BERT – An In-depth View

PART2

Learning Notes of Senthil Kumar

Agenda

In-depth Technical View (PART2)

- Transformer and Attention
- BERT Word Embedding
- Code Snippets
- Beyond BERT

BERT's Transformer Architecture Overview

- A transformer is a sequence model that gives up the recurrent structure of RNNs for a fully attention-based approach
- Bi-directional Encoder Representations from Transformer (BERT) uses
  - multiple attention layers (L = 12 for $BERT_{base}$ and L = 24 for $BERT_{large}$)
  - each attention layer incorporates multiple attention heads (A = 12 for $BERT_{base}$ and A = 16 for $BERT_{large}$)
- $BERT_{base}$ hidden layer H = 768 dimensions; $BERT_{large}$ H = 1024
  - In other words, BERT produces (contextualized) word embeddings that are of H dimension

Source: BERT GitHub Repository: https://github.com/google-research/bert

What is Attention?

Concept of Attention in Images

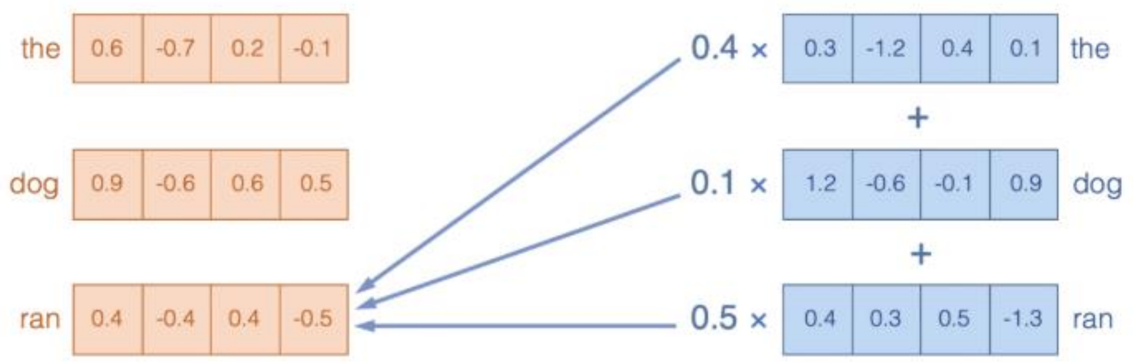
Neural Image Caption Generation with Visual Attention

A woman is throwing a <u>frisbee</u> in a park.

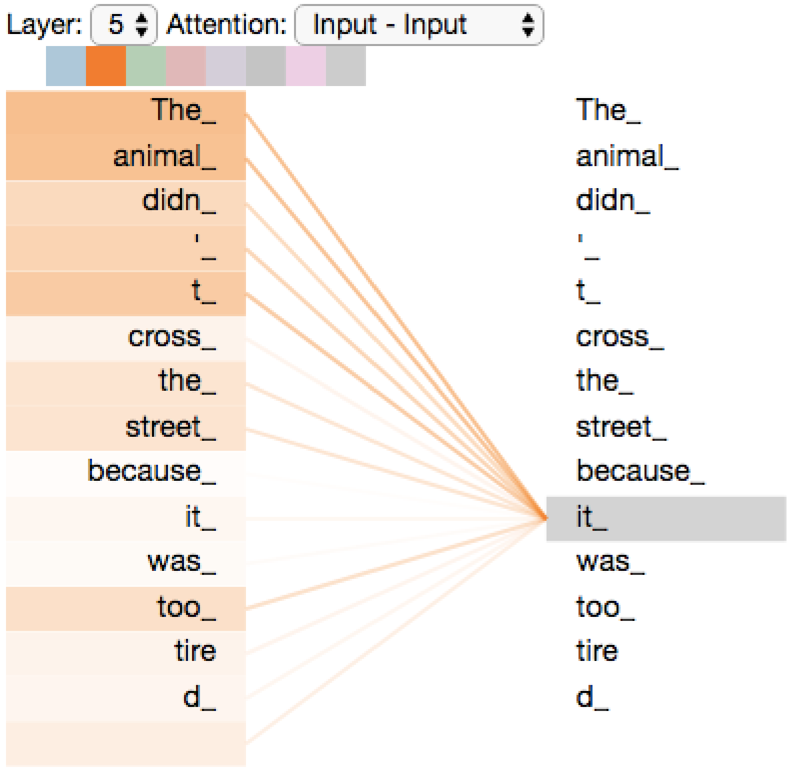
Source:
https://arxiv.org/pdf/1502.03044.pdf

Ok, What is Attention w.r.t Text?

A fancy term for weighted average of input words in a sequence of text



Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words (e.g.: 'The', 'animal') into the one (e.g.: 'it') we're currently processing in the same input sequence.



Source:

- https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1
- http://jalammar.github.io/illustrated-transformer/

Why Attention?

One main limitation of Seq2Seq Models – Long range dependencies

- Sequence-to-sequence learning is that it requires to compress the entire content of the source sequence into a fixed-size vector
-
- Attention solves this limitation by allowing the decoder to look back at the source sequence hidden states, which are then provided as a **weighted average** as additional input to the decoder
- But still?
  ==> Attention == Fuzzy Memory ?!
  ==> Doesn't work that well on longer sequences!

- Attention can be seen as a form of fuzzy memory where the memory consists of the past hidden states of the model, with the model choosing what to retrieve from memory.

Memory networks have more explicit/active memory.

Memory-based models are typically applied to tasks, where retaining information over longer time spans should be useful such as language modelling and reading comprehension.

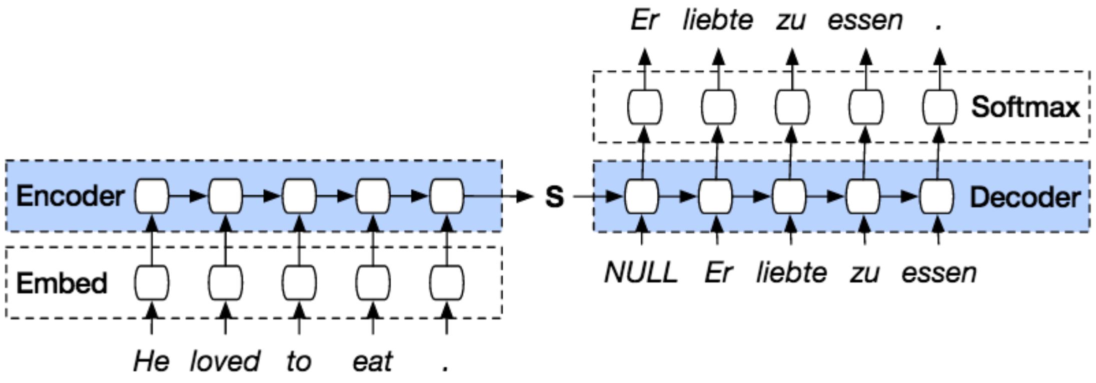
**However, active memory has not improved over attention for most natural language processing tasks, in particular for machine translation.
**Active memory and attention are not exclusive. May be future holds promise for these.

Source:
Advanced NLP and Deep Learning Course on Udemy (by Lazy Programmer)
**https://papers.nips.cc/paper/6295-can-active-memory-replace-attention.pdf

How is Attention incorporated in a Seq2Seq Model?

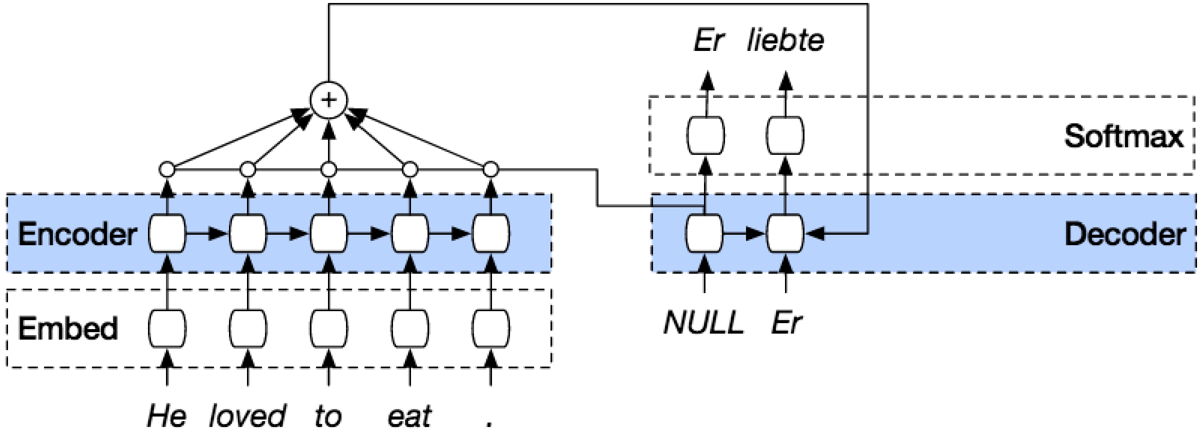
Evolution of Seq2Seq Model in NMT

Typical Seq2Seq Model



{ Encoder + Decoder }
Recurrent Structure: Processes all input elements sequentially

Seq2Seq Model with Attention

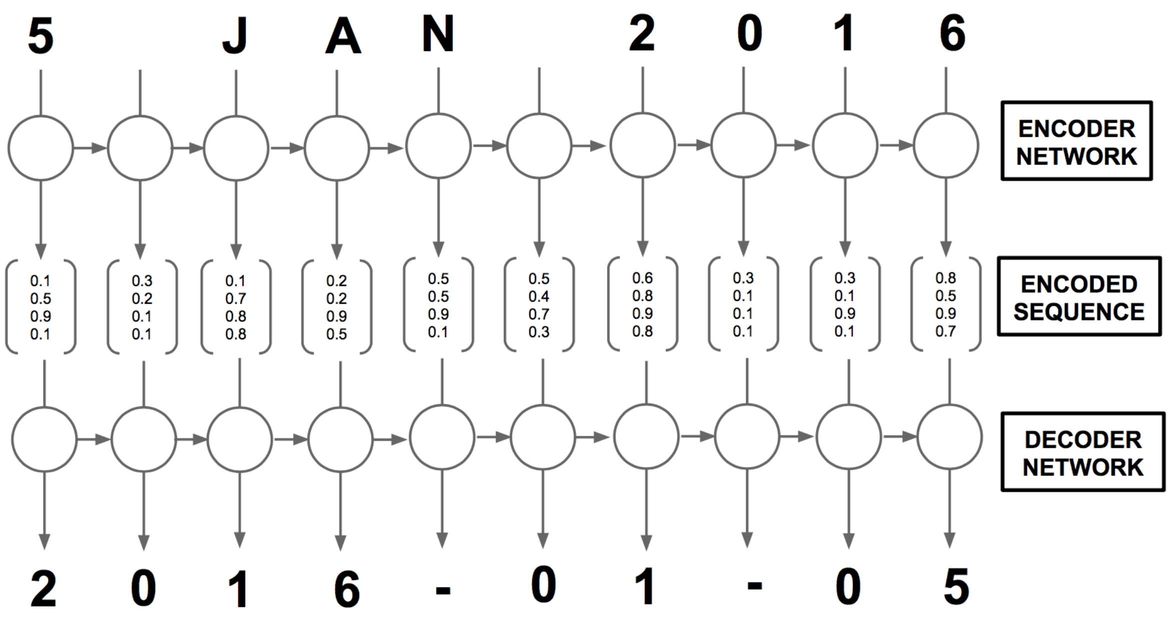


{ Encoder + (attention-mechanism) + Decoder }
Attention-based Approach: Process all input elements SIMULTANEOUSLY

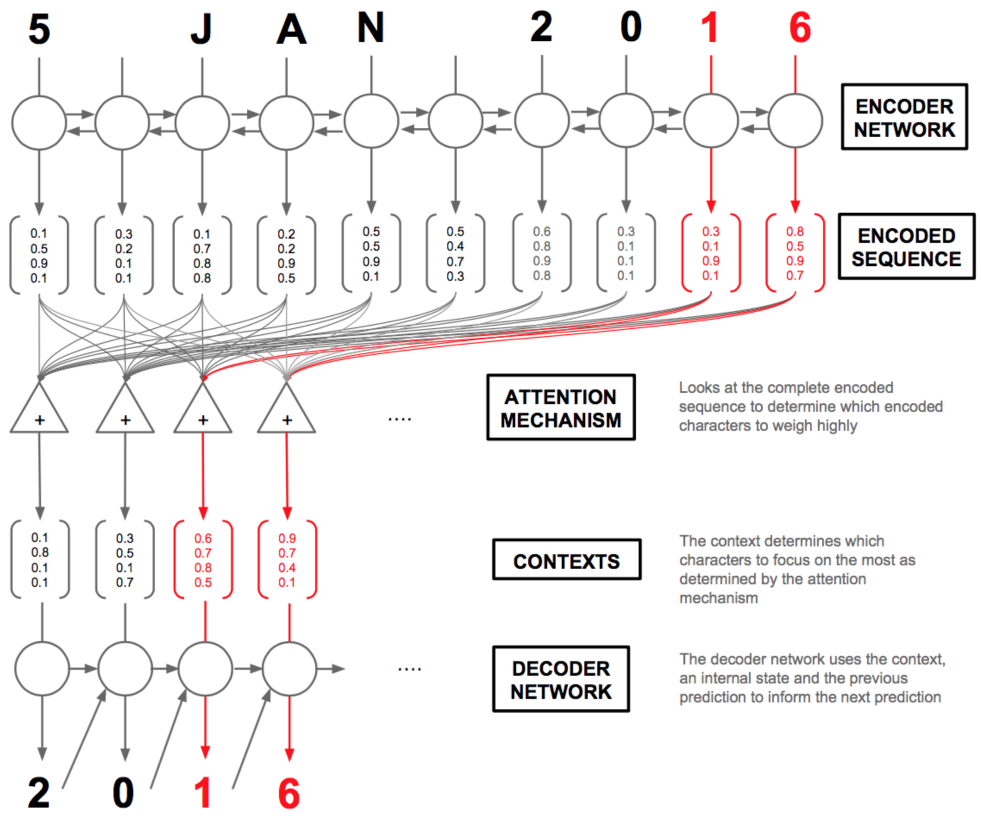
Source: https://smerity.com/articles/2016/google_nmt_arch.html

Evolution of Seq2Seq Model in NMT

Typical Seq2Seq Model



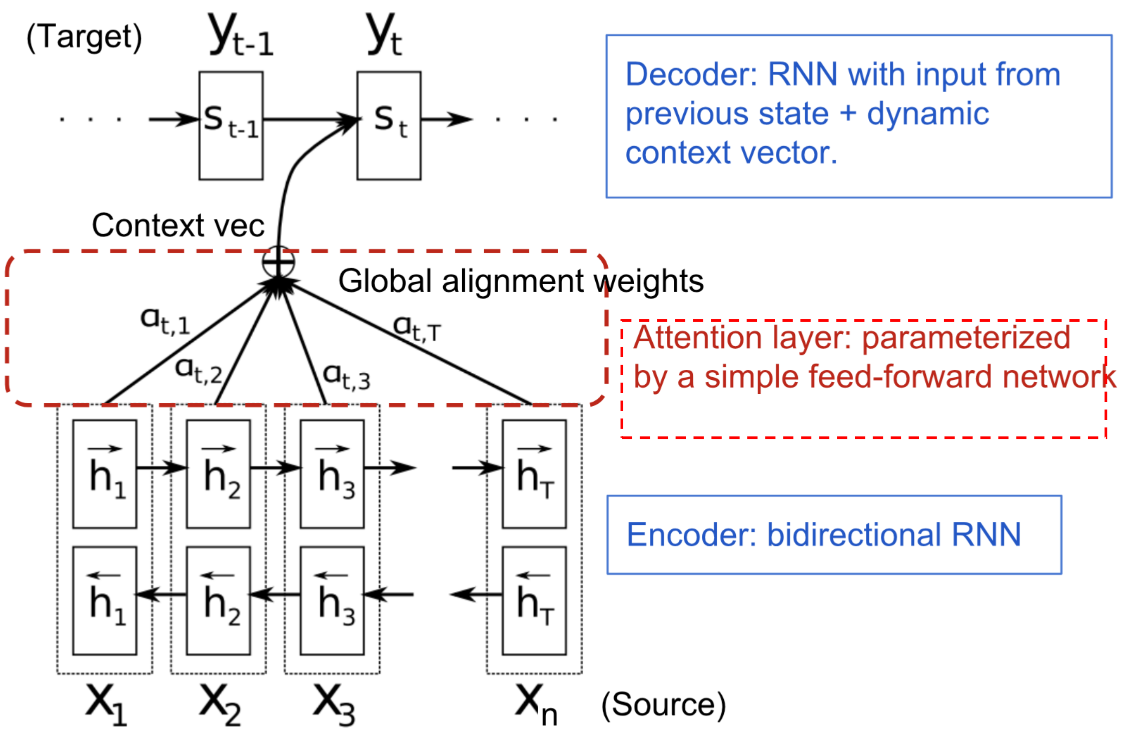
Seq2Seq Model with Attention



Source: https://medium.com/datalogue/attention-in-keras-1892773a4f22

Example of an Additive Attention

The encoder-decoder model with additive attention mechanism



Additive Attention

- **Attention**
- Attention in RNNsearch
  - Query(Q): S_{t-1}
  - Keys(K): $[h_1, h_2, \ldots, h_T]$
  - Values(V): $[h_1, h_2, \ldots, h_T]$
  - Attention(Q, K, V) = $\boxed{\text{softmax}(v^* \tanh(Q+K))}$ * V *(additive attention)*

Weight/ alignment score

Value

Query = Dynamic Context Vector that needs to be translated
Key = Encoder Hidden State
Value = Encoder Hidden State

More about how attention works in Seq2Seq Models
https://machinelearningmastery.com/how-does-attention-work-in-encoder-decoder-recurrent-neural-networks/

Source: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

How is the Attention employed in Transformer?

Self Attention Technique used in Transformer

Name	Alignment score function
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.

Name	Definition
Self-Attention(&)	Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.
Global/Soft	Attending to the entire input state space.
Local/Hard	Attending to the part of input state space; i.e. a patch of the input image.

More about Self Attention in Text: https://medium.com/@_init_/how-self-attention-with-relative-position-representations-works-28173b8c245a

Source: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Self Attention – An Example Pattern

Attention to next word. **Left:** attention weights for all tokens. **Right:** attention weights for selected token ("i")

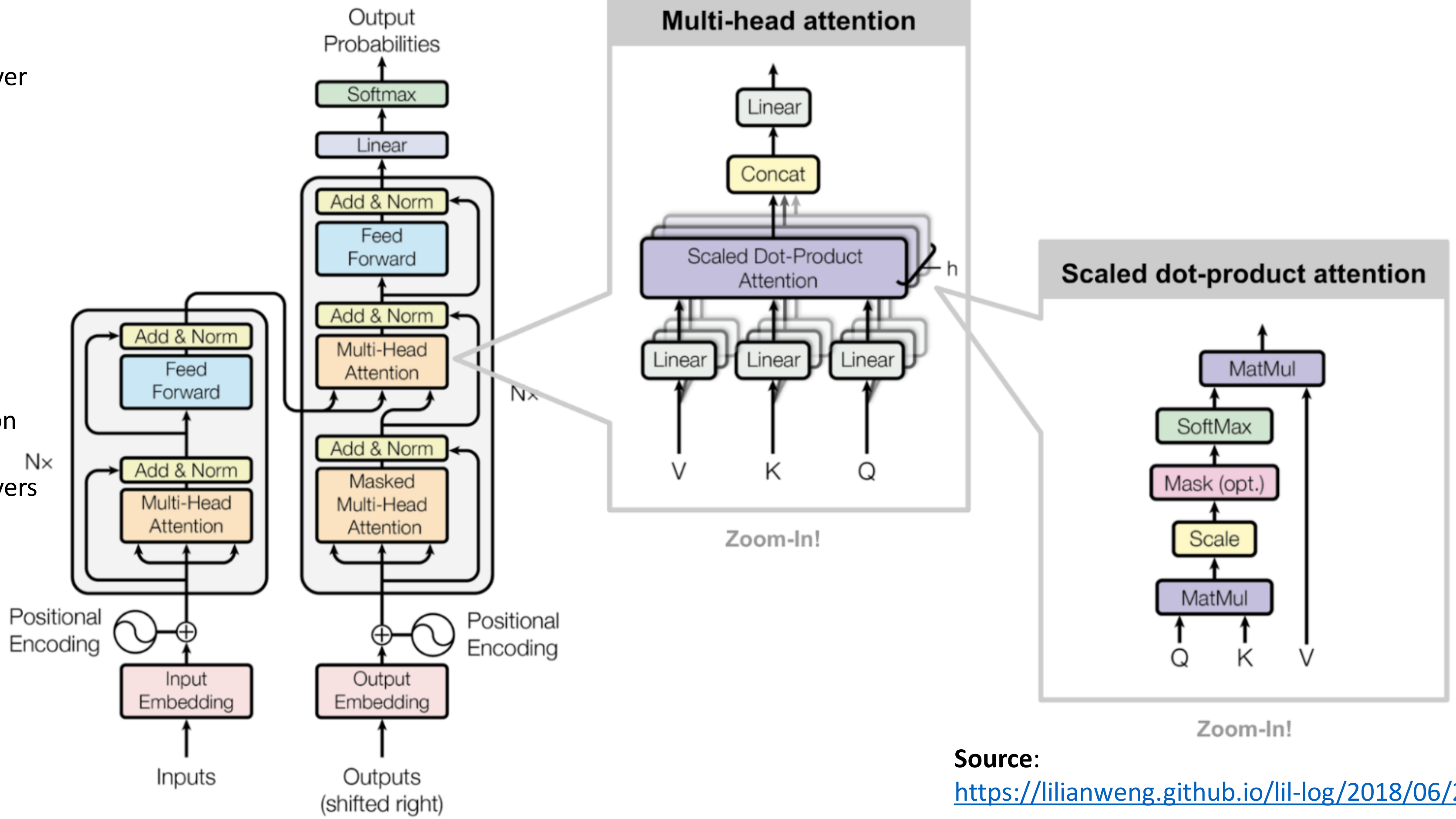
Source:

- https://towardsdatascience.com/deconstructing-bert-distilling-6-patterns-from-100-million-parameters-b49113672f77

Multi-head Attention in a Transformer

Encoder =
Multi-head (self) attention layer
+
Feed Forward layer
+
Normalization layer
+
(residual network)

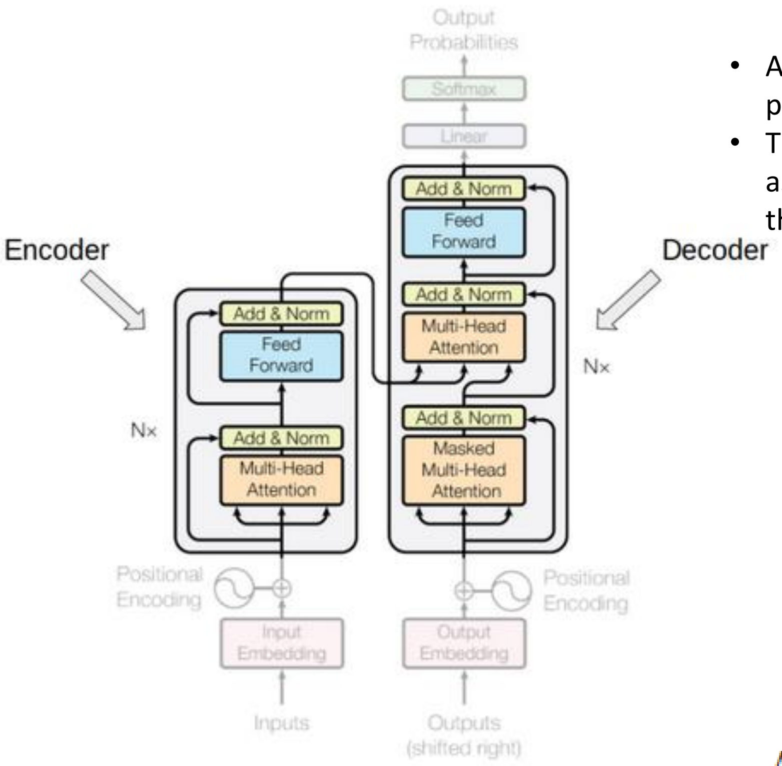
Decoder =
(Masked) Multi-head Attention
+
Multi-head (self) attention layers
+
Feed Forward
+
Normalization layers
+
(residual network)



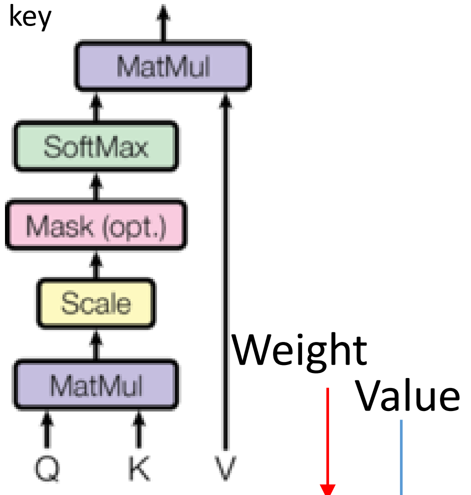
Source: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Multi-head (Self) Attention in a Transformer

- An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors.
- The **output** is computed as a **weighted sum of the values**, where the weight assigned to each value is computed by **a compatibility function** of the query with the corresponding key

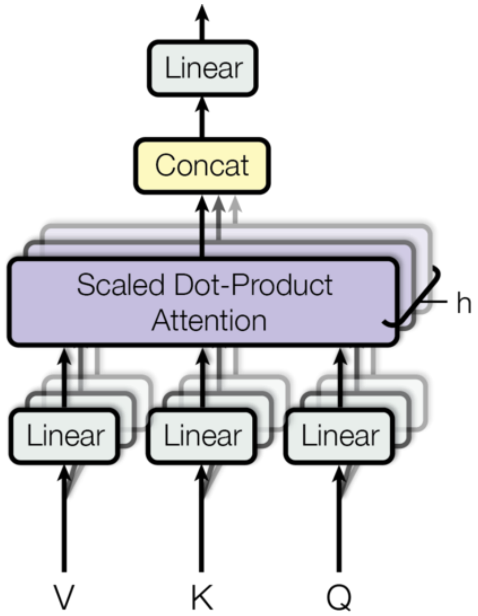


Transformer



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

Scaled Dot-product Attention

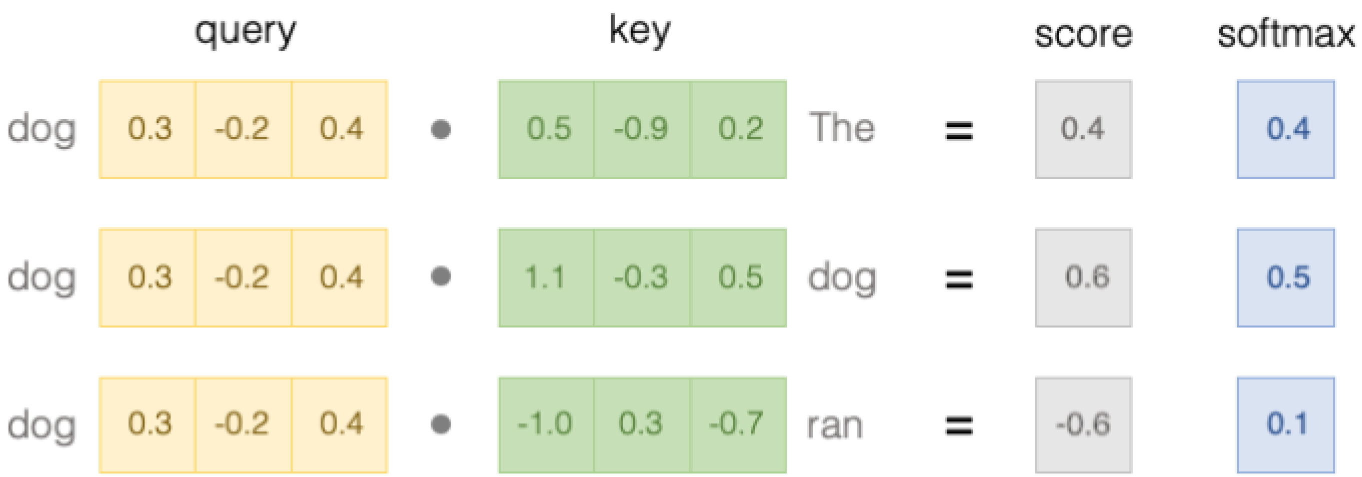


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-head Attention

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

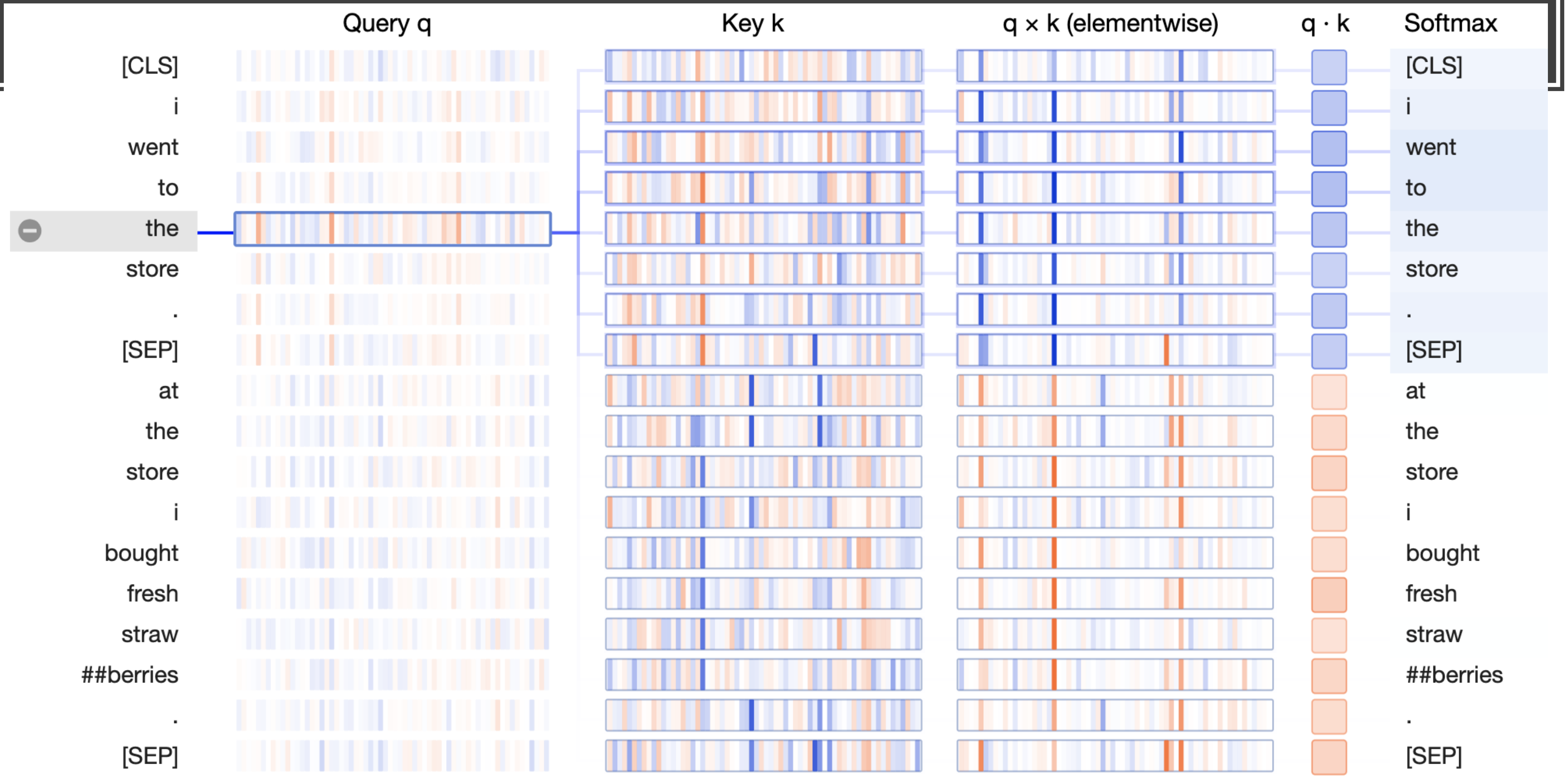
Dot Product Attention



Source:
- https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

Attention – 'Neuron' View

Neuron view of sentence-focused attention pattern for Layer 0, Head 0 of the BERT-base pretrained model

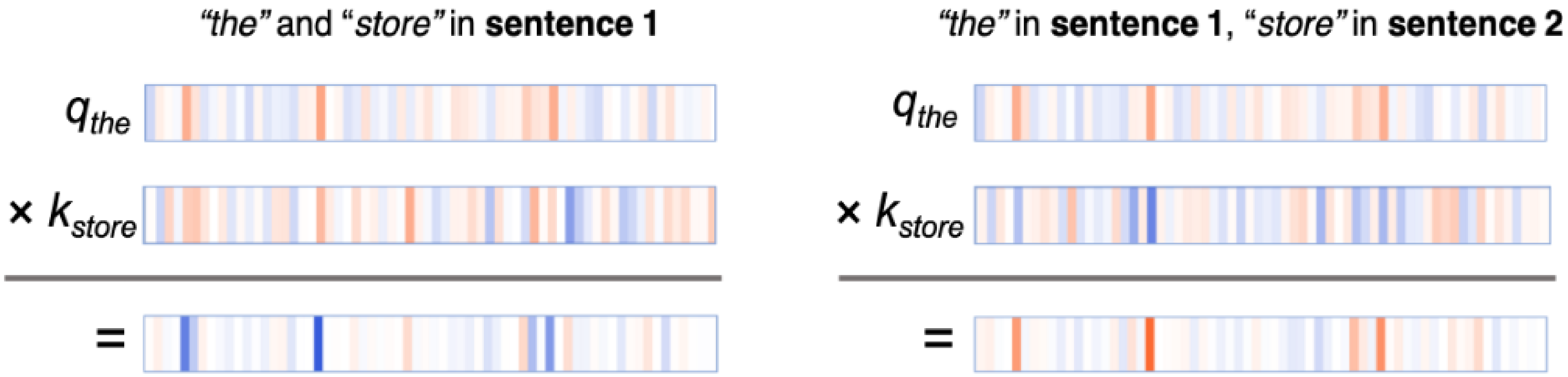


Source:

- https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

Attention Pattern – Layer0, Head0

The query-key product tends to be positive when query and key are in the same sentence (left), and negative when query and key are in different sentences (right)

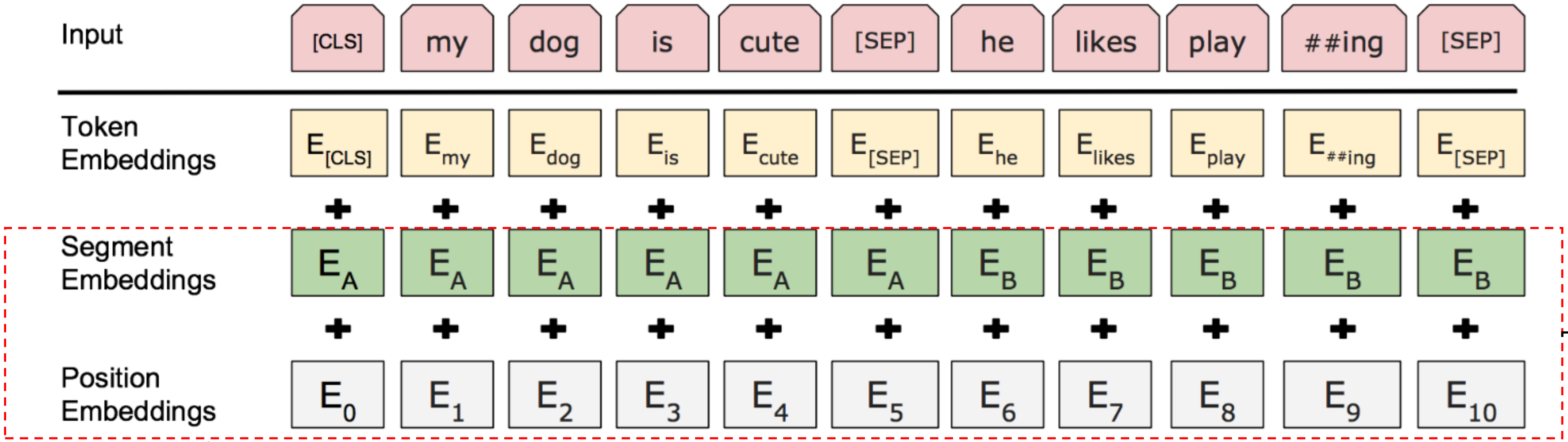


Source:
- https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

What powers QKV vectors - Embedding

When query and key are both from sentence 1, they tend to have values with the same sign along the active neurons, resulting in a positive product. When the query is from sentence 1, and the key is from sentence 2, the same neurons tend to have values with opposite signs, resulting in a negative product.

But how does BERT know the concept of “sentence”, especially in the first layer of the network before higher-level abstractions are formed?
- The information encoded in these sentence embeddings flows to downstream variables, i.e. queries and keys, and enables them to acquire sentence-specific values
- BERT learns a unique position embedding for each of the 512 positions in the input sequence, and this position-specific information can flow through the model to the key and query vectors



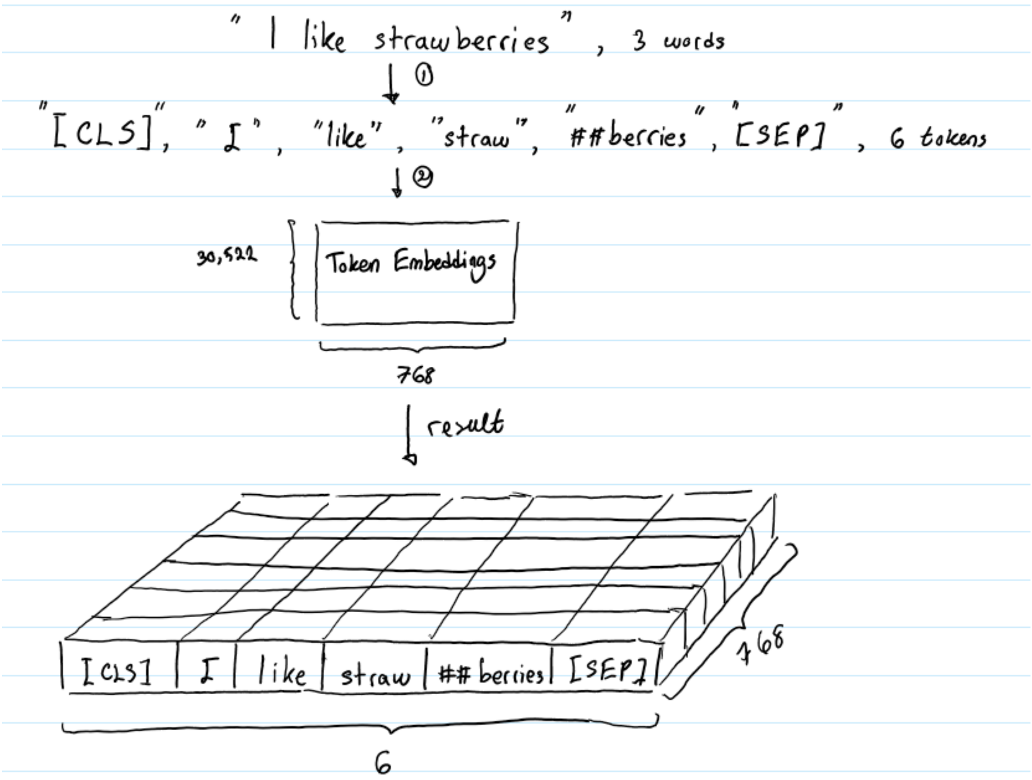
Why?
Ans: In the next few slides

Source:
- https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

How BERT Embedding works?

How BERT Embedding works?

Token Embedding Layer



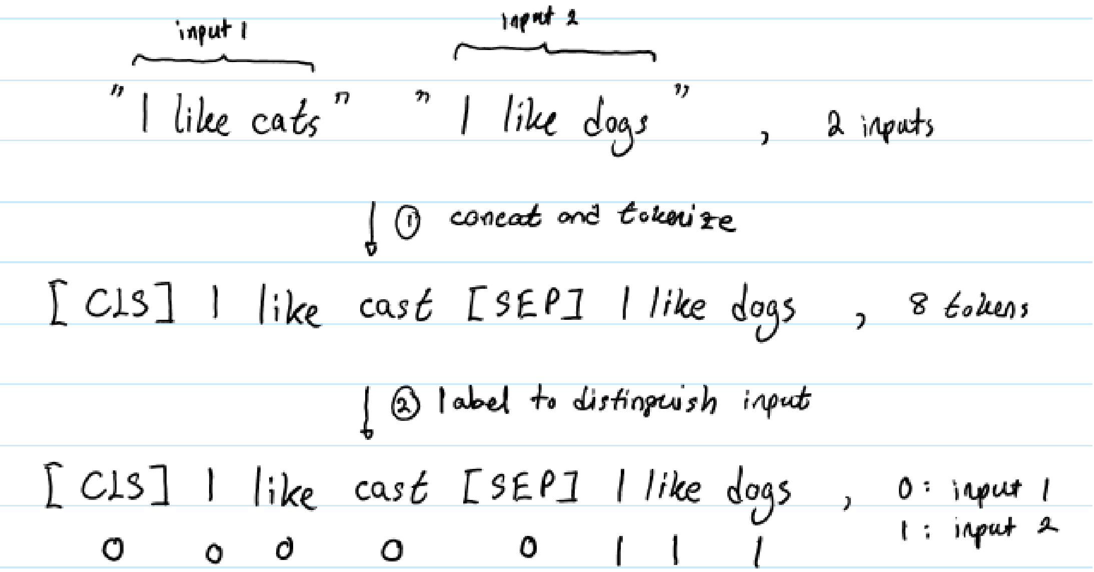
- In $\text{BERT}_{\text{base}}$, each word is represented as a 768-dimensional vector; In $\text{BERT}_{\text{large}}$, each word is represented in 1024d
- The input text is first tokenized before it gets passed to the Token Embeddings layer
- [CLS] – Extra token for Classification Tasks
- [SEP] – Extra token for Sentence pair classification, Q/A pair, etc.,
- Tokenization method – WordPiece Tokenization
- **WordPiece tokenization** enables BERT to only store 30,522 “words” in the vocabulary

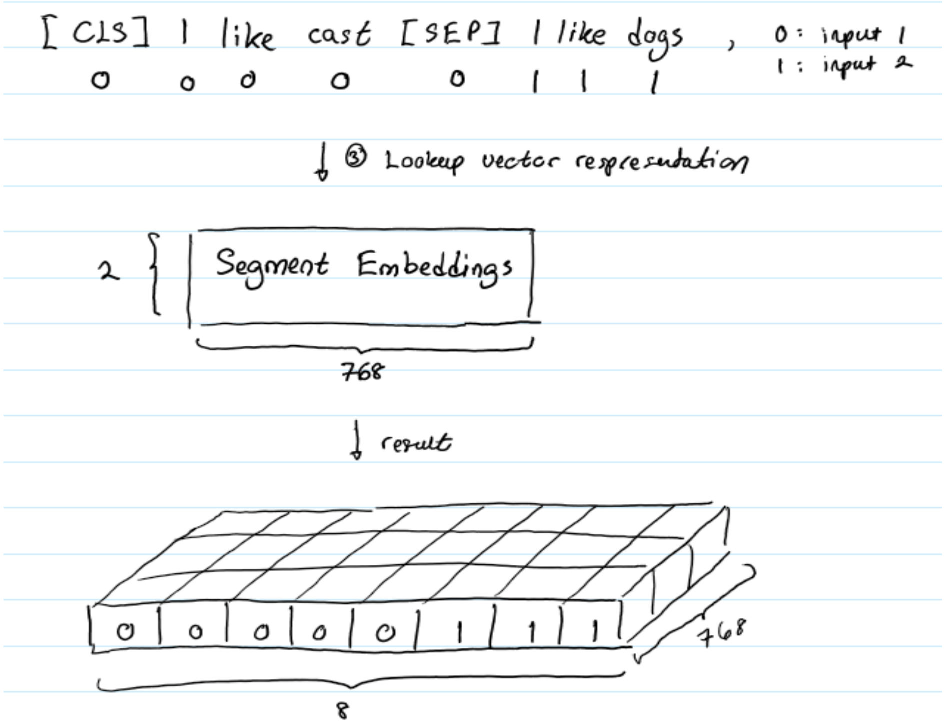
More about how WordPiece Tokenization works
https://medium.com/@makcedward/how-subword-helps-on-your-nlp-model-83dd1b836f46

Source:
- https://medium.com/@_init_/why-bert-has-3-embedding-layers-and-their-implementation-details-9c261108e28a

How BERT Embedding works?

Segment Embedding Layer





The Segment Embeddings layer only has 2 vector representations. The first vector (index 0) is assigned to all tokens that belong to input 1 while the last vector (index 1) is assigned to all tokens that belong to input 2

Source:

- https://medium.com/@_init_/why-bert-has-3-embedding-layers-and-their-implementation-details-9c261108e28a

How BERT Embedding works?

Position Embedding Layer

Transformers do not encode the sequential nature of their inputs.
How Position Embeddings help?

- *I think, therefore I am*
- the first “I” should not have the same vector representation as the second “I”.

BERT was designed to process input sequences of up to length 512
The Position Embeddings layer is a lookup table of size (512, 768).

Inputs like “Hello world” and “Hi there”, both “Hello” and “Hi” will have identical position embeddings. Similarly, both “world” and “there” will have the same position embedding

--

Source:

- https://medium.com/@_init_/why-bert-has-3-embedding-layers-and-their-implementation-details-9c261108e28a

Main Code Sources for BERT

PyTorch

https://github.com/huggingface/transformers/blob/master/transformers/modeling_bert.py

https://github.com/huggingface/transformers/blob/master/transformers/tokenization_bert.py

TensorFlow

https://github.com/google-research/bert/blob/master/modeling.py

https://github.com/google-research/bert/blob/master/tokenization.py

Code Snippets - PyTorch

SelfAttention Class

```
class BertSelfAttention(nn.Module):
    def __init__(self, config):
        super(BertSelfAttention, self).__init__()
        if config.hidden_size % config.num_attention_heads != 0:
            raise ValueError(
                "The hidden size (%d) is not a multiple of the number of attention "
                "heads (%d)" % (config.hidden_size, config.num_attention_heads))
        self.output_attentions = config.output_attentions

        self.num_attention_heads = config.num_attention_heads
        self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
        self.all_head_size = self.num_attention_heads * self.attention_head_size

        self.query = nn.Linear(config.hidden_size, self.all_head_size)
        self.key = nn.Linear(config.hidden_size, self.all_head_size)
        self.value = nn.Linear(config.hidden_size, self.all_head_size)

        self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
```

Gaussian Error Linear unit Activation

```
def gelu(x):
    """ Original Implementation of the gelu activation function in Google Bert repo when initially created.
        For information: OpenAI GPT's gelu is slightly different (and gives slightly different results):
        0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
        Also see https://arxiv.org/abs/1606.08415
    """
    return x * 0.5 * (1.0 + torch.erf(x / math.sqrt(2.0)))

def gelu_new(x):
    """ Implementation of the gelu activation function currently in Google Bert repo (identical to OpenAI GPT).
        Also see https://arxiv.org/abs/1606.08415
    """
    return 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
```

More about GELU

- https://arxiv.org/pdf/1606.08415.pdf
- https://datascience.stackexchange.com/questions/49522/what-is-gelu-activation

Code Snippets - TF

```
class BertConfig(object):
  """Configuration for `BertModel`."""

  def __init__(self,
               vocab_size,
               hidden_size=768,
               num_hidden_layers=12,
               num_attention_heads=12,
               intermediate_size=3072,
               hidden_act="gelu",
               hidden_dropout_prob=0.1,
               attention_probs_dropout_prob=0.1
               max_position_embeddings=512,
               type_vocab_size=16,
               initializer_range=0.02):
```

```
"""Constructs BertConfig.

Args:
  vocab_size: Vocabulary size of `inputs_ids` in `BertModel`.
  hidden_size: Size of the encoder layers and the pooler layer.
  num_hidden_layers: Number of hidden layers in the Transformer encoder.
  num_attention_heads: Number of attention heads for each attention layer in
    the Transformer encoder.
  intermediate_size: The size of the "intermediate" (i.e., feed-forward)
    layer in the Transformer encoder.
  hidden_act: The non-linear activation function (function or string) in the
    encoder and pooler.
  hidden_dropout_prob: The dropout probability for all fully connected
    layers in the embeddings, encoder, and pooler.
  attention_probs_dropout_prob: The dropout ratio for the attention
    probabilities.
  max_position_embeddings: The maximum sequence length that this model might
    ever be used with. Typically set this to something large just in case
    (e.g., 512 or 1024 or 2048).
  type_vocab_size: The vocabulary size of the `token_type_ids` passed into
    `BertModel`.
  initializer_range: The stdev of the truncated_normal_initializer for
    initializing all weight matrices.
"""
```

```
class WordpieceTokenizer(object):
  """Runs WordPiece tokenziation."""

  def __init__(self, vocab, unk_token="[UNK]", max_input_chars_per_word=200):
    self.vocab = vocab
    self.unk_token = unk_token
    self.max_input_chars_per_word = max_input_chars_per_word

  def tokenize(self, text):
    """Tokenizes a piece of text into its word pieces.

    This uses a greedy longest-match-first algorithm to perform tokenization
    using the given vocabulary.

    For example:
      input = "unaffable"
      output = ["un", "##aff", "##able"]

    Args:
      text: A single token or whitespace separated tokens. This should have
        already been passed through `BasicTokenizer.

    Returns:
      A list of wordpiece tokens.
    """
```

Code Snippets - TF

```
def transformer_model(input_tensor,
                      attention_mask=None,
                      hidden_size=768,
                      num_hidden_layers=12,
                      num_attention_heads=12,
                      intermediate_size=3072,
                      intermediate_act_fn=gelu,
                      hidden_dropout_prob=0.1,
                      attention_probs_dropout_prob=0.1,
                      initializer_range=0.02,
                      do_return_all_layers=False):
```

```
"""Multi-headed, multi-layer Transformer from "Attention is All You Need".

This is almost an exact implementation of the original Transformer encoder.

See the original paper:
https://arxiv.org/abs/1706.03762

Also see:
https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py

Args:
  input_tensor: float Tensor of shape [batch_size, seq_length, hidden_size].
  attention_mask: (optional) int32 Tensor of shape [batch_size, seq_length,
    seq_length], with 1 for positions that can be attended to and 0 in
    positions that should not be.
  hidden_size: int. Hidden size of the Transformer.
  num_hidden_layers: int. Number of layers (blocks) in the Transformer.
  num_attention_heads: int. Number of attention heads in the Transformer.
  intermediate_size: int. The size of the "intermediate" (a.k.a., feed
    forward) layer.
  intermediate_act_fn: function. The non-linear activation function to apply
    to the output of the intermediate/feed-forward layer.
  hidden_dropout_prob: float. Dropout probability for the hidden layers.
  attention_probs_dropout_prob: float. Dropout probability of the attention
    probabilities.
  initializer_range: float. Range of the initializer (stddev of truncated
    normal).
  do_return_all_layers: Whether to also return all layers or just the final
    layer.
```

Beyond BERT

	BERT	RoBERTa	DistilBERT	XLNet
Size (millions)	**Base**: 110 **Large**: 340	**Base**: 110 **Large**: 340	**Base:** 66	**Base**: ~110 **Large**: ~340
Training Time	**Base**: 8 x V100 x 12 days* **Large:** 64 TPU Chips x 4 days (or 280 x V100 x 1 days*)	**Large:** 1024 x V100 x 1 day; 4-5 times more than BERT.	**Base**: 8 x V100 x 3.5 days; 4 times less than BERT.	**Large:** 512 TPU Chips x 2.5 days; 5 times more than BERT.
Performance	Outperforms state-of-the-art in Oct 2018	2-20% improvement over BERT	3% degradation from BERT	2-15% improvement over BERT
Data	16 GB BERT data (Books Corpus + Wikipedia). 3.3 Billion words.	160 GB (16 GB BERT data + 144 GB additional)	16 GB BERT data. 3.3 Billion words.	**Base**: 16 GB BERT data **Large**: 113 GB (16 GB BERT data + 97 GB additional). 33 Billion words.
Method	BERT (Bidirectional Transformer with MLM and NSP)	BERT without NSP**	BERT Distillation	Bidirectional Transformer with Permutation based modeling

Source:

- https://towardsdatascience.com/bert-roberta-distilbert-xlnet-which-one-to-use-3d5ab82ba5f8

Beyond BERT – A Lite BERT

Motivation: Larger models are not always good for NLP

Model	Hidden Size	Parameters	RACE (Accuracy)
BERT-large (Devlin et al., 2019)	1024	334M	72.0%
BERT-large (ours)	1024	334M	73.9%
BERT-xlarge (ours)	2048	1270M	54.3%

Table 1: Increasing hidden size of BERT-large leads to worse performance on RACE.

Models	SQuAD1.1 dev	SQuAD2.0 dev	SQuAD2.0 test	RACE test (Middle/High)
Single model (from leaderboard as of Sept. 23, 2019)				
BERT-large	90.9/84.1	81.8/79.0	89.1/86.3	72.0 (79.6/70.1)
XLNet	94.5/89.0	88.8/86.1	89.1/86.3	81.8 (85.5/80.2)
RoBERTa	94.6/88.9	89.4/86.5	89.8/86.8	83.2 (86.5/81.3)
UPM	-	-	89.9/87.2	-
XLNet + SG-Net Verifier++	-	-	90.1/87.2	-
ALBERT (1M)	94.8/89.2	89.9/87.2	-	86.0 (88.2/85.1)
ALBERT (1.5M)	**94.8/89.3**	**90.2/87.4**	**90.9/88.1**	**86.5 (89.0/85.5)**
Ensembles (from leaderboard as of Sept. 23, 2019)				
BERT-large	92.2/86.2	-	-	-
XLNet + SG-Net Verifier	-	-	90.7/88.2	-
UPM	-	-	90.7/88.2	
XLNet + DAAF + Verifier	-	-	90.9/88.6	-
DCMI+	-	-	-	84.1 (88.5/82.3)
ALBERT	**95.5/90.1**	**91.4/88.9**	**92.2/89.7**	**89.4 (91.2/88.6)**

Table 14: State-of-the-art results on the SQuAD and RACE benchmarks.

Source:
- https://medium.com/@lessw/meet-albert-a-new-lite-bert-from-google-toyota-with-state-of-the-art-nlp-performance-and-18x-df8f7b58fa28
- https://medium.com/syncedreview/googles-albert-is-a-leaner-bert-achieves-sota-on-3-nlp-benchmarks-f64466dd583

Appendix

BERT FAQs

https://yashuseth.blog/2019/06/12/bert-explained-faqs-understand-bert-working/

Thank You