Attention Is All You Need

- Ho Beom Jeon
- UST-ETRI HRI-Lab

- 2021. 5. 28.

Contents

1. Introduction
2. Background
3. Model Architecture
4. Why Self-Attention
5. Training
6. Results
7. Conclusion

1. Introduction

Attention Is All You Need

21032회 인용

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

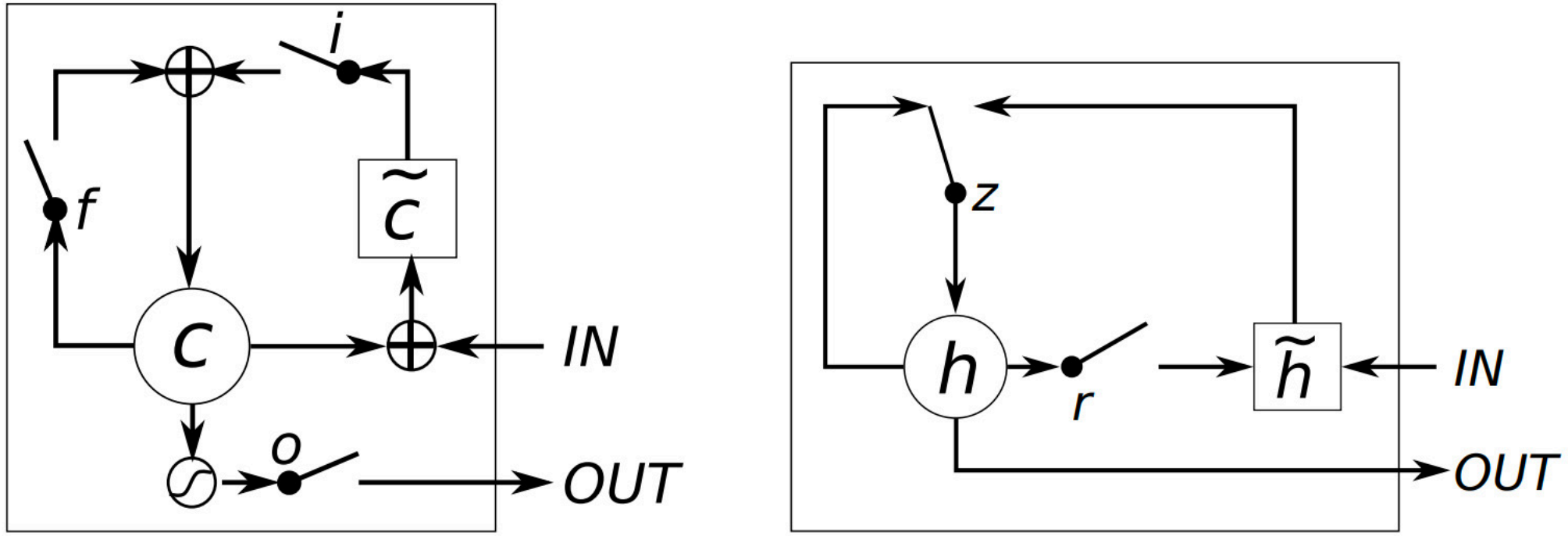
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

1. Introduction



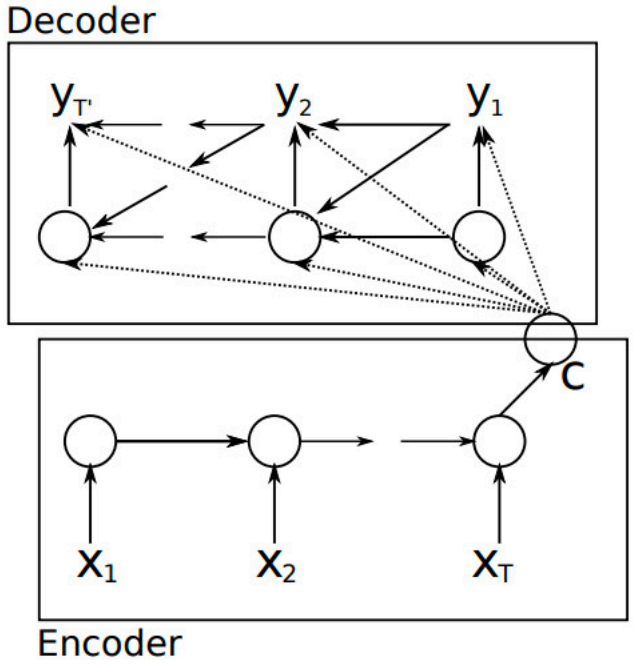
(a) Long Short-Term Memory

(b) Gated Recurrent Unit

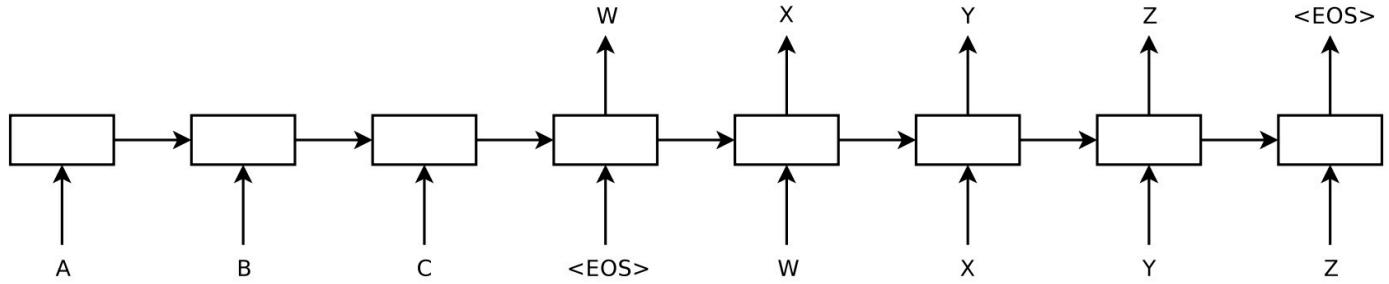
Figure 1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and $\tilde{c}$ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and $\tilde{h}$ are the activation and the candidate activation.

[7] Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling (2014) 6678

1. Introduction



[5] Figure 1: An illustration of the proposed RNN Encoder–Decoder.

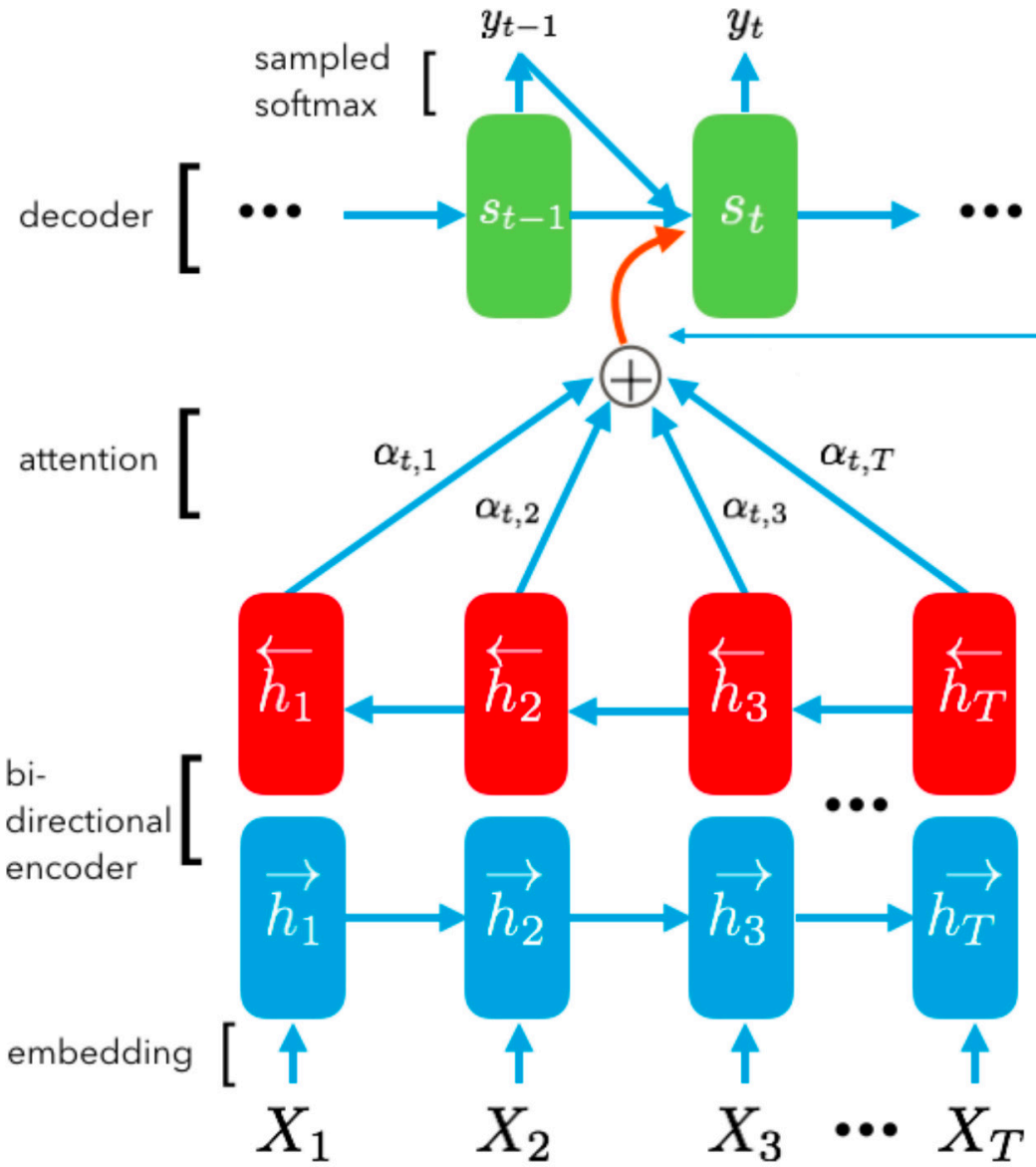


[29] Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

[5] Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation (2014) 13775
[29] Sequence to sequence learning with neural networks (2014) 14766

1. Introduction

목표 단어를 생성할 때 연관성 높은 문장의 단어를 찾을 수 있도록 한다.



$$p(y_i|y_1,...,y_{i-1},X) = g(y_{i-1},s_i,c_i)$$

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

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

$$\alpha_{i,j} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})}$$

$$e_{ij} = a(s_{i-1},h_j)$$

where a is a feed forward neural network.

$$h_j = [\rightarrow h_j; \leftarrow h_j]_{concat}$$

[2] Neural machine translation by jointly learning to align and translate (2014) 17808 (Slide credit: Xinyuan Zhang) 중요!

1. Introduction

고정 길이의 문맥 벡터(Context vector)는 긴 문장을 번역하기 힘들다.

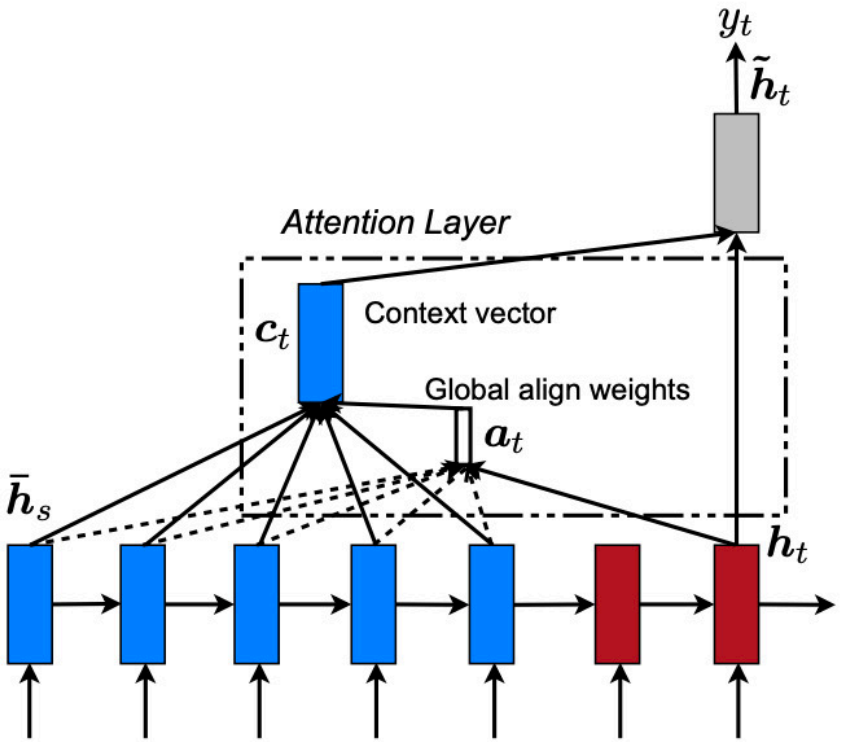


Figure 2: **Global attentional model** – at each time step t, the model infers a *variable-length* alignment weight vector a_t based on the current target state h_t and all source states $\bar{h}_s$. A global context vector c_t is then computed as the weighted average, according to a_t, over all the source states.

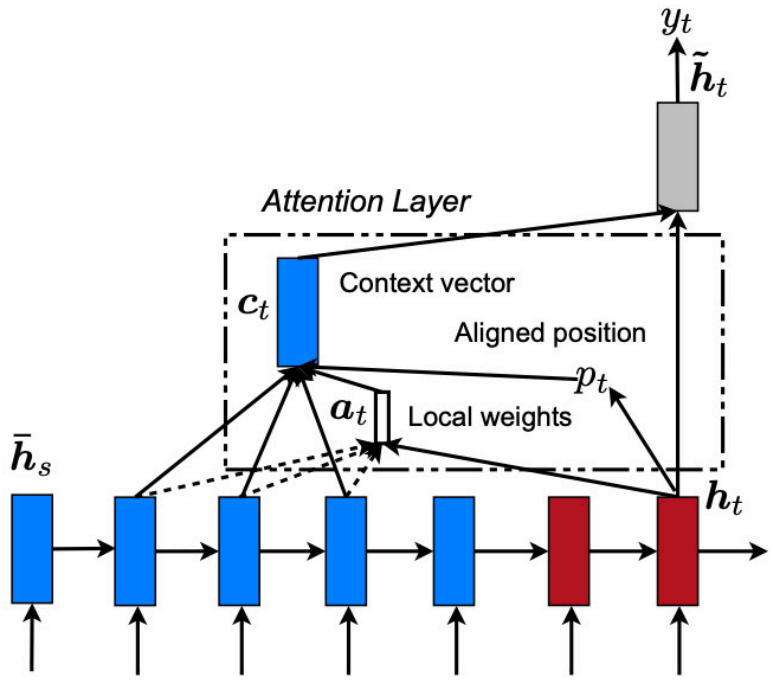


Figure 3: **Local attention model** – the model first predicts a single aligned position p_t for the current target word. A window centered around the source position p_t is then used to compute a context vector c_t, a weighted average of the source hidden states in the window. The weights a_t are inferred from the current target state h_t and those source states $\bar{h}_s$ in the window.

[21] Effective approaches to attention- based neural machine translation (2015) 5593

1. Introduction

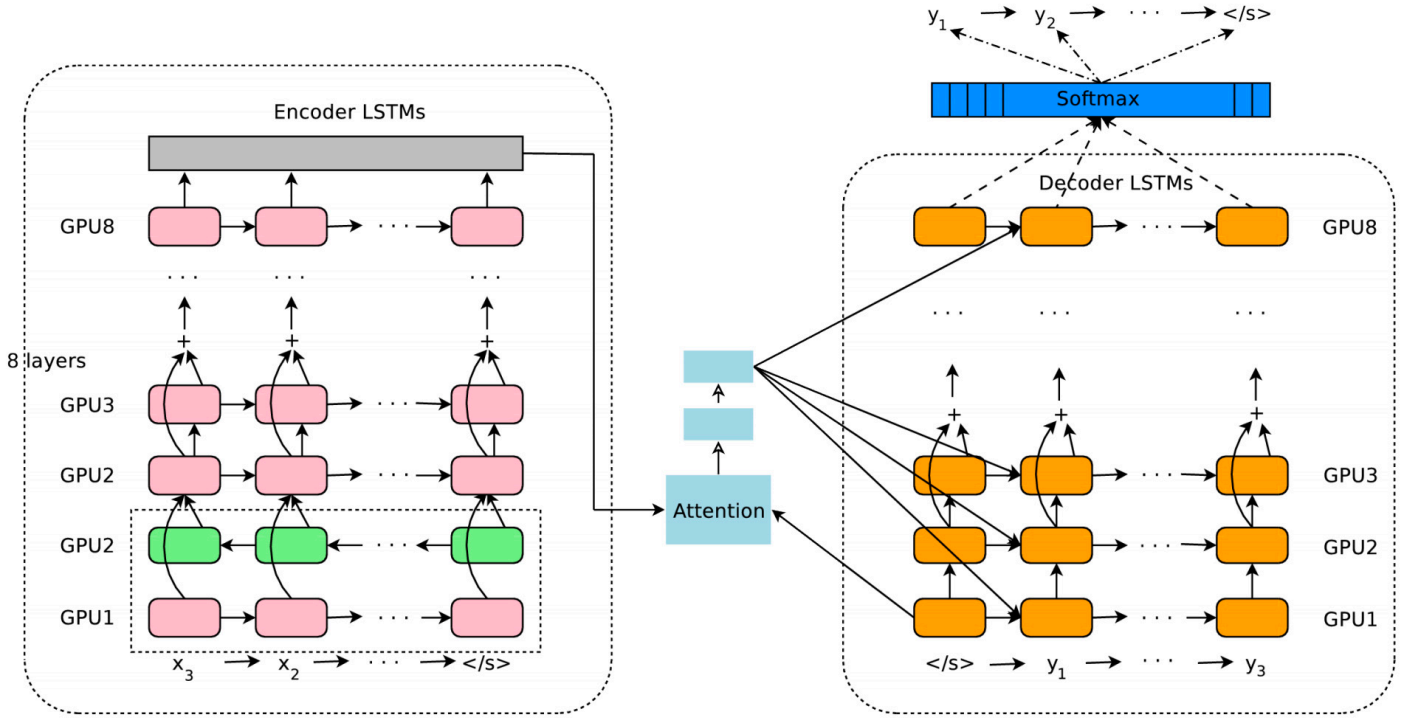


Figure 1: The model architecture of GNMT, Google's Neural Machine Translation system. On the left is the encoder network, on the right is the decoder network, in the middle is the attention module. The bottom encoder layer is bi-directional: the pink nodes gather information from left to right while the green nodes gather information from right to left. The other layers of the encoder are uni-directional. Residual connections start from the layer third from the bottom in the encoder and decoder. The model is partitioned into multiple GPUs to speed up training. In our setup, we have 8 encoder LSTM layers (1 bi-directional layer and 7 uni-directional layers), and 8 decoder layers. With this setting, one model replica is partitioned 8-ways and is placed on 8 different GPUs typically belonging to one host machine. During training, the bottom bi-directional encoder layers compute in parallel first. Once both finish, the uni-directional encoder layers can start computing, each on a separate GPU. To retain as much parallelism as possible during running the decoder layers, we use the bottom decoder layer output only for obtaining recurrent attention context, which is sent directly to all the remaining decoder layers. The softmax layer is also partitioned and placed on multiple GPUs. Depending on the output vocabulary size we either have them run on the same GPUs as the encoder and decoder networks, or have them run on a separate set of dedicated GPUs.

[31] Google's neural machine translation system: Bridging the gap between human and machine translation. (2016) 4106

1. Introduction

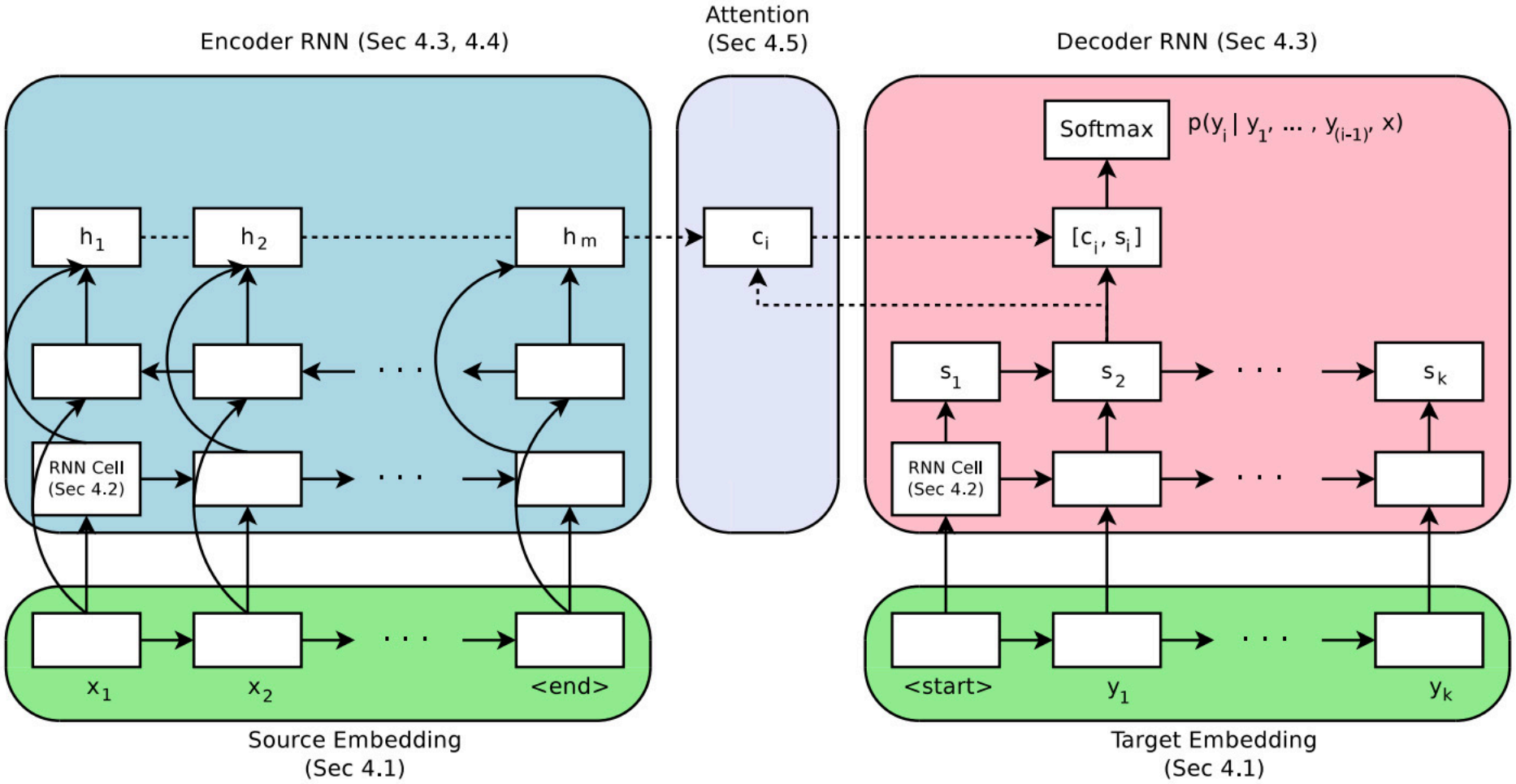


Figure 1: Encoder-Decoder architecture with attention module. Section numbers reference experiments corresponding to the components.

[3] Massive Exploration of Neural Machine Translation Architectures (2017) 406

2. Background

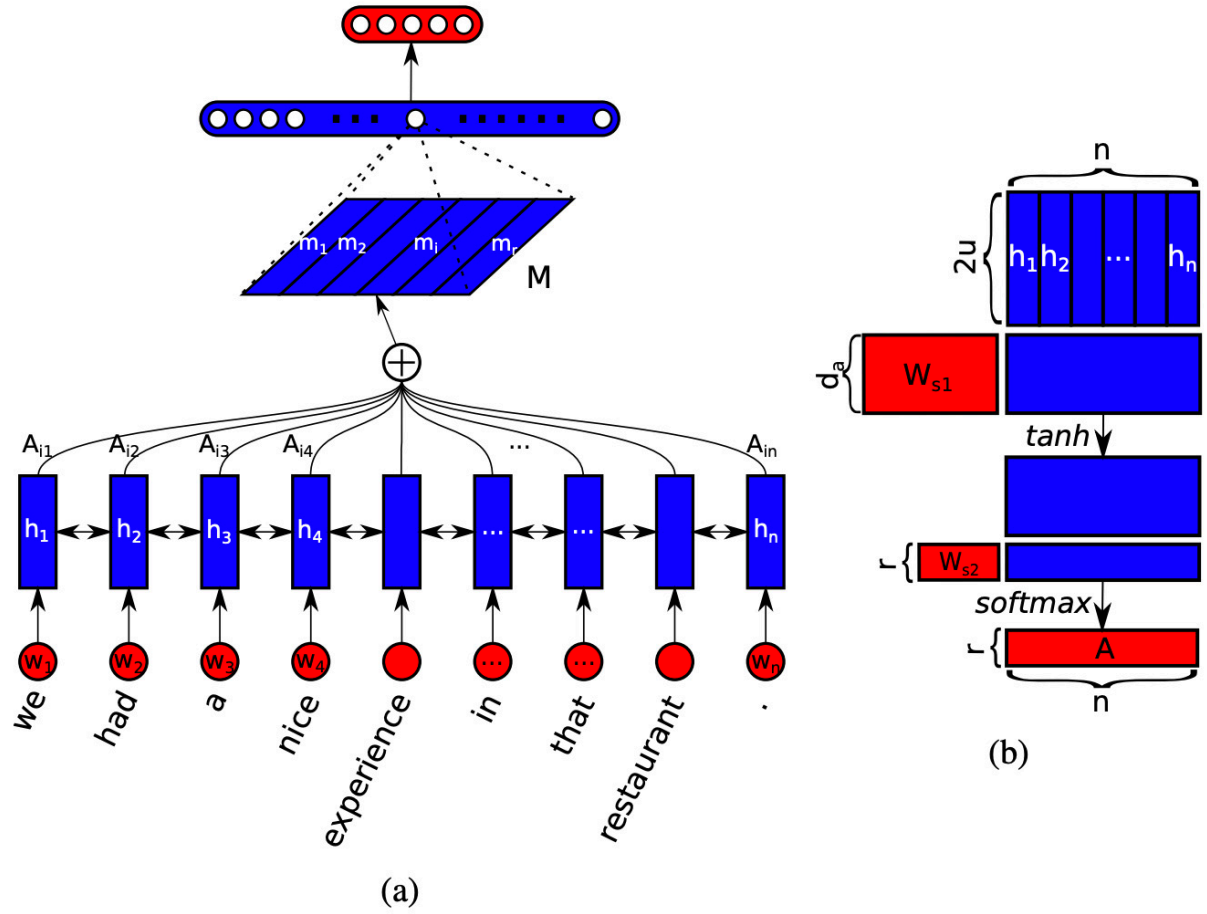


Figure 1: A sample model structure showing the sentence embedding model combined with a fully connected and softmax layer for sentiment analysis (a). The sentence embedding M is computed as multiple weighted sums of hidden states from a bidirectional LSTM ($\mathbf{h_1}, ..., \mathbf{h_n}$), where the summation weights ($A_{i1}, ..., A_{in}$) are computed in a way illustrated in (b). Blue colored shapes stand for hidden representations, and red colored shapes stand for weights, annotations, or input/output.

[19] A structured self-attentive sentence embedding (2017) 1301

2. Background

Position Encoding (PE) 사용

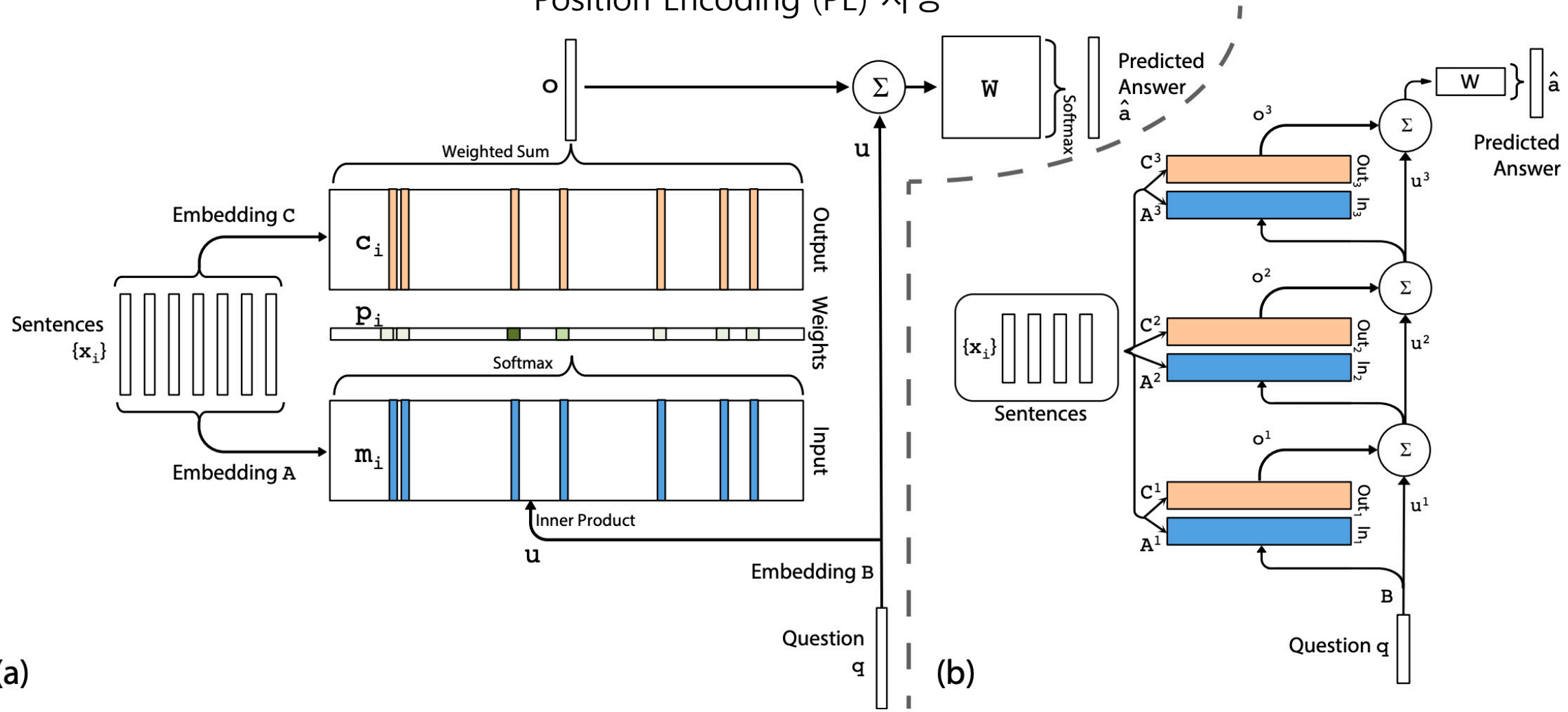
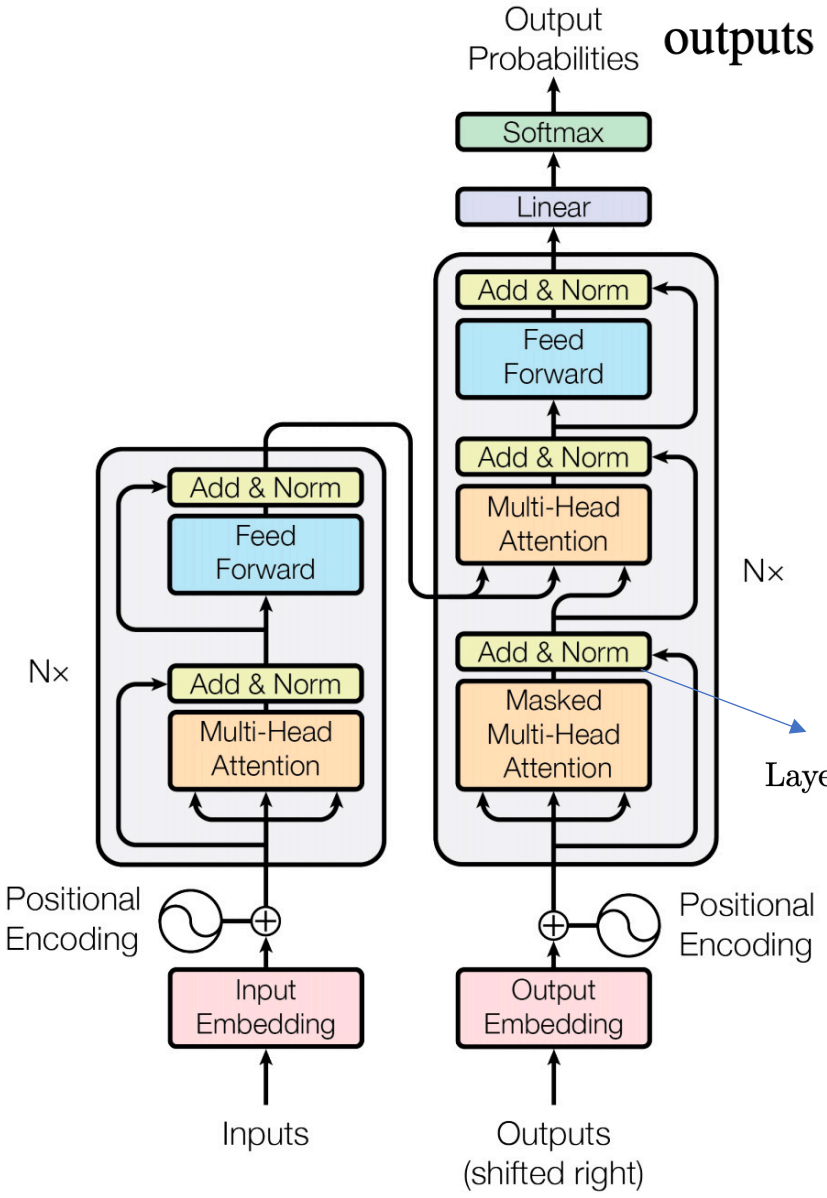


Figure 1: (a): A single layer version of our model. (b): A three layer version of our model. In practice, we can constrain several of the embedding matrices to be the same (see Section 2.2).

[28] End-to-end memory networks (2015) 2053

1) Encoder and Decoder Stacks

outputs of dimension $d_{\text{model}} = 512$.



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

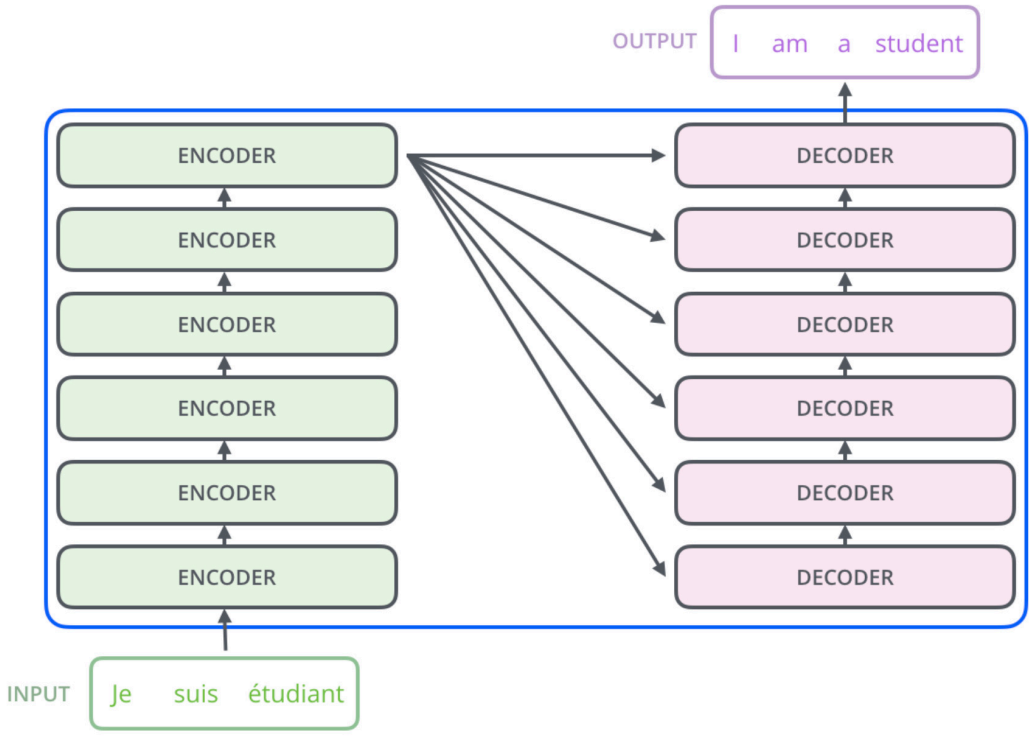


Image from [jalammar]

Figure 1: The Transformer - model architecture.

3.2.1 Scaled Dot-Product Attention

1. Query
2. Key
3. Value

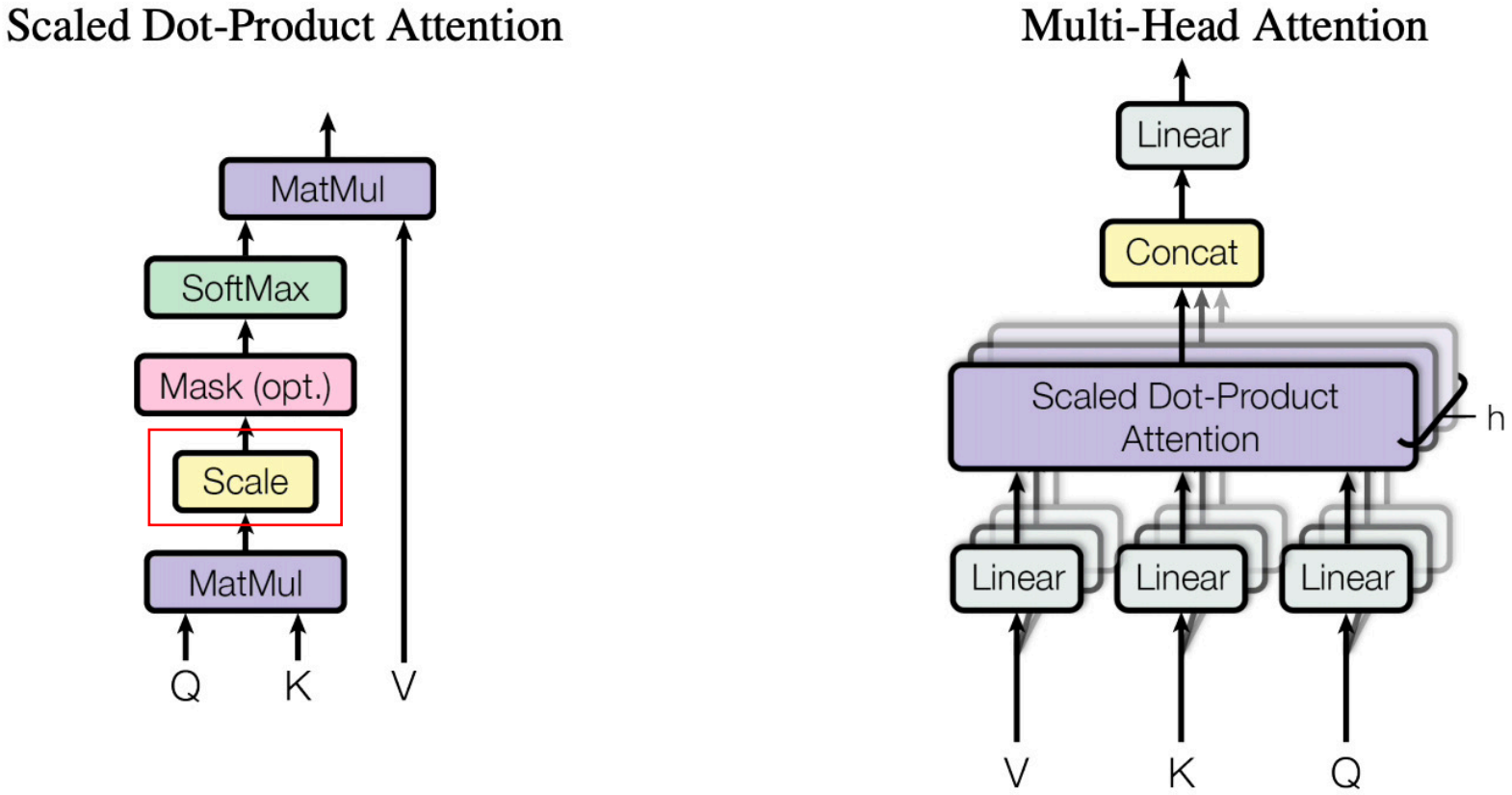


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

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

[4]To illustrate why the dot products get large, assume that the components of q and k are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance d_k.

3.2.2 Multi-Head Attention

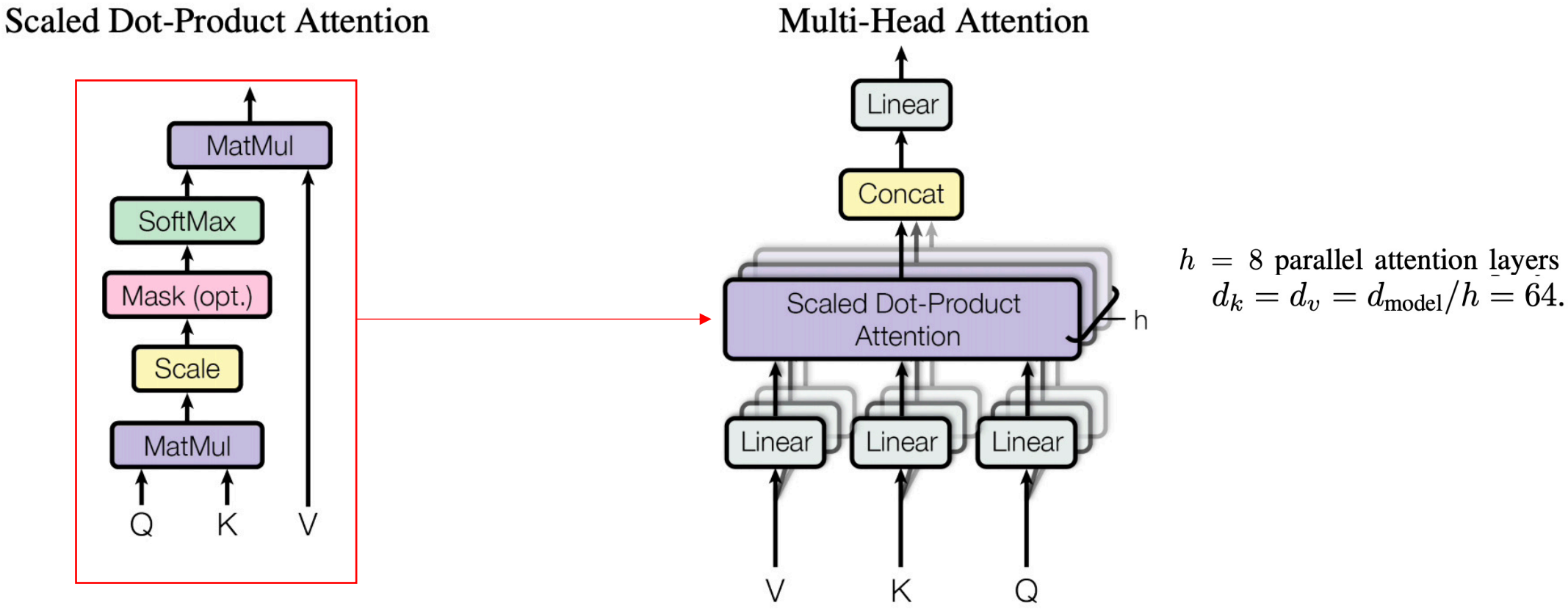
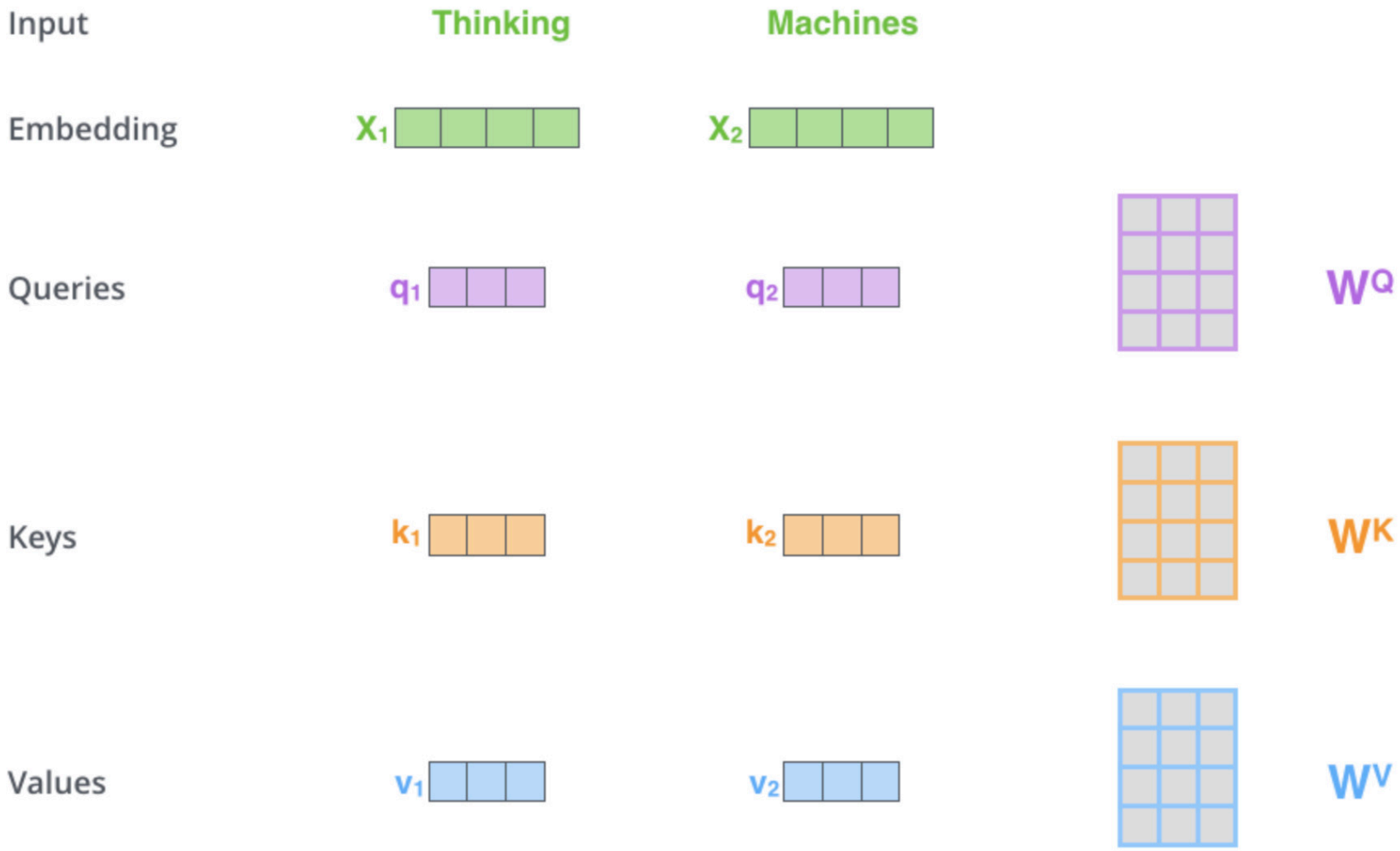


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

3.2.3 Applications of Attention in our Model



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

Image from jalammar

3.2.3 Applications of Attention in our Model

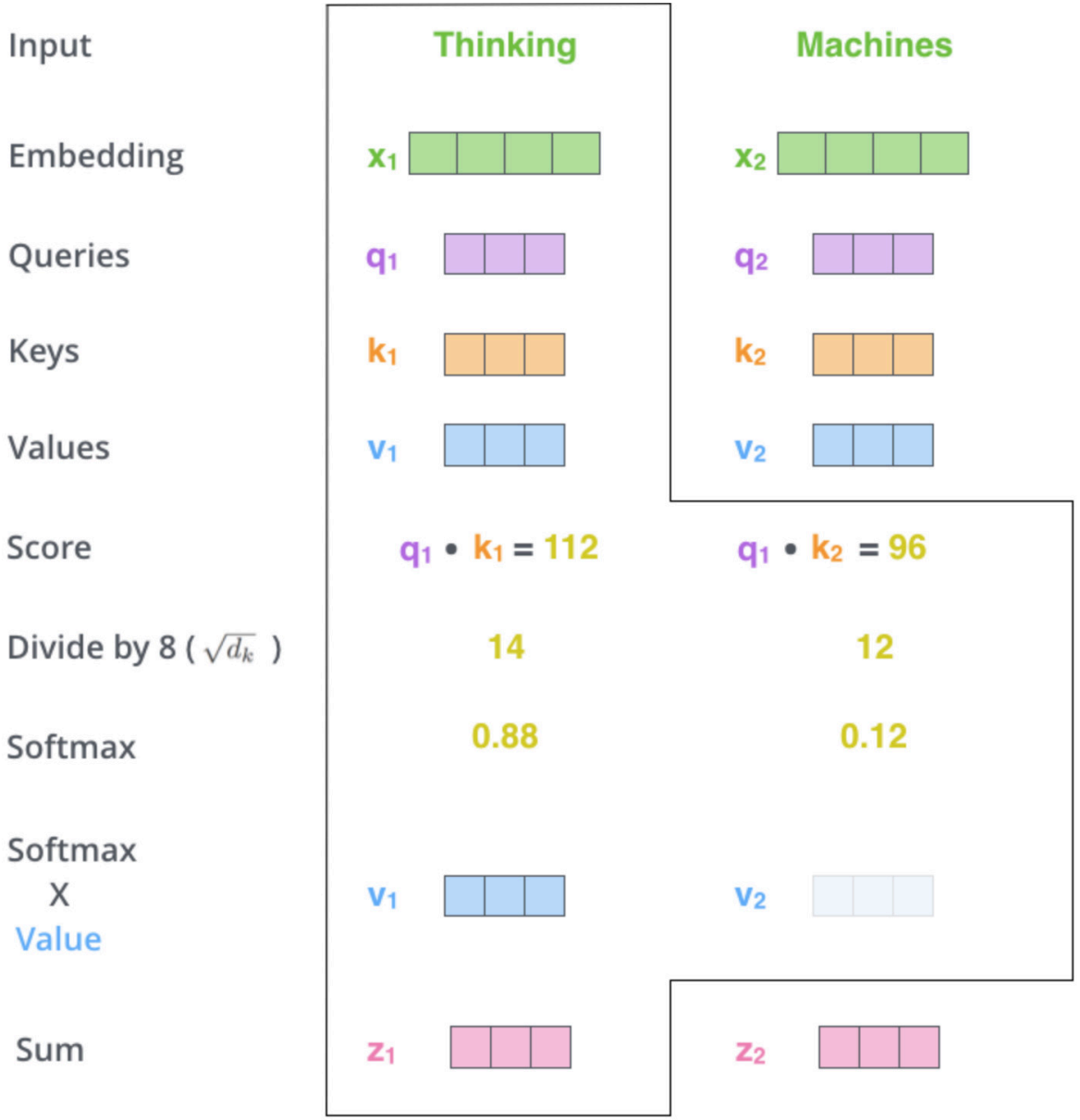
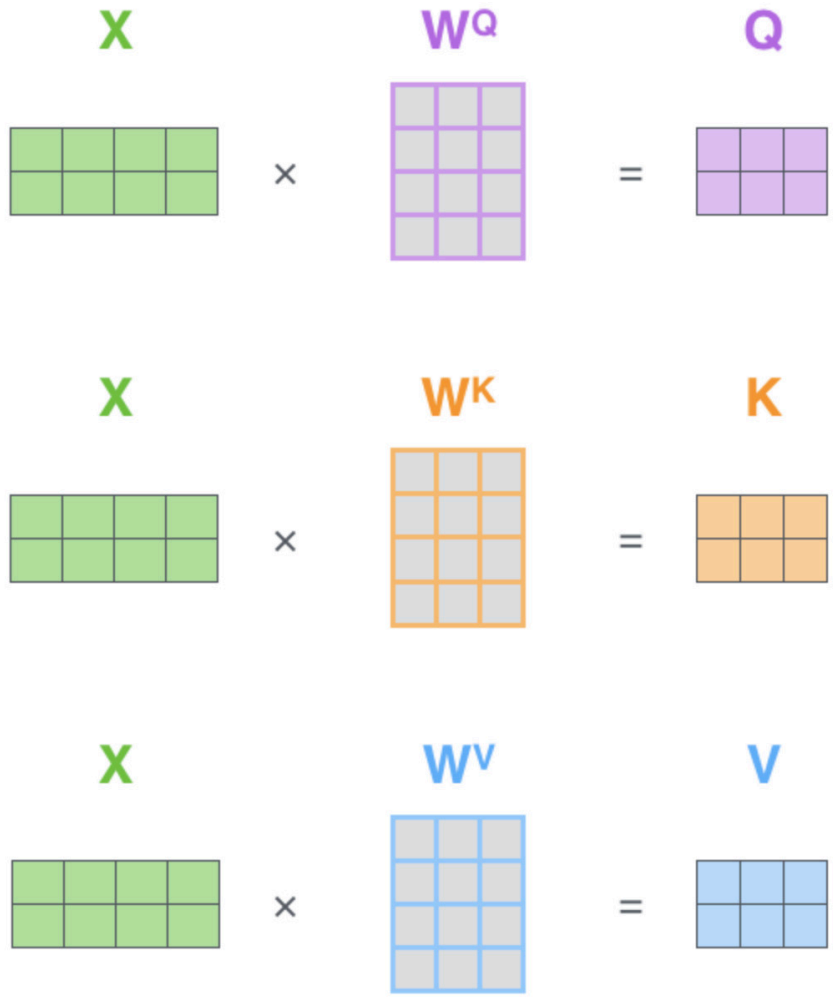


Image from jalammar

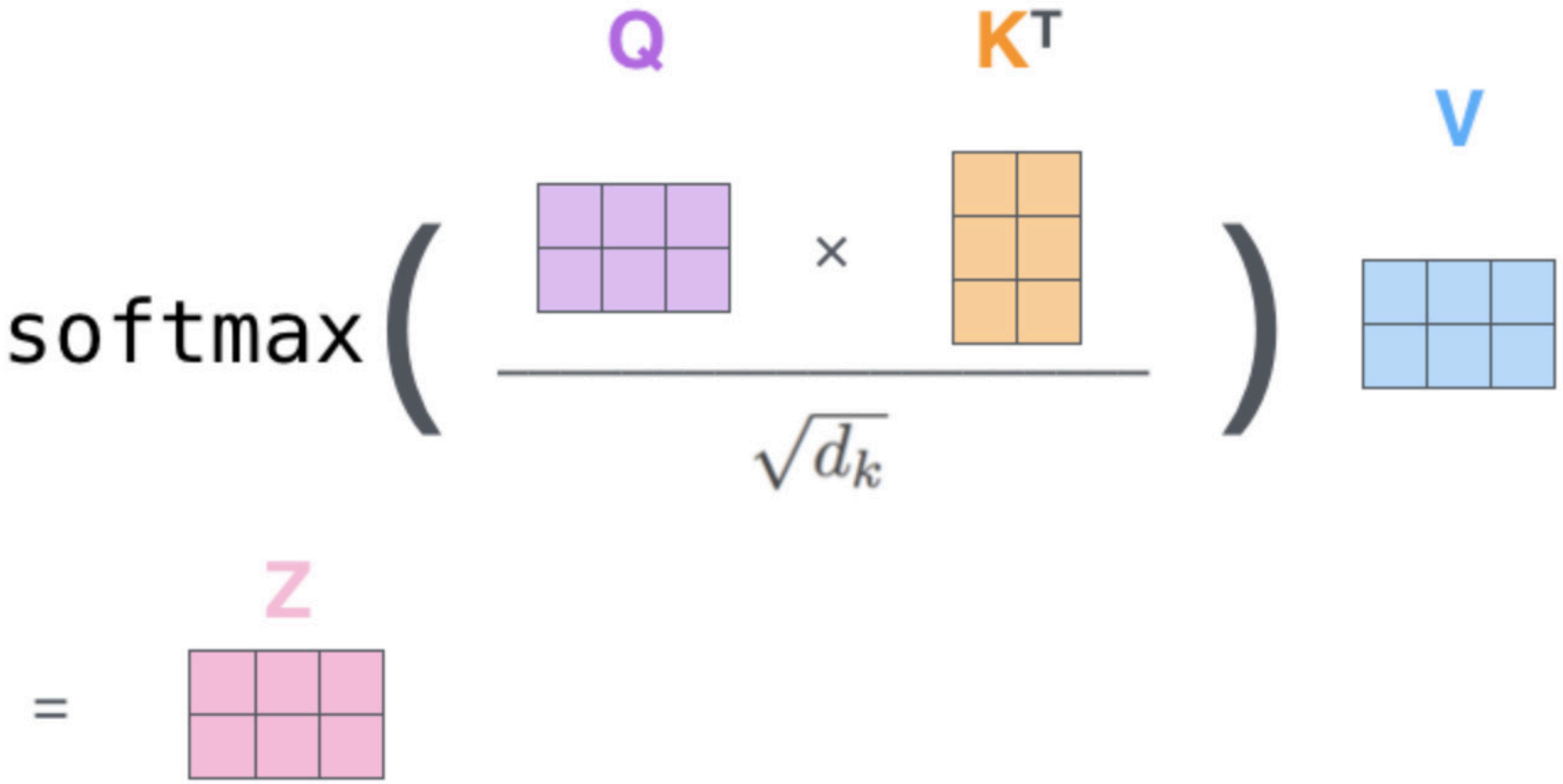
3.2.3 Applications of Attention in our Model



Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

Image from jalammar

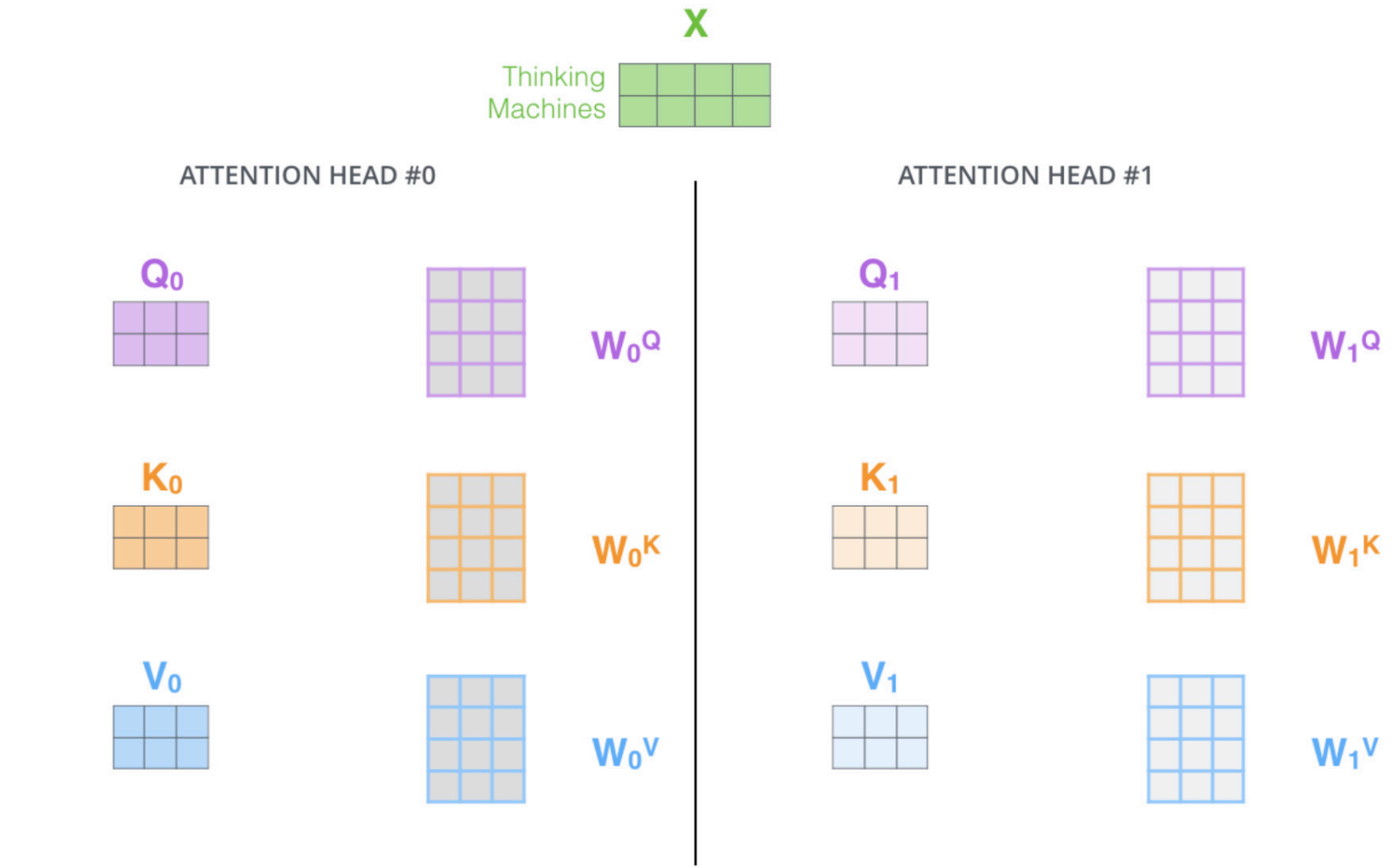
3.2.3 Applications of Attention in our Model



The self-attention calculation in matrix form

Image from jalammar

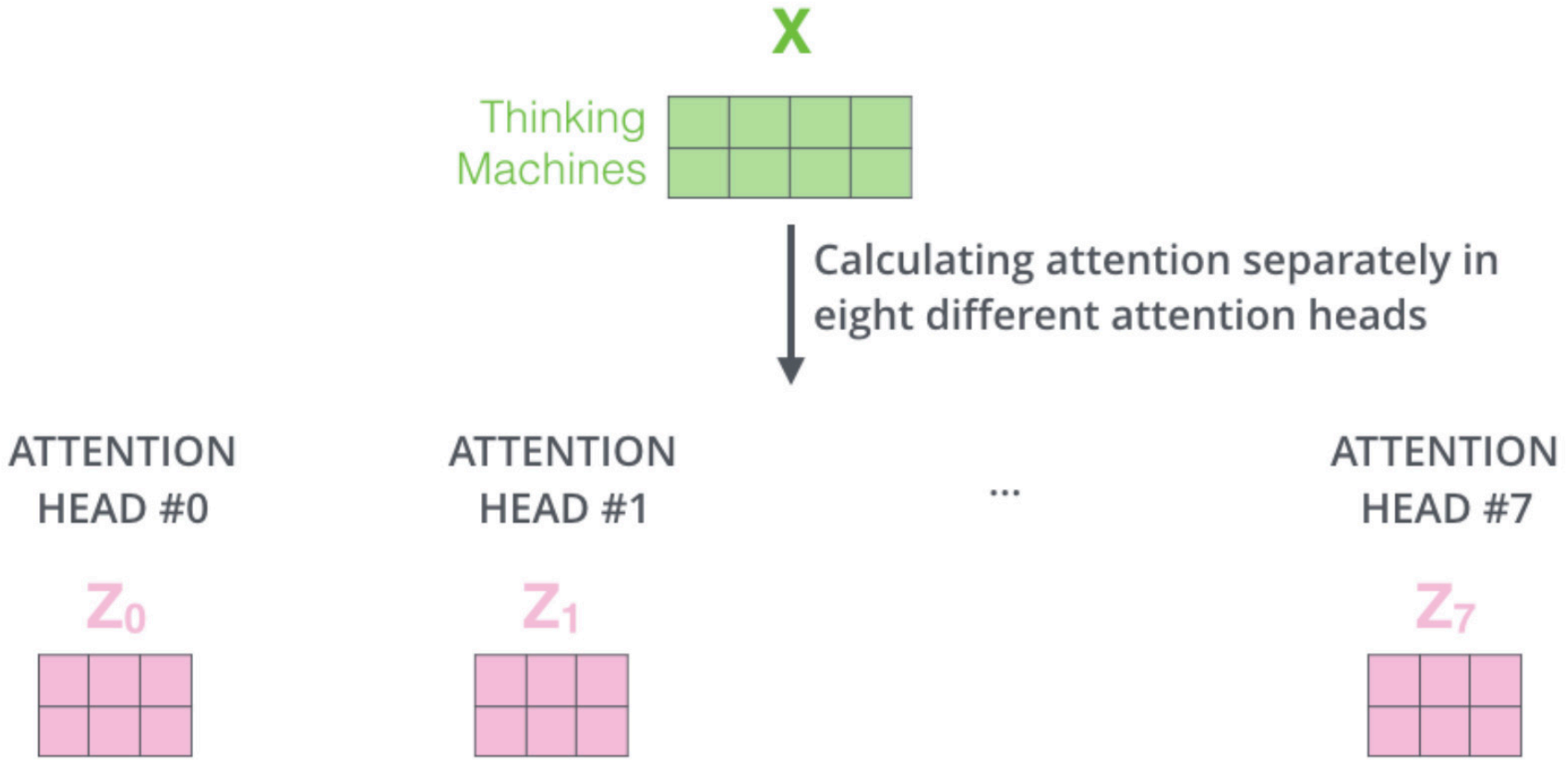
3.2.3 Applications of Attention in our Model



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

Image from jalammar

3.2.3 Applications of Attention in our Model



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

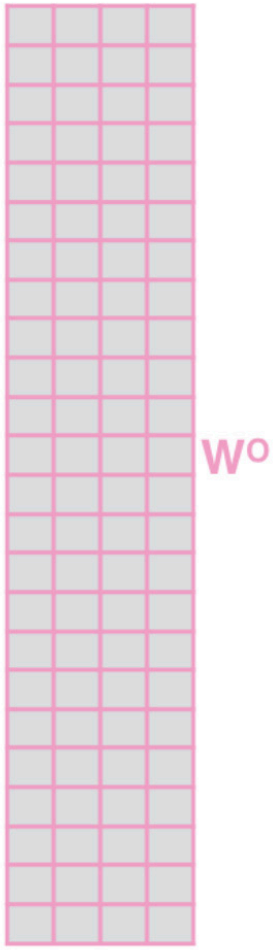
Image from jalammar

3.2.3 Applications of Attention in our Model

1) Concatenate all the attention heads

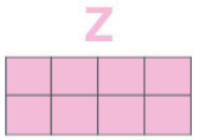
2) Multiply with a weight matrix W^O that was trained jointly with the model

X



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

=

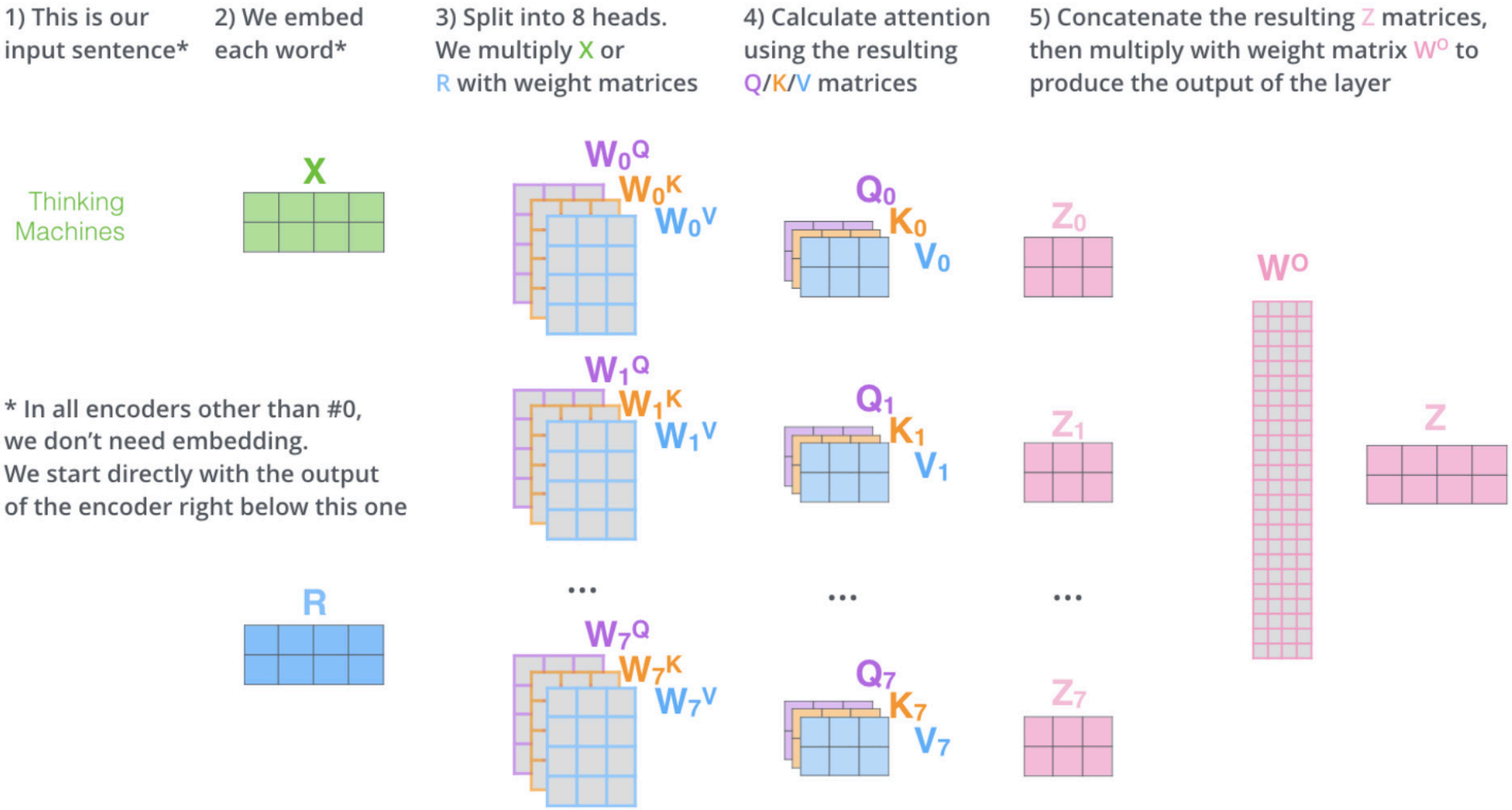


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Image from jalammar

3.2.3 Applications of Attention in our Model

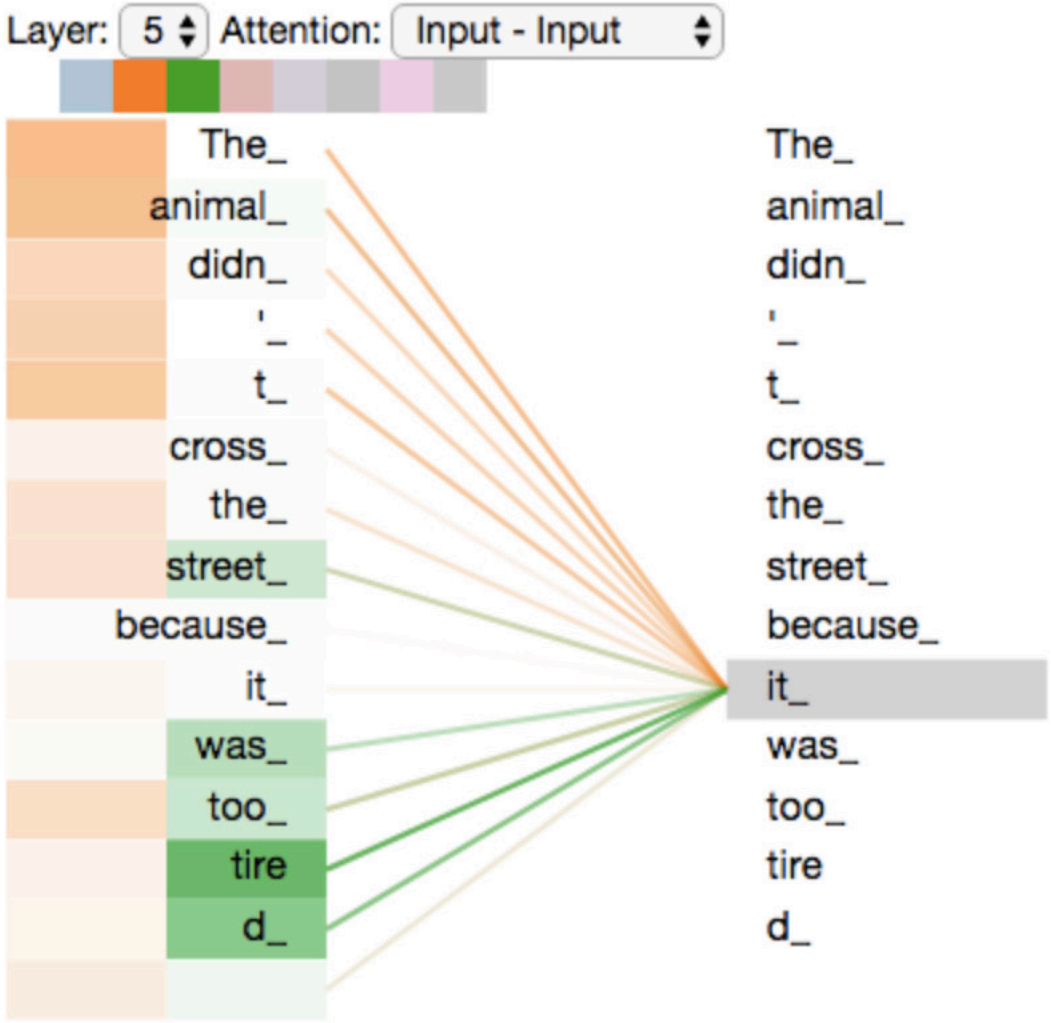


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Image from jalammar

3.2.3 Applications of Attention in our Model



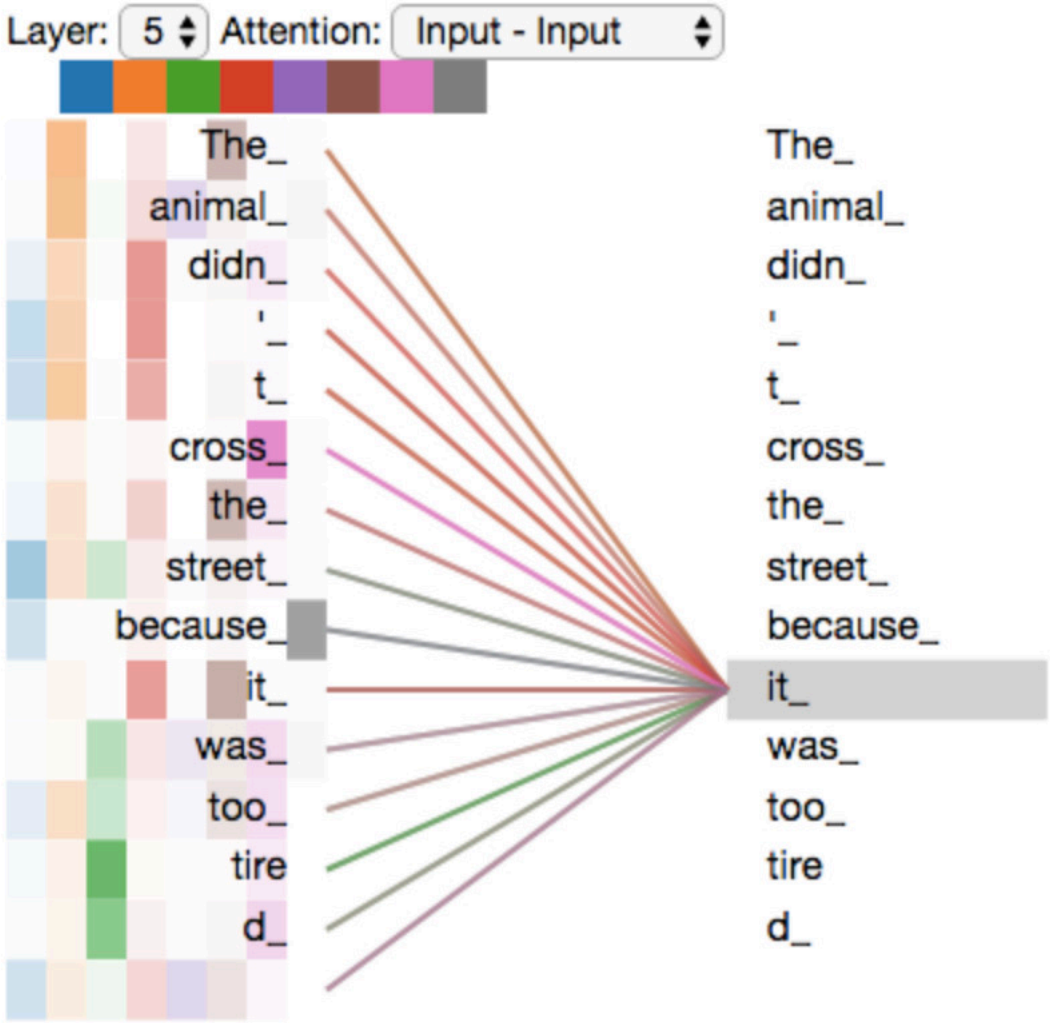


Image from jalammar

3.2.3 Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways:

- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as [31, 2, 8].
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. See Figure 2.

3.2.3 Applications of Attention in our Model

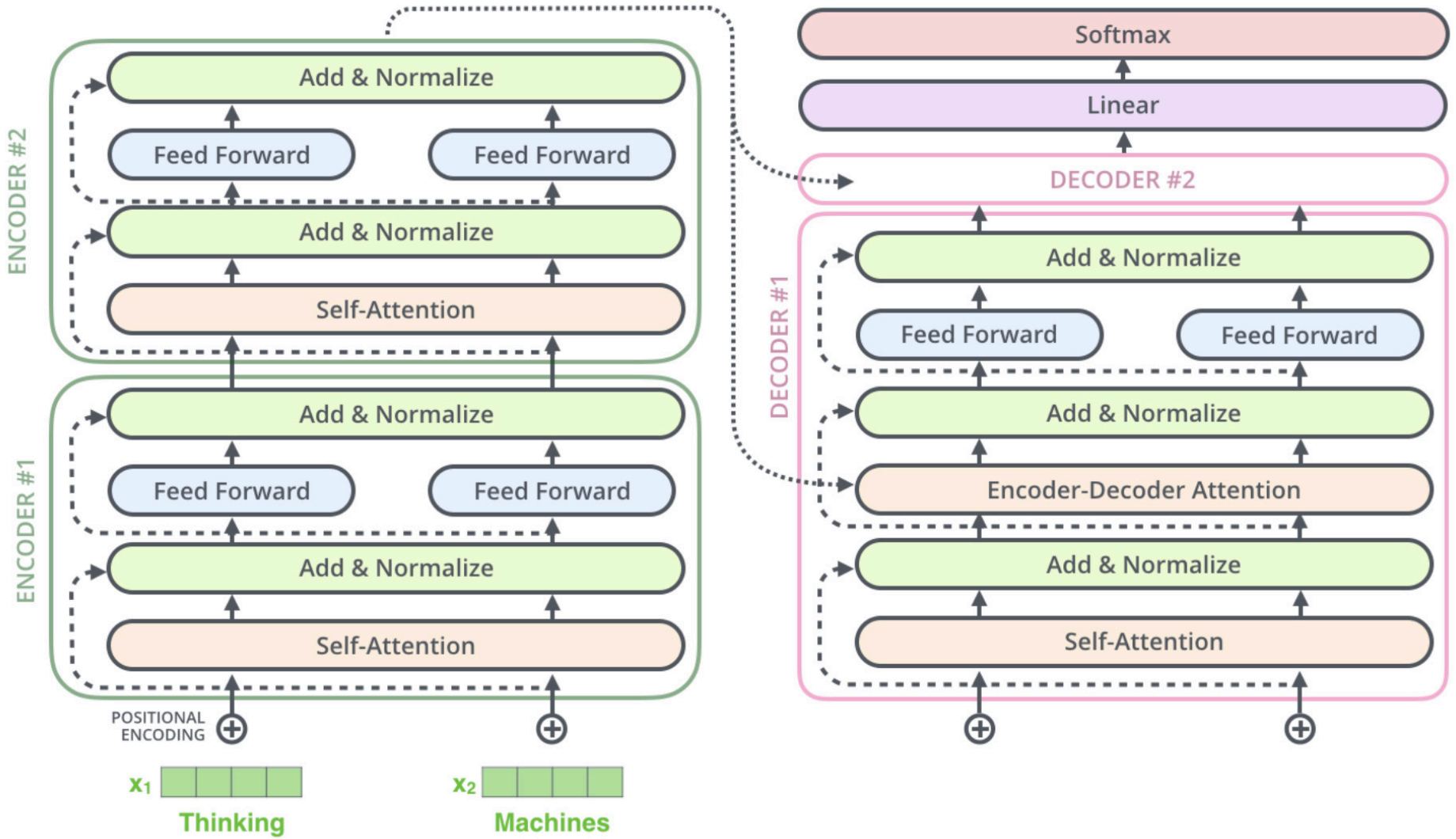


Image from jalammar

3) Position-wise Feed-Forward Networks

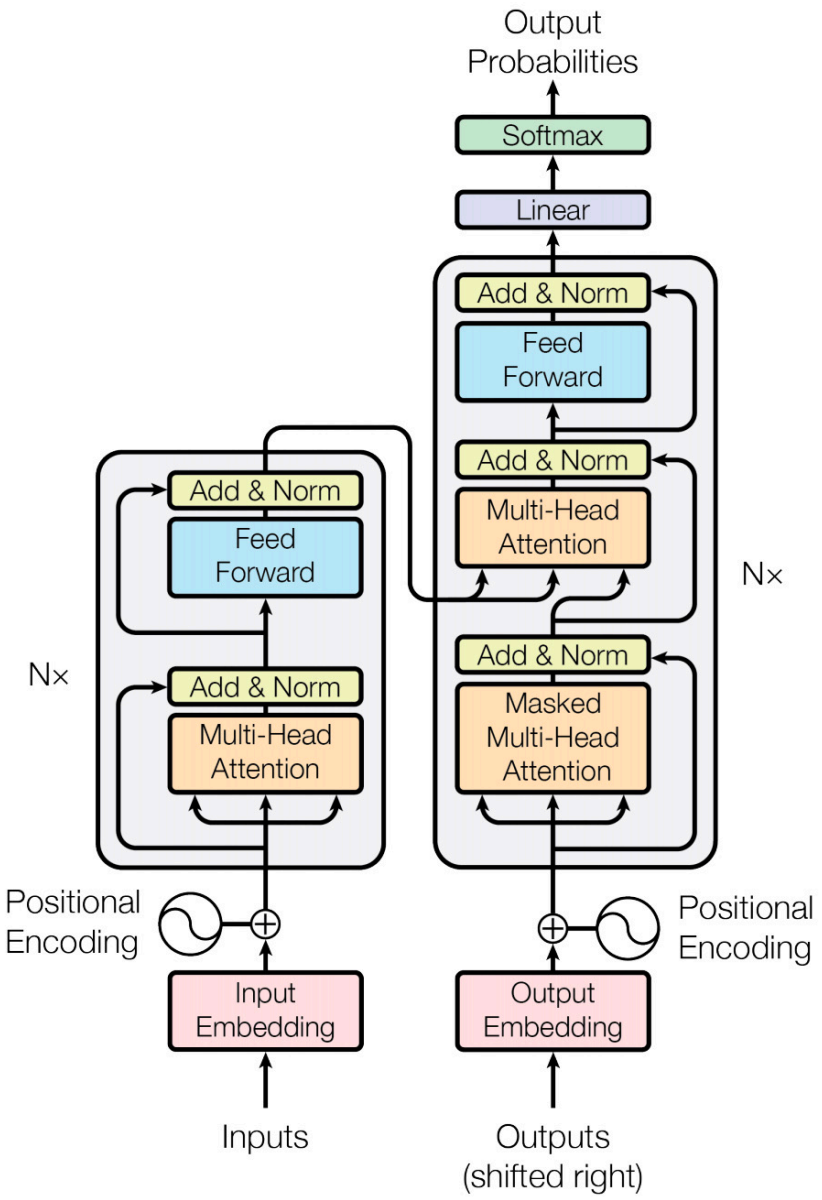
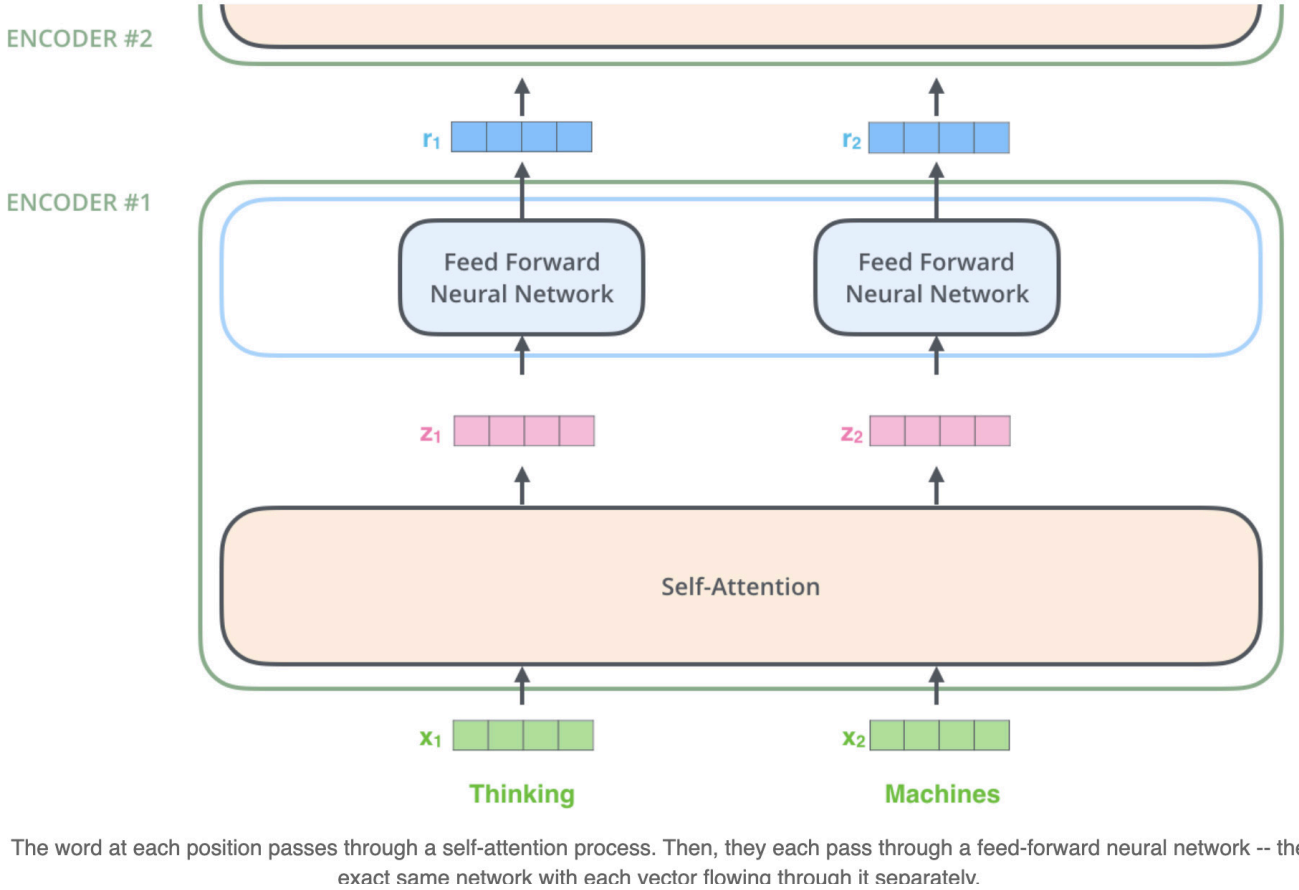


Figure 1: The Transformer - model architecture.



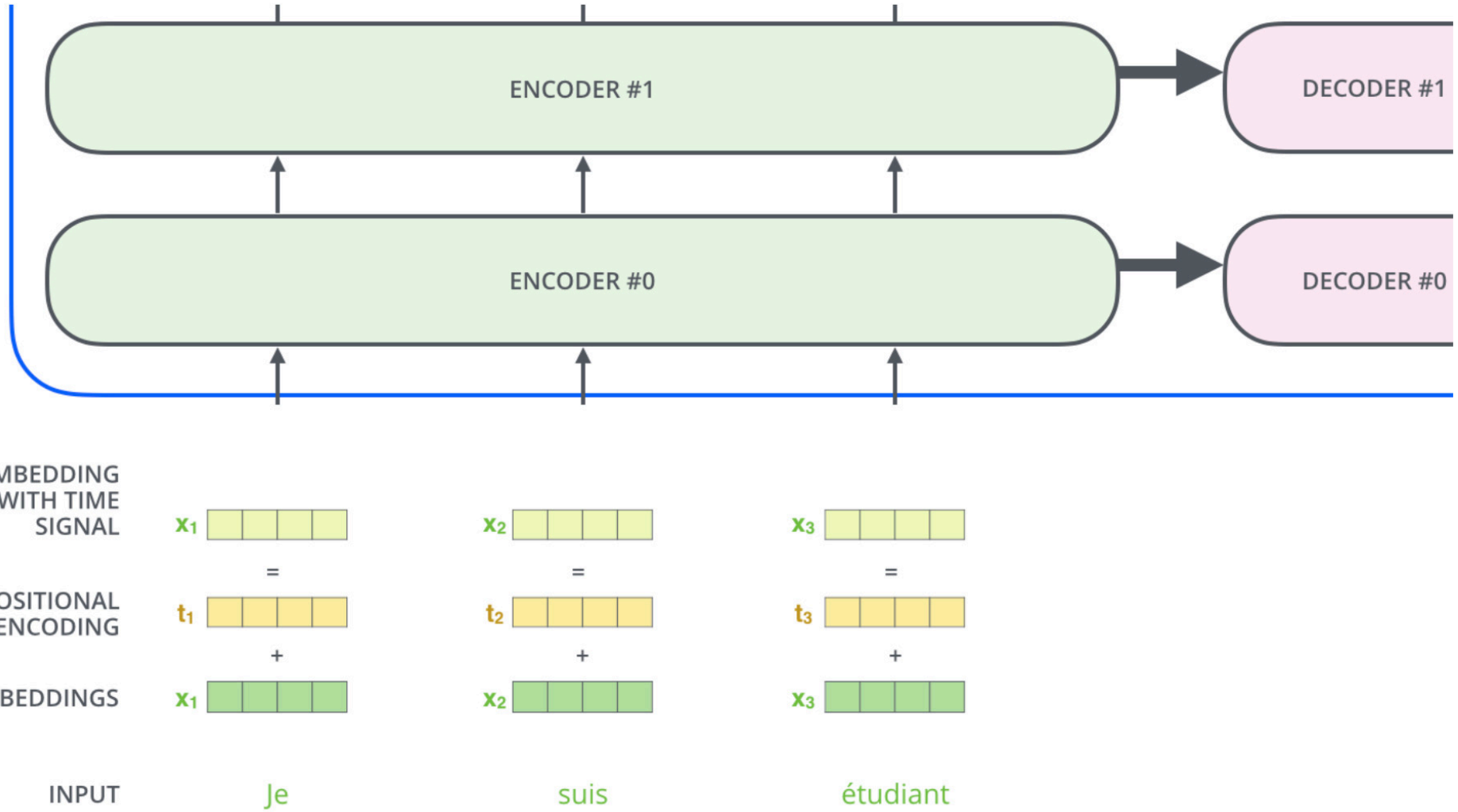
The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

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

The dimensionality of input and output is $d_{model} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [24]. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.

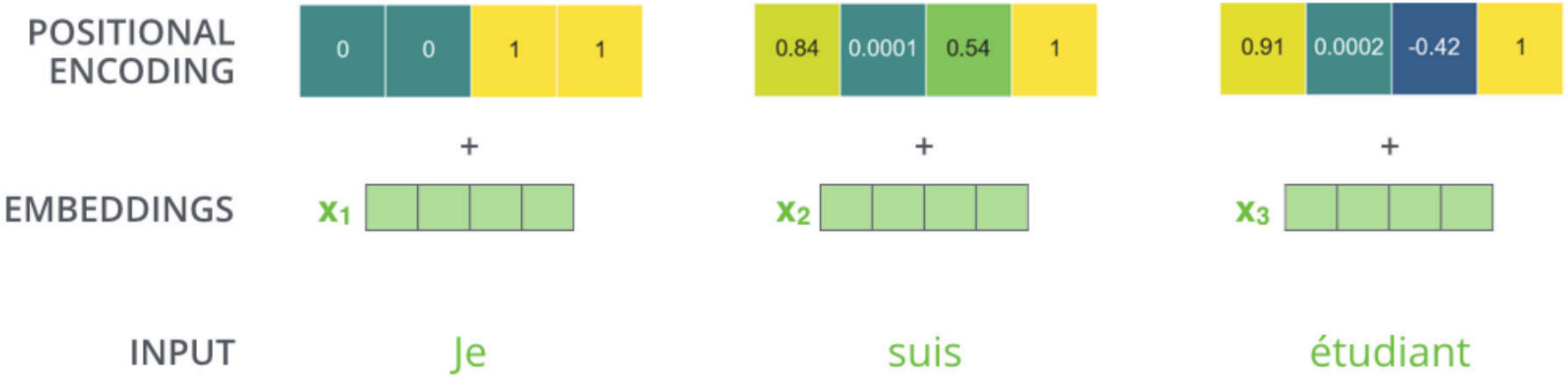
5) Positional Encoding



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

Image from jalammar

5) Positional Encoding



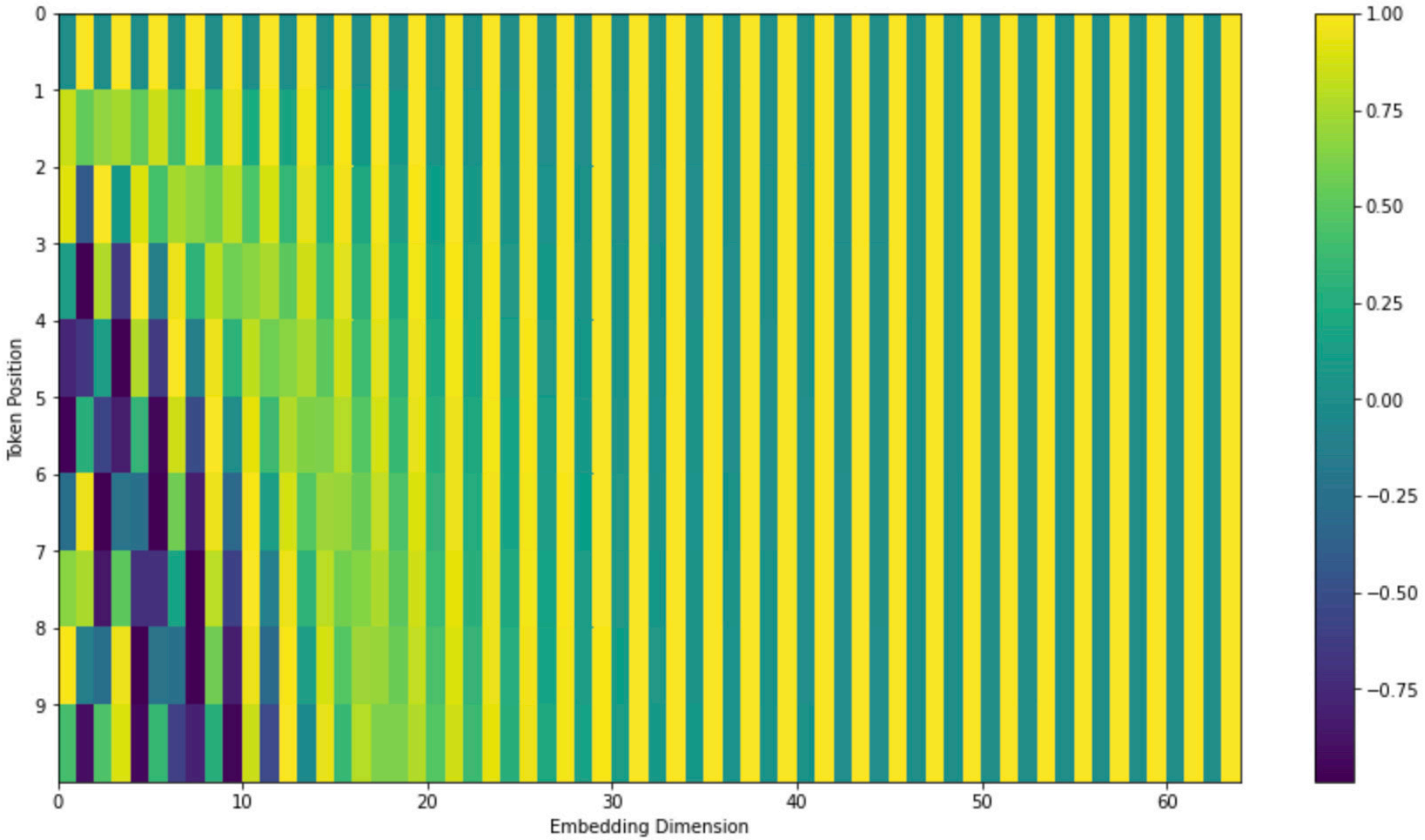
A real example of positional encoding with a toy embedding size of 4

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

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

Image from jalammar

5) Positional Encoding

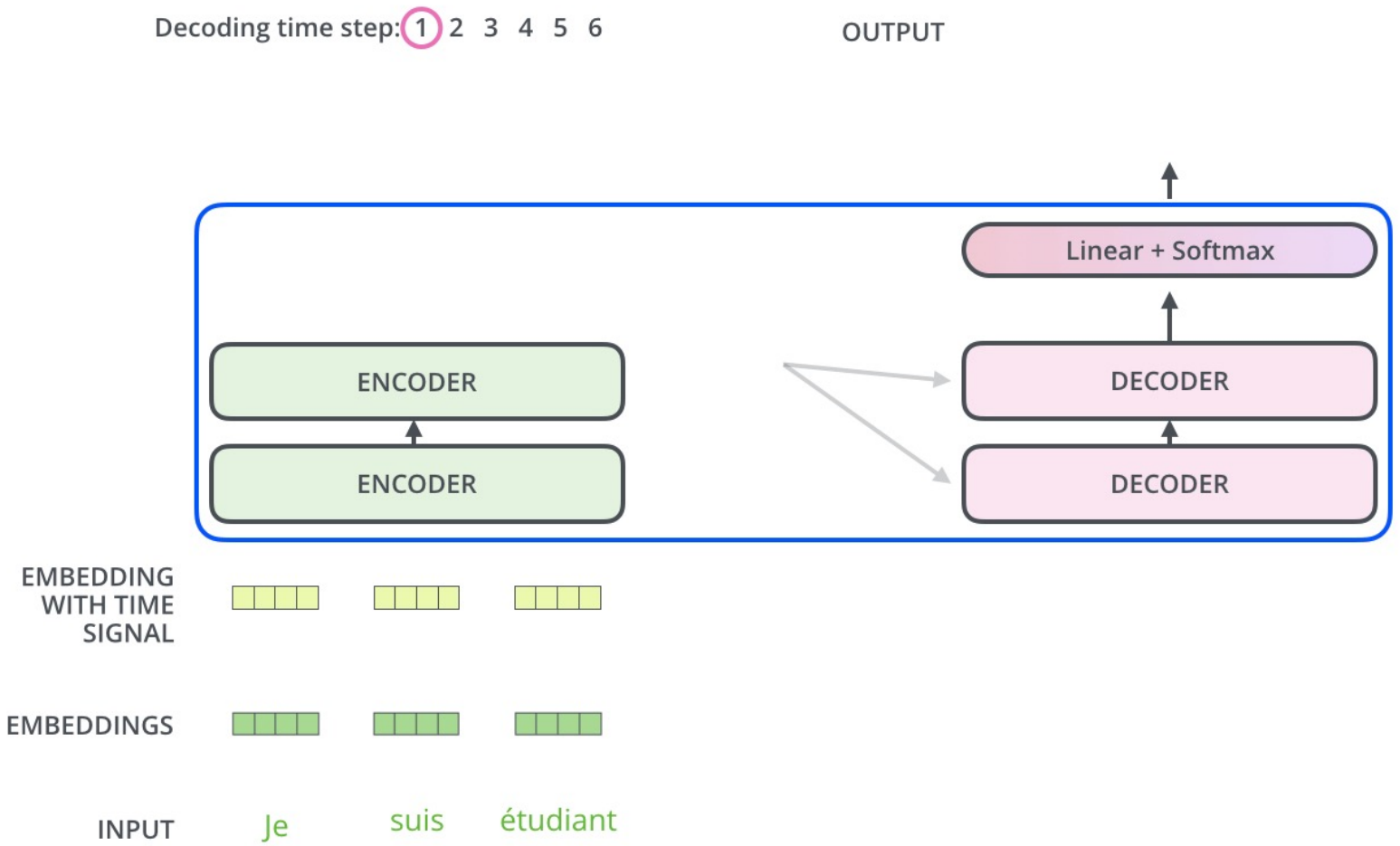


$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

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

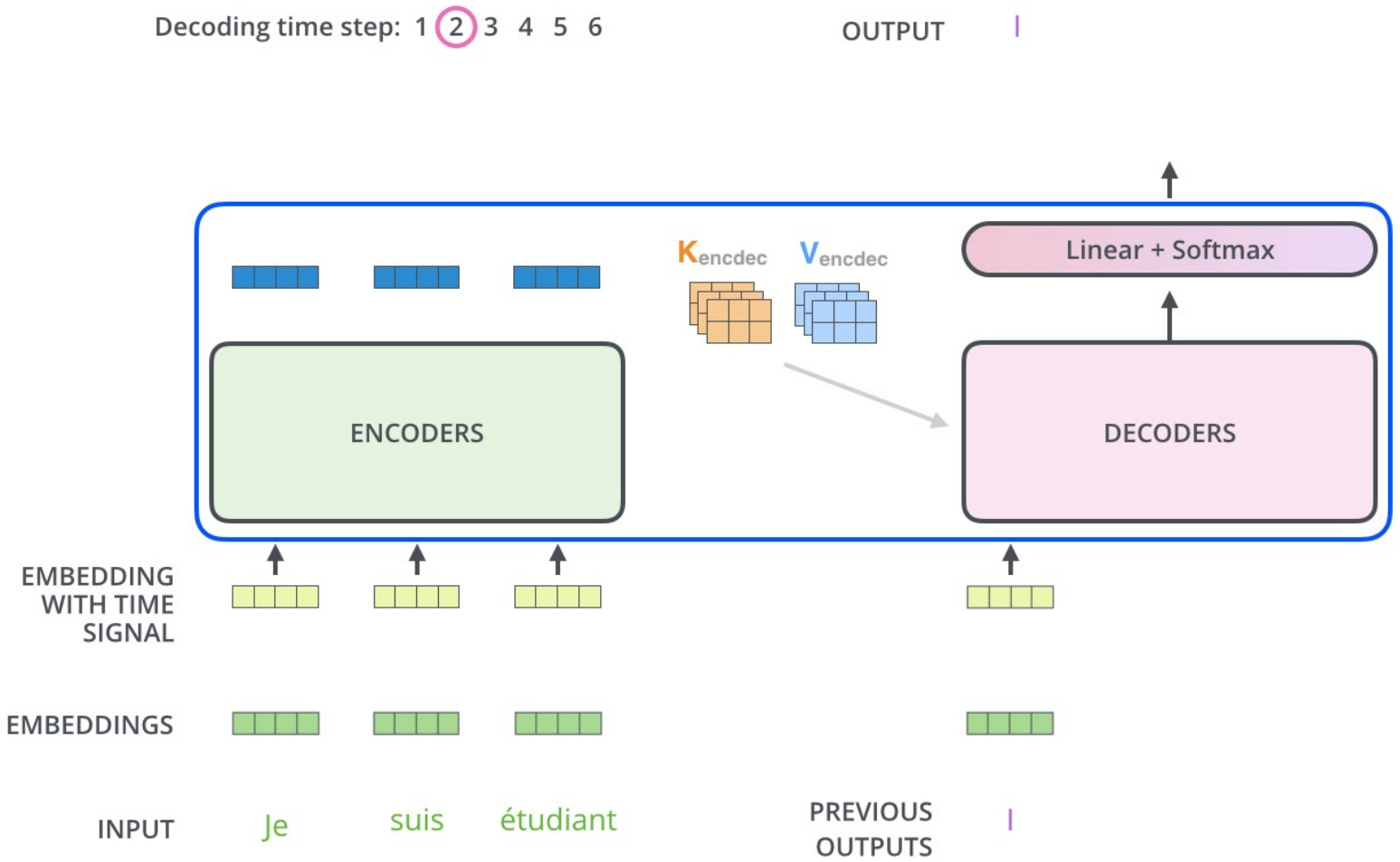
Colab from jalammar

3. Model Architecture



GIF from jalammar

3. Model Architecture



GIF from jalammar

3. Model Architecture

```
def scaled_dot_product_attention(q, k, v, mask):
  matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

  # scale matmul_qk
  dk = tf.cast(tf.shape(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  # add the mask to the scaled tensor.
  if mask is not None:
    scaled_attention_logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k) so that the scores
  # add up to 1.
  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (..., seq_len_q, seq_len_k)

  output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

  return output, attention_weights
```

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

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

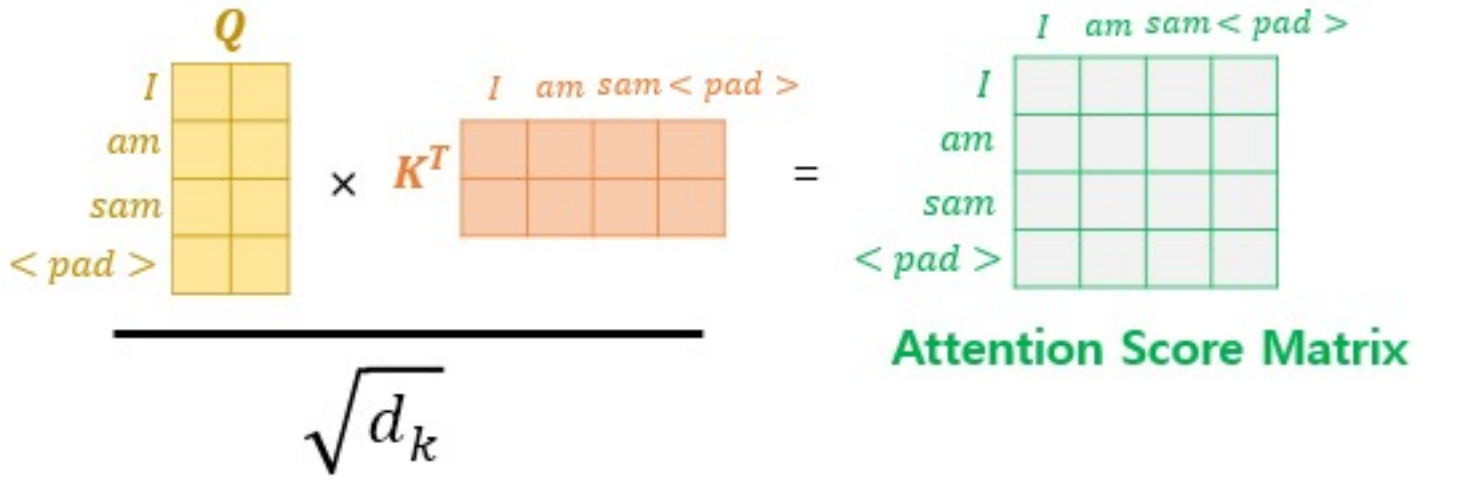
```
def create_padding_mask(seq):
  seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

  # add extra dimensions to add the padding
  # to the attention logits.
  return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)
```

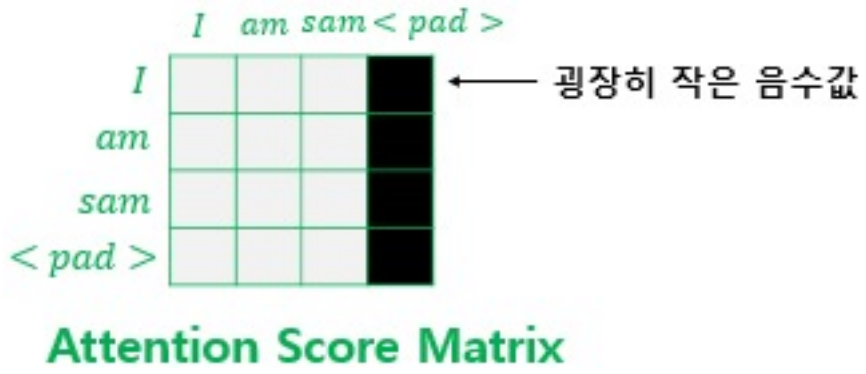
```
x = tf.constant([[7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]])
create_padding_mask(x)
```

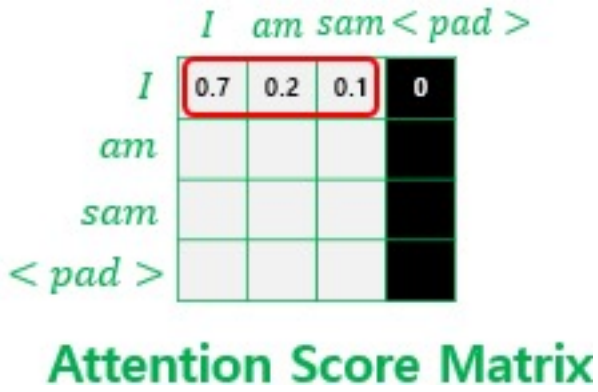
```
<tf.Tensor: shape=(3, 1, 1, 5), dtype=float32, numpy=
array([[[[0., 0., 1., 1., 0.]]],


       [[[0., 0., 0., 1., 1.]]],


       [[[1., 1., 1., 0., 0.]]]], dtype=float32)>
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture







https://wikidocs.net/31379

3. Model Architecture

```
temp_k = tf.constant([[10, 0, 0],
                      [0, 10, 0],
                      [0, 0, 10],
                      [0, 0, 10]], dtype=tf.float32)  # (4, 3)

temp_v = tf.constant([[1, 0],
                      [10, 0],
                      [100, 5],
                      [1000, 6]], dtype=tf.float32)  # (4, 2)
```

```
# This `query` aligns with the second `key`,
# so the second `value` is returned.
temp_q = tf.constant([[0, 10, 0]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
```

```
Attention weights are:
tf.Tensor([[0. 1. 0. 0.]], shape=(1, 4), dtype=float32)
Output is:
tf.Tensor([[10.  0.]], shape=(1, 2), dtype=float32)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
temp_k = tf.constant([[10, 0, 0],
                      [0, 10, 0],
                      [0, 0, 10],
                      [0, 0, 10]], dtype=tf.float32)  # (4, 3)

temp_v = tf.constant([[1, 0],
                      [10, 0],
                      [100, 5],
                      [1000, 6]], dtype=tf.float32)  # (4, 2)
```

```
# This query aligns with a repeated key (third and fourth),
# so all associated values get averaged.
temp_q = tf.constant([[0, 0, 10]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
```

```
Attention weights are:
tf.Tensor([[0.  0.  0.5 0.5]], shape=(1, 4), dtype=float32)
Output is:
tf.Tensor([[550.    5.5]], shape=(1, 2), dtype=float32)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
temp_q = tf.constant([[0, 0, 10],
                      [0, 10, 0],
                      [10, 10, 0]], dtype=tf.float32)  # (3, 3)
print_out(temp_q, temp_k, temp_v)
```

```
Attention weights are:
tf.Tensor(
[[0.  0.  0.5 0.5]
 [0.  1.  0.  0. ]
 [0.5 0.5 0.  0. ]], shape=(3, 4), dtype=float32)
Output is:
tf.Tensor(
[[550.    5.5]
 [ 10.    0. ]
 [  5.5   0. ]], shape=(3, 2), dtype=float32)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
class MultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadAttention, self).__init__()
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    self.depth = d_model // self.num_heads

    self.wq = tf.keras.layers.Dense(d_model)
    self.wk = tf.keras.layers.Dense(d_model)
    self.wv = tf.keras.layers.Dense(d_model)

    self.dense = tf.keras.layers.Dense(d_model)

  def split_heads(self, x, batch_size):
    """Split the last dimension into (num_heads, depth).
    Transpose the result such that the shape is (batch_size, num_heads, seq_len, depth)
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(x, perm=[0, 2, 1, 3])
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
def call(self, v, k, q, mask):
  batch_size = tf.shape(q)[0]

  q = self.wq(q)  # (batch_size, seq_len, d_model)
  k = self.wk(k)  # (batch_size, seq_len, d_model)
  v = self.wv(v)  # (batch_size, seq_len, d_model)

  q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, depth)
  k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, depth)
  v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, depth)

  # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
  # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
  scaled_attention, attention_weights = scaled_dot_product_attention(
      q, k, v, mask)

  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  # (batch_size, seq_len_q, num_

  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))  # (batch_size, seq_len_q, d_model)

  output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)

  return output, attention_weights
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
temp_mha = MultiHeadAttention(d_model=512, num_heads=8)
y = tf.random.uniform((1, 60, 512))  # (batch_size, encoder_sequence, d_model)
out, attn = temp_mha(y, k=y, q=y, mask=None)
out.shape, attn.shape
```

```
(TensorShape([1, 60, 512]), TensorShape([1, 8, 60, 60]))
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
def point_wise_feed_forward_network(d_model, dff):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])
```

```
sample_ffn = point_wise_feed_forward_network(512, 2048)
sample_ffn(tf.random.uniform((64, 50, 512))).shape
```

```
TensorShape([64, 50, 512])
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(EncoderLayer, self).__init__()

    self.mha = MultiHeadAttention(d_model, num_heads)
    self.ffn = point_wise_feed_forward_network(d_model, dff)

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)

  def call(self, x, training, mask):

    attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_model)
    attn_output = self.dropout1(attn_output, training=training)
    out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model)

    ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
    ffn_output = self.dropout2(ffn_output, training=training)
    out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_model)

    return out2
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
sample_encoder_layer = EncoderLayer(512, 8, 2048)

sample_encoder_layer_output = sample_encoder_layer(
    tf.random.uniform((64, 43, 512)), False, None)

sample_encoder_layer_output.shape  # (batch_size, input_seq_len, d_model)
```

```
TensorShape([64, 43, 512])
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(DecoderLayer, self).__init__()

    self.mha1 = MultiHeadAttention(d_model, num_heads)
    self.mha2 = MultiHeadAttention(d_model, num_heads)

    self.ffn = point_wise_feed_forward_network(d_model, dff)

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
    self.dropout3 = tf.keras.layers.Dropout(rate)
```

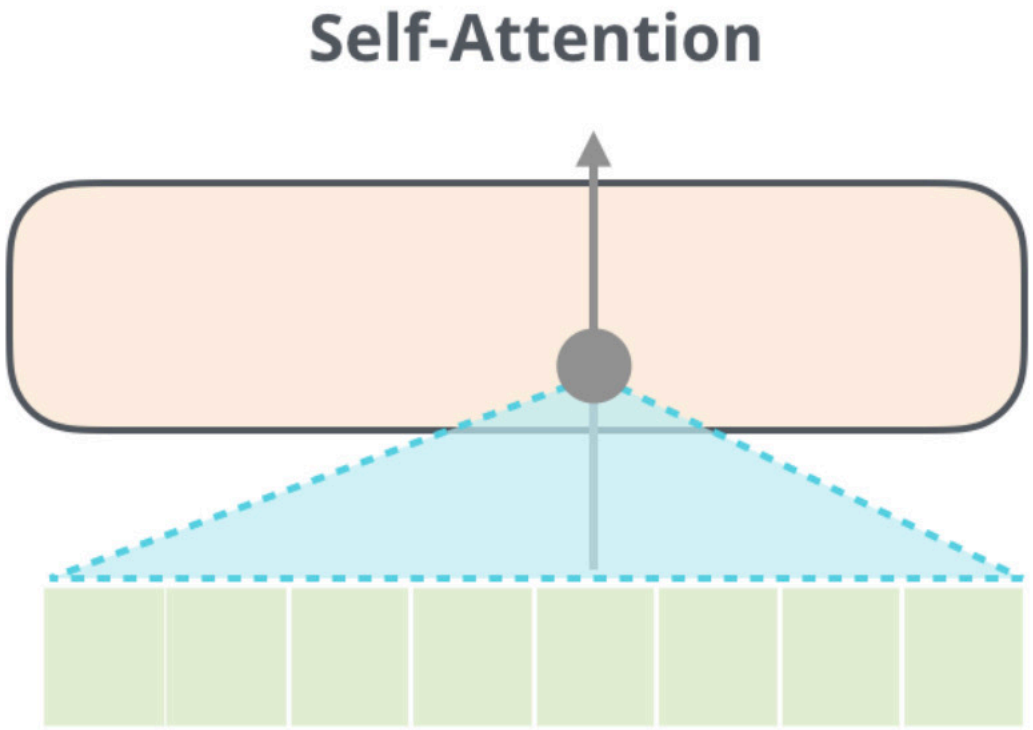
https://www.tensorflow.org/text/tutorials/transformer

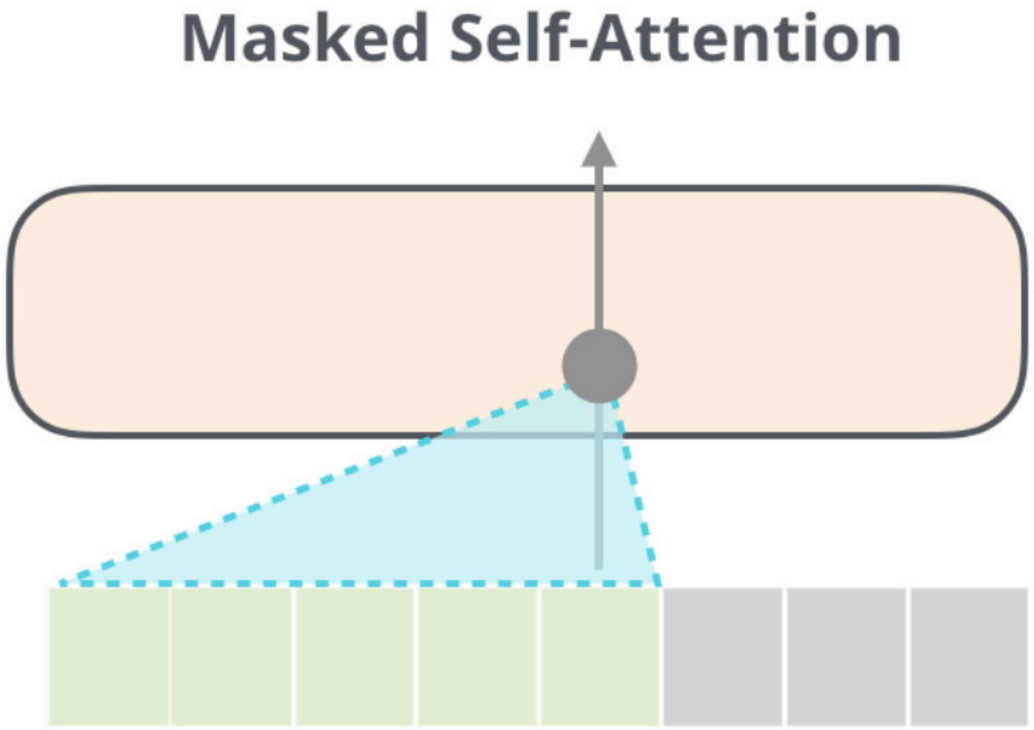
3. Model Architecture

```
def call(self, x, enc_output, training,
         look_ahead_mask, padding_mask):
  # enc_output.shape == (batch_size, input_seq_len, d_model)

  attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_size, target_se
  attn1 = self.dropout1(attn1, training=training)
  out1 = self.layernorm1(attn1 + x)

  attn2, attn_weights_block2 = self.mha2(
      enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len, d_model)
  attn2 = self.dropout2(attn2, training=training)
  out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)

  ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
  ffn_output = self.dropout3(ffn_output, training=training)
  out3 = self.layernorm3(ffn_output + out2)  # (batch_size, target_seq_len, d_model)

  return out3, attn_weights_block1, attn_weights_block2
```

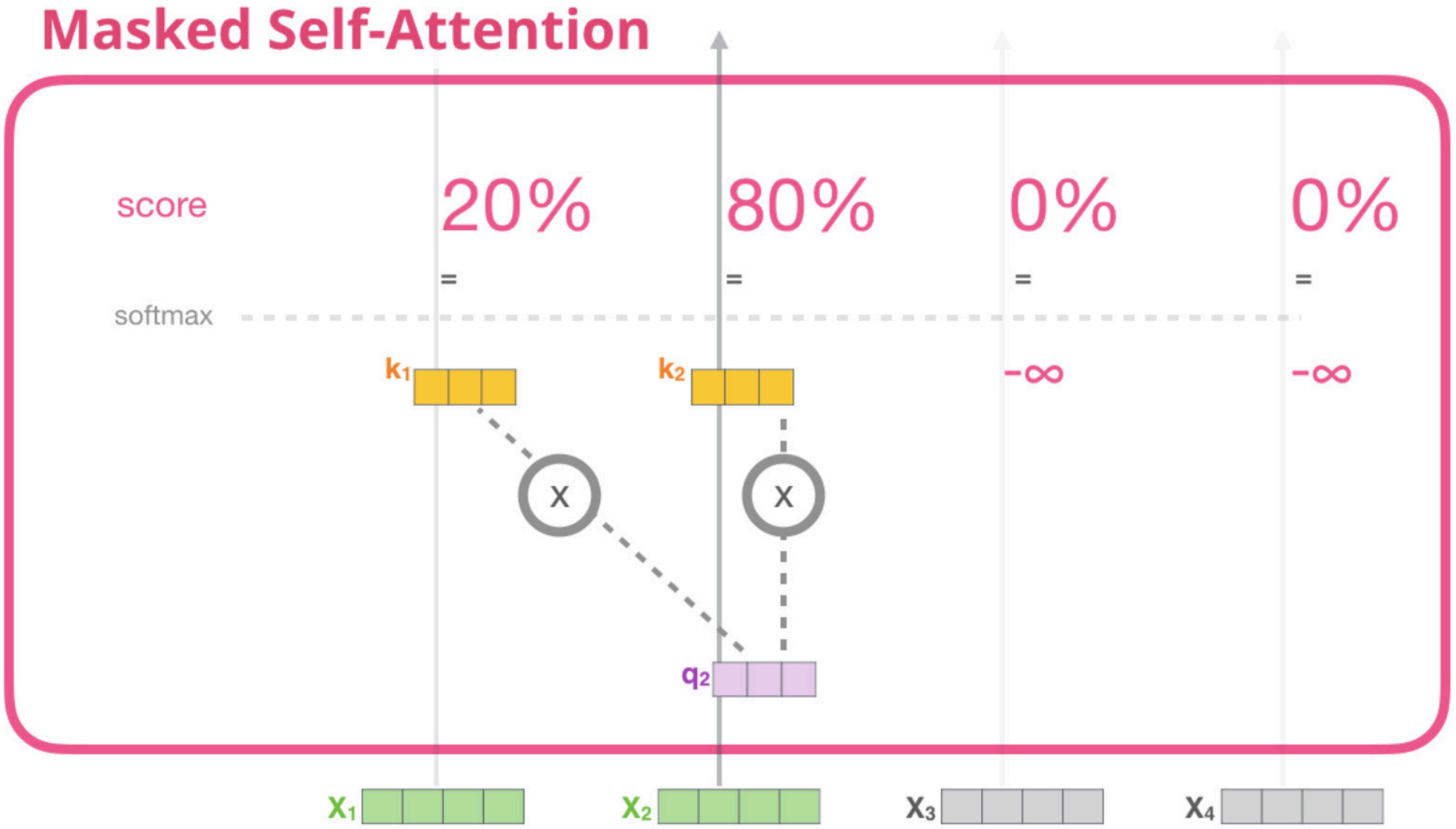
https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture



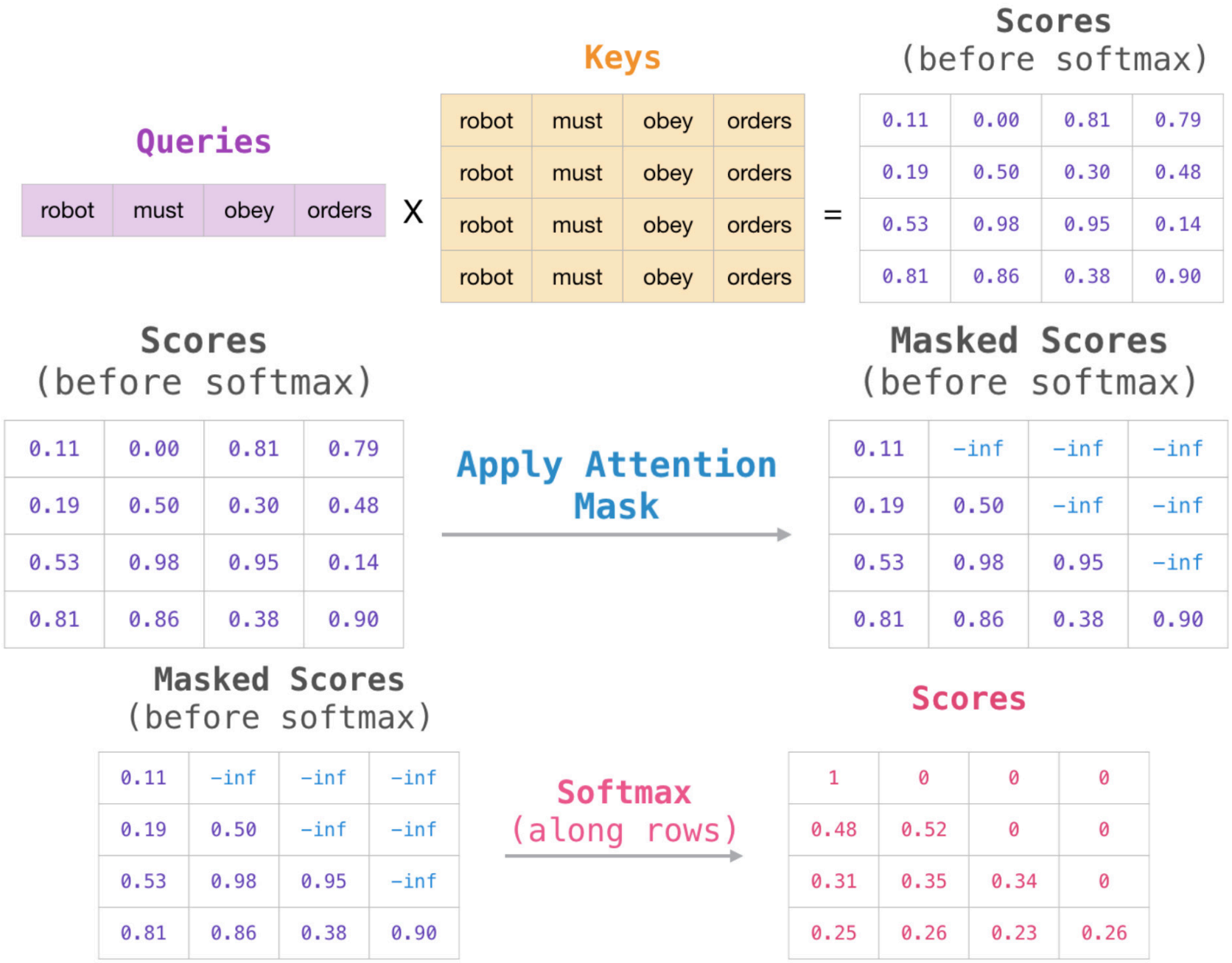


http://jalammar.github.io/illustrated-gpt2/



http://jalammar.github.io/illustrated-gpt2/

3. Model Architecture



3. Model Architecture

```
def create_look_ahead_mask(size):
  mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
  return mask  # (seq_len, seq_len)
```

```
x = tf.random.uniform((1, 3))
temp = create_look_ahead_mask(x.shape[1])
temp
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0., 1., 1.],
       [0., 0., 1.],
       [0., 0., 0.]], dtype=float32)>
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
sample_decoder_layer = DecoderLayer(512, 8, 2048)

sample_decoder_layer_output, _, _ = sample_decoder_layer(
    tf.random.uniform((64, 50, 512)), sample_encoder_layer_output,
    False, None, None)

sample_decoder_layer_output.shape  # (batch_size, target_seq_len, d_model)
```

```
TensorShape([64, 50, 512])
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Encoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding,
                                            self.d_model)

    self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]

    self.dropout = tf.keras.layers.Dropout(rate)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
def call(self, x, training, mask):

  seq_len = tf.shape(x)[1]

  # adding embedding and position encoding.
  x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
  x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
  x += self.pos_encoding[:, :seq_len, :]

  x = self.dropout(x, training=training)

  for i in range(self.num_layers):
    x = self.enc_layers[i](x, training, mask)

  return x  # (batch_size, input_seq_len, d_model)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
sample_encoder = Encoder(num_layers=2, d_model=512, num_heads=8,
                         dff=2048, input_vocab_size=8500,
                         maximum_position_encoding=10000)
temp_input = tf.random.uniform((64, 62), dtype=tf.int64, minval=0, maxval=200)

sample_encoder_output = sample_encoder(temp_input, training=False, mask=None)

print(sample_encoder_output.shape)  # (batch_size, input_seq_len, d_model)
```

```
(64, 62, 512)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)

    self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(rate)
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
def call(self, x, enc_output, training,
         look_ahead_mask, padding_mask):

  seq_len = tf.shape(x)[1]
  attention_weights = {}

  x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
  x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
  x += self.pos_encoding[:, :seq_len, :]

  x = self.dropout(x, training=training)

  for i in range(self.num_layers):
    x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                           look_ahead_mask, padding_mask)

    attention_weights[f'decoder_layer{i+1}_block1'] = block1
    attention_weights[f'decoder_layer{i+1}_block2'] = block2

  # x.shape == (batch_size, target_seq_len, d_model)
  return x, attention_weights
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
sample_decoder = Decoder(num_layers=2, d_model=512, num_heads=8,
                         dff=2048, target_vocab_size=8000,
                         maximum_position_encoding=5000)
temp_input = tf.random.uniform((64, 26), dtype=tf.int64, minval=0, maxval=200)

output, attn = sample_decoder(temp_input,
                              enc_output=sample_encoder_output,
                              training=False,
                              look_ahead_mask=None,
                              padding_mask=None)

output.shape, attn['decoder_layer2_block2'].shape
```

```
(TensorShape([64, 26, 512]), TensorShape([64, 8, 26, 62]))
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
class Transformer(tf.keras.Model):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               target_vocab_size, pe_input, pe_target, rate=0.1):
    super(Transformer, self).__init__()

    self.tokenizer = Encoder(num_layers, d_model, num_heads, dff,
                             input_vocab_size, pe_input, rate)

    self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                           target_vocab_size, pe_target, rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inp, tar, training, enc_padding_mask,
           look_ahead_mask, dec_padding_mask):

    enc_output = self.tokenizer(inp, training, enc_padding_mask)  # (batch_size, inp_seq_len, d_model)

    # dec_output.shape == (batch_size, tar_seq_len, d_model)
    dec_output, attention_weights = self.decoder(
        tar, enc_output, training, look_ahead_mask, dec_padding_mask)

    final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, target_vocab_size)

    return final_output, attention_weights
```

https://www.tensorflow.org/text/tutorials/transformer

3. Model Architecture

```
sample_transformer = Transformer(
    num_layers=2, d_model=512, num_heads=8, dff=2048,
    input_vocab_size=8500, target_vocab_size=8000,
    pe_input=10000, pe_target=6000)

temp_input = tf.random.uniform((64, 38), dtype=tf.int64, minval=0, maxval=200)
temp_target = tf.random.uniform((64, 36), dtype=tf.int64, minval=0, maxval=200)

fn_out, _ = sample_transformer(temp_input, temp_target, training=False,
                               enc_padding_mask=None,
                               look_ahead_mask=None,
                               dec_padding_mask=None)

fn_out.shape  # (batch_size, tar_seq_len, target_vocab_size)
```

```
TensorShape([64, 36, 8000])
```

https://www.tensorflow.org/text/tutorials/transformer

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.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

5. Training

5.1 Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary [31]. Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

5.2 Hardware and Schedule

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models,(described on the bottom line of table 3), step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

5.3 Optimizer

We used the Adam optimizer [17] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \quad (3)$$

This corresponds to increasing the learning rate linearly for the first $warmup\_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup\_steps = 4000$.

5.4 Regularization

We employ three types of regularization during training:

Residual Dropout We apply dropout [27] to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop} = 0.1$.

Label Smoothing During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ [30]. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

BLEU

BLEU(Bilingual Evaluation Understudy)score란 성과지표로 데이터의 X가 순서정보를 가진 단어들(문장)로 이루어져 있고, y 또한 단어들의 시리즈(문장)로 이루어진 경우에 사용되며, 번역을 하는 모델에 주로 사용된다. 3가지 요소를 살펴보자.

- n-gram을 통한 순서쌍들이 얼마나 겹치는지 측정(precision)
- 문장길이에 대한 과적합 보정 (Brevity Penalty)
- 같은 단어가 연속적으로 나올때 과적합 되는 것을 보정(Clipping)

$$BLEU = min(1, \frac{output\ length(\text{예측 문장})}{reference\ length(\text{실제 문장})})(\prod_{i=1}^{4} precision_i)^{\frac{1}{4}}$$

1) Machine Translation

1. n-gram(1~4)을 통한 순서쌍들이 얼마나 겹치는지 측정(precision)

- `예측된 sentence` : 빛이 쐬는 노인은 완벽한 어두운곳에서 잠든 사람과 비교할 때 강박증이 심해질 기회가 훨씬 높았다
- `true sentence` : 빛이 쐬는 사람은 완벽한 어둠에서 잠든 사람과 비교할 때 우울증이 심해질 가능성이 훨씬 높았다

- 1-gram precision: $\frac{\text{일치하는} 1-gram\text{의 수(예측된 } sentence\text{중에서)}}{\text{모든} 1-gram\text{쌍 (예측된 } sentence\text{중에서)}} = \frac{10}{14}$
- 2-gram precision: $\frac{\text{일치하는} 2-gram\text{의 수(예측된 } sentence\text{중에서)}}{\text{모든} 2-gram\text{쌍 (예측된 } sentence\text{중에서)}} = \frac{5}{13}$
- 3-gram precision: $\frac{\text{일치하는} 3-gram\text{의 수(예측된 } sentence\text{중에서)}}{\text{모든} 3-gram\text{쌍 (예측된 } sentence\text{중에서)}} = \frac{2}{12}$
- 4-gram precision: $\frac{\text{일치하는} 4-gram\text{의 수(예측된 } sentence\text{중에서)}}{\text{모든} 4-gram\text{쌍 (예측된 } sentence\text{중에서)}} = \frac{1}{11}$

$$(\prod_{i=1}^{4} precision_i)^{\frac{1}{4}} = (\frac{10}{14} \times \frac{5}{13} \times \frac{2}{12} \times \frac{1}{11})^{\frac{1}{4}}$$

Slide credit dongha-kim

2. 같은 단어가 연속적으로 나올때 과적합 되는 것을 보정(Clipping)

> 위 예제에서 단어 단위로 n-gram을 할 경우 보정할 것이 없지만, 영어의 한 예제에서 1-gram precision를 구하면, 예측된 문장에 중복된 단어들(`the`:3, `more`:2)이 있다. 이를 보정하기 위해 `true sentence`에 있는 중복되는 단어의 max count(`the`:2, `more`:1)를 고려하게 된다. (Clipping). 다른 n-gram도 같은 방식으로 처리하면 된다.

- `예측된 sentence` : The more decomposition the more flavor the food has
- `true sentence` : The more the merrier I always say

- 1-gram precision: $\frac{\text{일치하는} 1-gram \text{의 수(예측된 } sentence \text{중에서)}}{\text{모든} 1-gram \text{쌍 (예측된 } sentence \text{중에서)}} = \frac{5}{9}$
- (보정 후) 1-gram precision: $\frac{\text{일치하는} 1-gram \text{의 수(예측된 } sentence \text{중에서)}}{\text{모든} 1-gram \text{쌍 (예측된 } sentence \text{중에서)}} = \frac{3}{9}$

3. 문장길이에 대한 과적합 보정 (Brevity Penalty)

> 같은 예제에 문장길이에 대한 보정계수를 구하면 다음과 같다.

- `예측된 sentence` : 빛이 쐬는 노인은 완벽한 어두운곳에서 잠듬
- `true sentence` : 빛이 쐬는 사람은 완벽한 어둠에서 잠든 사람과 비교할 때 우울증이 심해질 가능성이 훨씬 높았다

$$min(1, \frac{\text{예측된 } sentence\text{의 길이(단어의 갯수)}}{true\ sentence\text{의 길이(단어의 갯수)}}) = min(1, \frac{6}{14}) = \frac{3}{7}$$

BLEU score

- 아래 예시의 BLEU score를 계산하면 다음과 같다.
- `예측된 sentence` : 빛이 쐬는 노인은 완벽한 어두운곳에서 잠든 사람과 비교할 때 강박증이 심해질 기회가 훨씬 높았다
- `true sentence` : 빛이 쐬는 사람은 완벽한 어둠에서 잠든 사람과 비교할 때 우울증이 심해질 가능성이 훨씬 높았다

$$BLEU = min(1, \frac{output\ length(\text{예측 문장})}{reference\ length(\text{실제 문장})})(\prod_{i=1}^{4} precision_i)^{\frac{1}{4}}$$

$$= min(1, \frac{14}{14}) \times (\frac{10}{14} \times \frac{5}{13} \times \frac{2}{12} \times \frac{1}{11})^{\frac{1}{4}}$$

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

2) Model Variations

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Table 4: The Transformer generalizes well to English constituency parsing (Results are on Section 23 of WSJ)

Parser	Training	WSJ 23 F1
Vinyals & Kaiser el al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser el al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

7. Conclusion