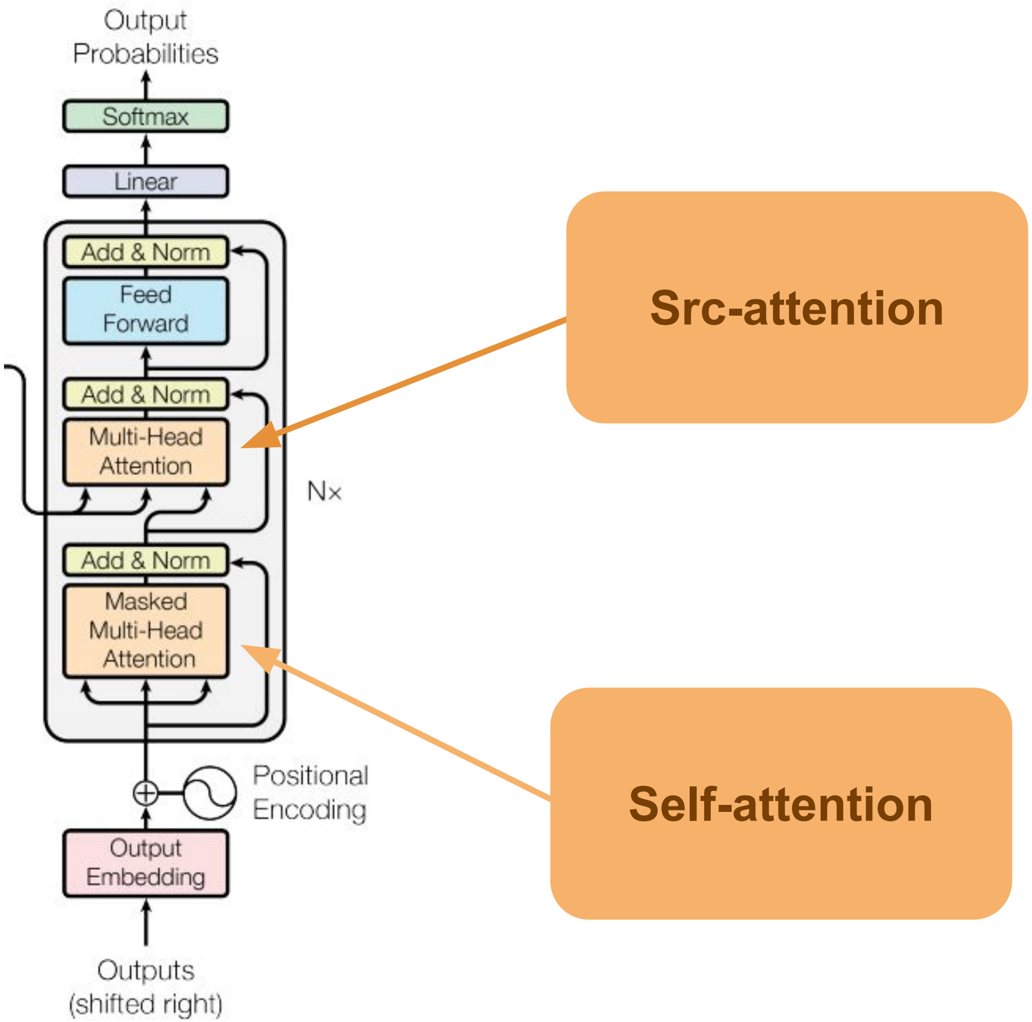




$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Decoder


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Add & Norm
Masked
Multi-Head
Attention
Positional
Encoding
Output
Embedding
Outputs
(shifted right)
Src-attention
Self-attention

Decoder

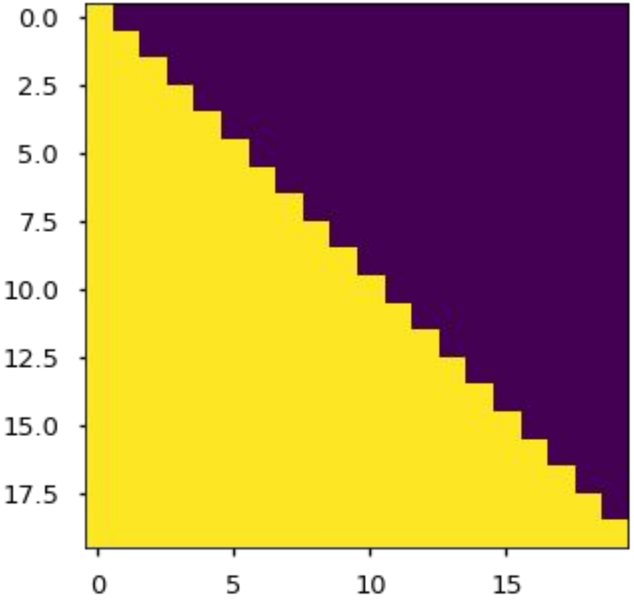
```python
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

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

Masking

The attention mask shows the position each tgt word (row) is allowed to look at (column).

Words are blocked for attending to future words during training.



https://research.googleblog.com/2017/08/transformer-novel-neural-network.html

Optional part

- Regularization
  (Label smoothing)
- BPE —
  Byte Pair Encoding for NMT
- Beam Search

Regularization

Label smoothing (LRS)

$$q'(k|x) = (1 - \epsilon)\delta_{k,y} + \epsilon u(k)$$

- Replace true label distribution $q(k|x) = \delta_{k,y}$ with $q'(k|x)$
- $u(k)$ - independent from x distribution (e.g. uniform)
- ϵ - just a small real number
- $\delta_{k,y}$ - Dirac delta; equals 1 if k = y else equals 0;

This hurts perplexity, as the model learns to be more unsure,
but improves accuracy and BLEU score.

Szegedy et. al, “Rethinking the Inception Architecture for Computer Vision”, 2015
https://arxiv.org/pdf/1512.00567.pdf

BPE

** **Byte Pair Encoding (BPE)** (Gage, 1994) is a simple data compression technique that iteratively replaces the most frequent pair of bytes in a sequence with a single, unused byte. We adapt this algorithm for word segmentation. Instead of merging frequent pairs of bytes, we merge characters or character sequences.*

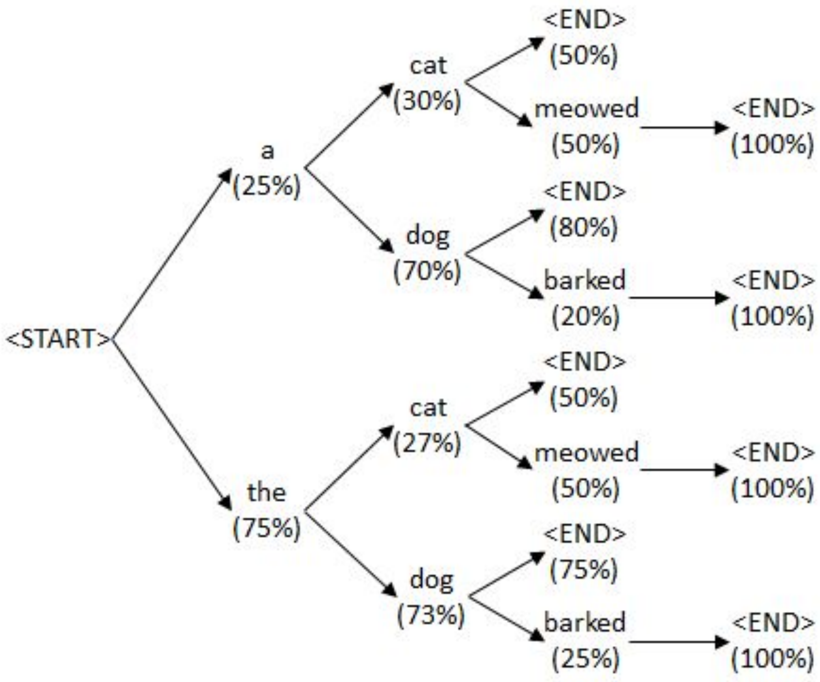
- *Official implementation: https://github.com/rsennrich/subword-nmt*

Example

- Input:
  - `“It is the case of Alexander Nikitin.”`
- After applying BPE:
  - `It is the case of Alexander Ni@@ ki@@ tin .`

Rico Sennrich et al, “Neural Machine Translation of Rare Words with Subword Units”, 2016
https://arxiv.org/pdf/1508.07909.pdf

Beam Search



PyTorch OpenNMT implementation:
https://github.com/OpenNMT/OpenNMT-py/blob/master/onmt/translate/Beam.py

Homework

- Выбрать одну из возможных маленьких исследовательских задачек и подготовить отчет с кодом по ней.
- Возможно придумать свою задачу подобного рода и подготовить отчет по ней.
- Возможные задачи для исследования:
    - Заменить прибавление синусоид к input в Positional Encoding на конкатенацию к input PE вектора фиксированной длины + projection до исходной размерности input.
    - Заменить multiplicative attention в на additive attention в multi-head (а именно в подсчете Scaled-Dot-Product Attention).
    - Попробовать BPE (воспользоваться реализацией)
    - Попробовать заменить в PyTorch NMT tutorial их энкодер на энкодер из кода трансформера.


вопросы