

CPSC 532P / LING 530A: Deep Learning for Natural Language Processing (DL-NLP)

Muhammad Abdul-Mageed

`muhammad.mageed@ubc.ca`

Natural Language Processing Lab

The University of British Columbia

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

Background

- **Goal:** to reduce sequential computation
- **Extended Neural GPU, ByteNet, and ConvS2Sb:** use convnets for computing hidden representations in parallel for all input and output positions.
- Still **difficult to learn dependencies between distant positions**
- **Transformer:** **Constant number of operations** to relate signals from two arbitrary input or output positions
- This is at the **cost of reduced effective resolution** due to **averaging attention-weighted positions**
- Counteracted with **Multi-Head Attention**

Self-Attention (aka intra-attention)

What is self-attention?

- An attention mechanism that **relates different positions of a single sequence** in order to compute a representation of the sequence
- Has been successfully applied to **reading comprehension, abstractive summarization, textual entailment, and learning task-independent sentence representations**

Self-Attention

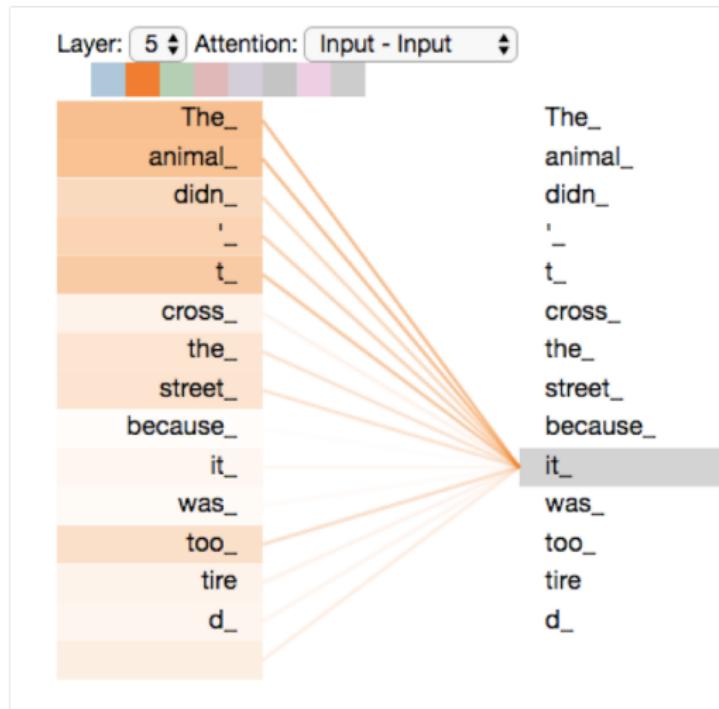


Figure: Self-attention. (Source: <http://jalammar.github.io>).

Transformer Architecture

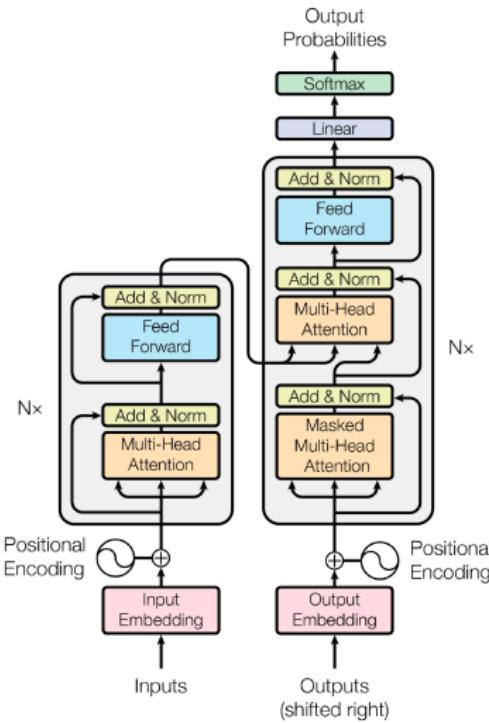
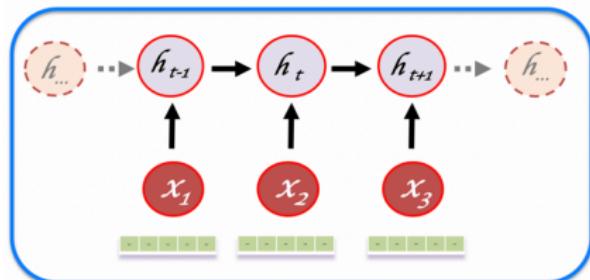
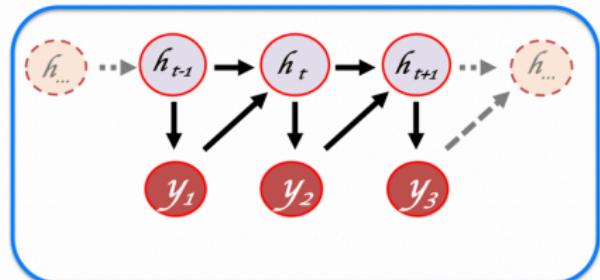


Figure: Transformer: Model Architecture.

Recall: Basic Encoder Decoder Idea



encoder



decoder

Figure: Transformer has an encoder and a decoder.

Transformer is based on encoder-decoder design

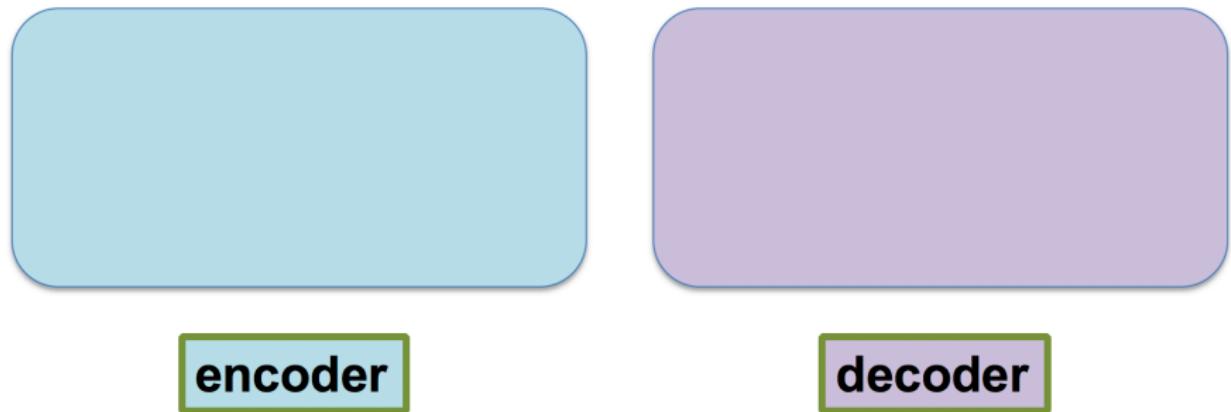


Figure: Transformer has an encoder and a decoder, **each with specialized building blocks.**

Basic Encoder-Decoder Operations

- **Encoder** maps an input sequence of symbol representations (x_1, \dots, x_n) to a sequence of continuous representations $\mathbf{z} = (z_1, \dots, z_n)$.
- Given \mathbf{z} , the **decoder** then generates an output sequence (y_1, \dots, y_m) of symbols one element at a time.
- **Auto-regressive:** At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

Input and Output Embeddings

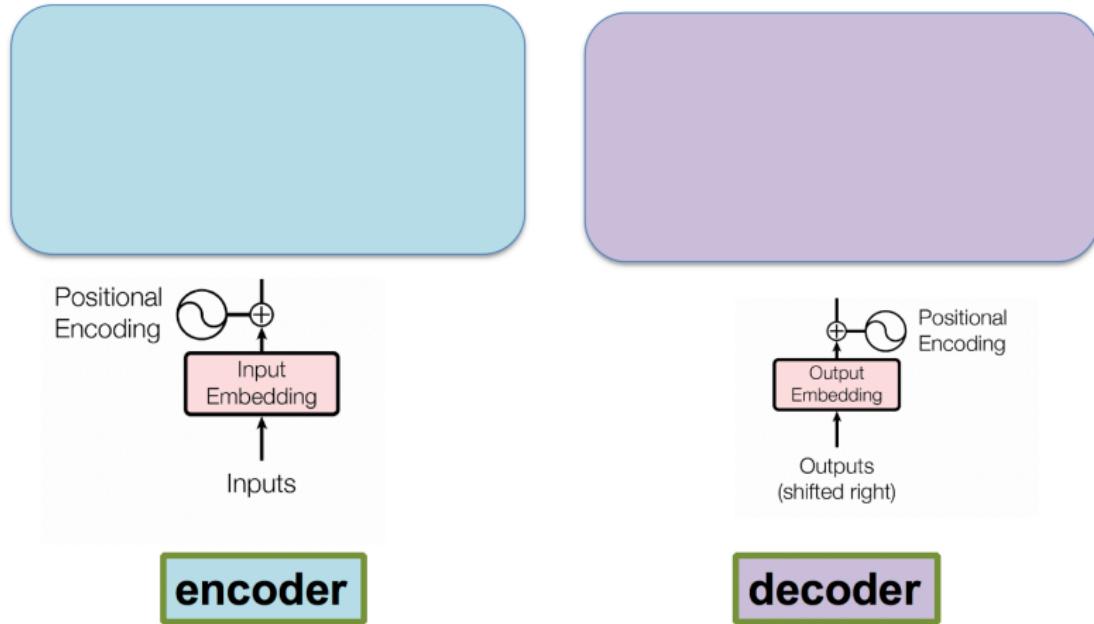
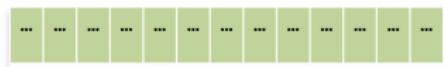


Figure: Embeddings for Encoder and Decoder.

Encoding Word Position



word embedding vector

+



sinusoidal vector



Figure: Simple Addition for encoding word positions

Positional Encoding

Recipe for PE

- Positional encodings added to the input embeddings at the bottoms of encoder and decoder stacks
- PEs *have the same dimension* d_{model} as the embeddings, so the two can be summed:
 $PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$ &
 $PE(pos, 2i + 1) = \cos(pos/10000^{2i/d_{model}})$
where pos is the position and i is the dimension.

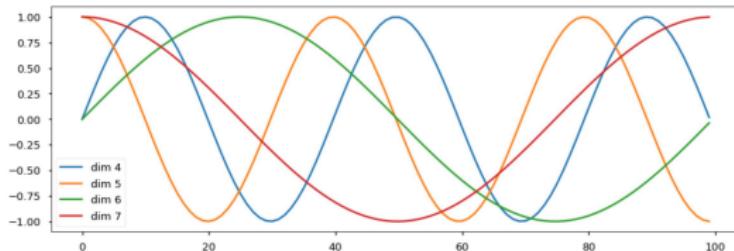


Figure: <https://nlp.seas.harvard.edu/2018/04/03/attention.html>.

Positional Encoding Example

PE Example

- For w at position $pos \in [0, L - 1]$, in sequence $s = (w_1, \dots, w_4)$, with 4-dimensional embedding e_w and $d_{model} = 4$:

$$\begin{aligned} e'_w &= e_w + \left[\sin\left(\frac{pos}{10000^0}\right), \cos\left(\frac{pos}{10000^0}\right), \sin\left(\frac{pos}{10000^{2/4}}\right), \cos\left(\frac{pos}{10000^{2/4}}\right) \right] \\ &= e_w + \left[\sin(pos), \cos(pos), \sin\left(\frac{pos}{100}\right), \cos\left(\frac{pos}{100}\right) \right] \end{aligned}$$

where the formula for positional encoding is as follows

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right),$$

$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right).$$

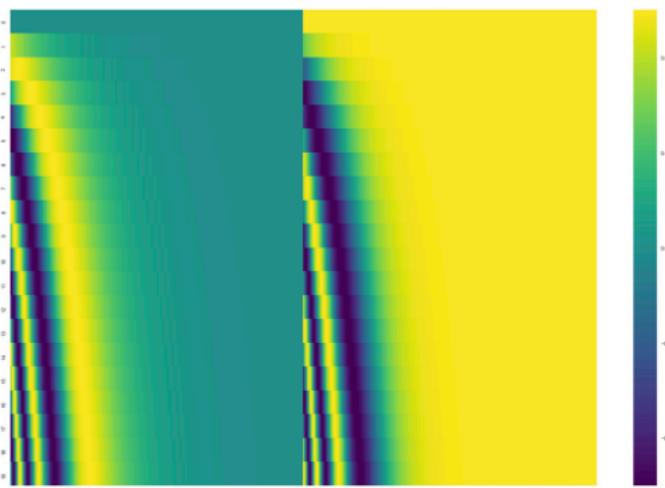
with $d_{model} = 512$ (thus $i \in [0, 255]$) in the original paper.

Figure: Source: <https://datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model>.

//datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model.

Positional Encoding Illustrated

In the following figure, each row corresponds to the positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

Figure: Source: <http://jalammar.github.io>. This illustration is unfortunately **incorrect** as it uses sin for the first half of the embedding and cos for the second half, instead of sin for even indices and cos for odd indices.

Torch Tensor for Positional Encodings

```
for pos in range(max_seq_len):
    for i in range(0, d_model, 2):
        pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
        pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/d_model)))
print(pe)

tensor([[ 0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000,
         0.0000,  1.0000],
       [ 0.8415,  0.9875,  0.0251,  1.0000,  0.0006,  1.0000,  0.0000,  1.0000,
         0.0000,  1.0000],
       [ 0.9093,  0.9502,  0.0502,  1.0000,  0.0013,  1.0000,  0.0000,  1.0000,
         0.0000,  1.0000],
       [ 0.1411,  0.8891,  0.0753,  0.9999,  0.0019,  1.0000,  0.0000,  1.0000,
         0.0000,  1.0000],
       [-0.7568,  0.8057,  0.1003,  0.9999,  0.0025,  1.0000,  0.0001,  1.0000,
         0.0000,  1.0000],
       [-0.9589,  0.7021,  0.1253,  0.9998,  0.0032,  1.0000,  0.0001,  1.0000,
         0.0000,  1.0000],
       [-0.2794,  0.5809,  0.1501,  0.9997,  0.0038,  1.0000,  0.0001,  1.0000,
         0.0000,  1.0000],
       [ 0.6570,  0.4452,  0.1749,  0.9996,  0.0044,  1.0000,  0.0001,  1.0000,
         0.0000,  1.0000],
       [ 0.9894,  0.2983,  0.1996,  0.9995,  0.0050,  1.0000,  0.0001,  1.0000,
         0.0000,  1.0000],
       [ 0.4121,  0.1439,  0.2241,  0.9994,  0.0057,  1.0000,  0.0001,  1.0000,
         0.0000,  1.0000]])
```

Figure: Tensor of 10×10 . (max_seq_len and d_model both set to 10).

Creating New Encodings

```
print("\n-- Positional encoding for w in position 3:\n\n", pe3)
print("\n-- Embedding vector for w in position 3:\n\n", word_embedding)

-- Positional encoding for w in position 3:
tensor([1.4112e-01, 8.8908e-01, 7.5285e-02, 9.9993e-01, 1.8929e-03, 1.0000e+00,
        4.7547e-05, 1.0000e+00, 1.1943e-06, 1.0000e+00])

-- Embedding vector for w in position 3:
[0.15127356 0.11721583 0.8375494  0.2517023  0.52421439 0.63624648
 0.52809084 0.21888487 0.93466182 0.419567   ]
```

```
# We simply add the two vectors
print("\n-- Embedding vector and positional encoding added:\n\n", word_embedding + pe3)
```

```
--Embedding vector and positional encoding added:
tensor([0.2924, 1.0063, 0.9128, 1.2516, 0.5261, 1.6362, 0.5281, 1.2189, 0.9347,
        1.4196], dtype=torch.float64)
```

Figure: New encoding is simply created by adding up embedding vector and sinusoidal vector.

Encoder Blocks (6 Identical Blocks)

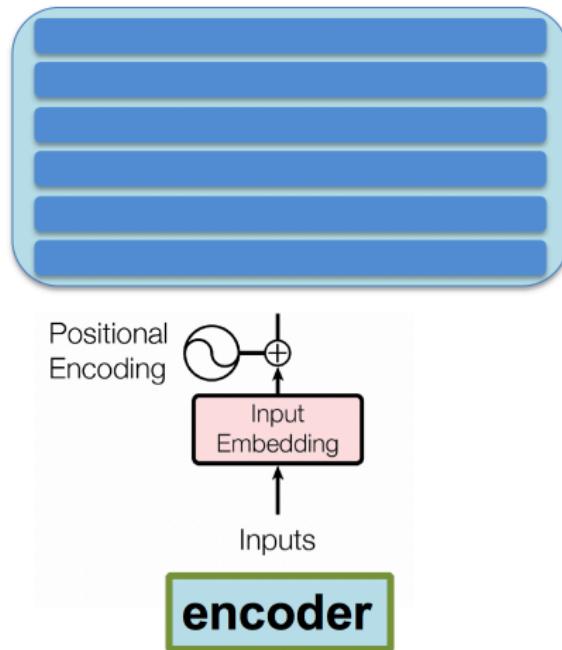


Figure: Encoder's 6 identical blocks.

An Encoder Block

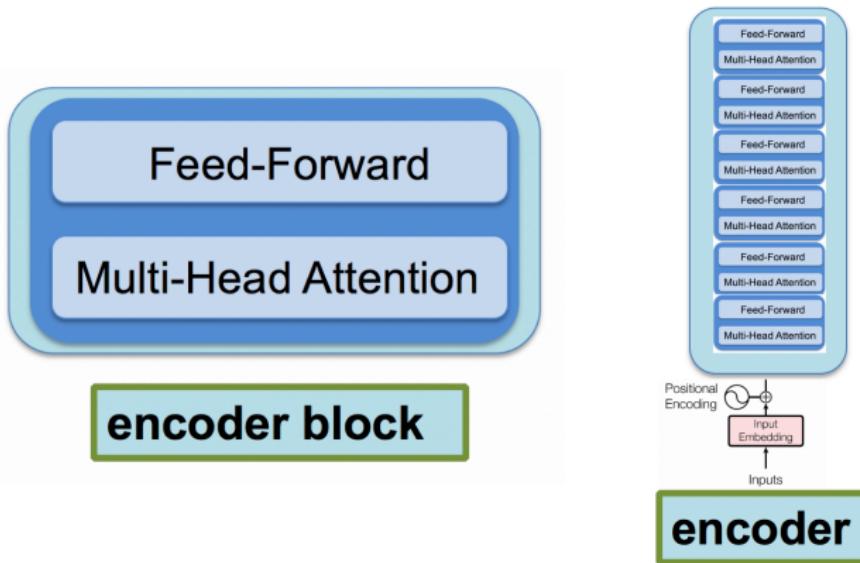


Figure: Each encoder block has two sub-layers, a multi-head attention and a feed-forward.

Self-Attention

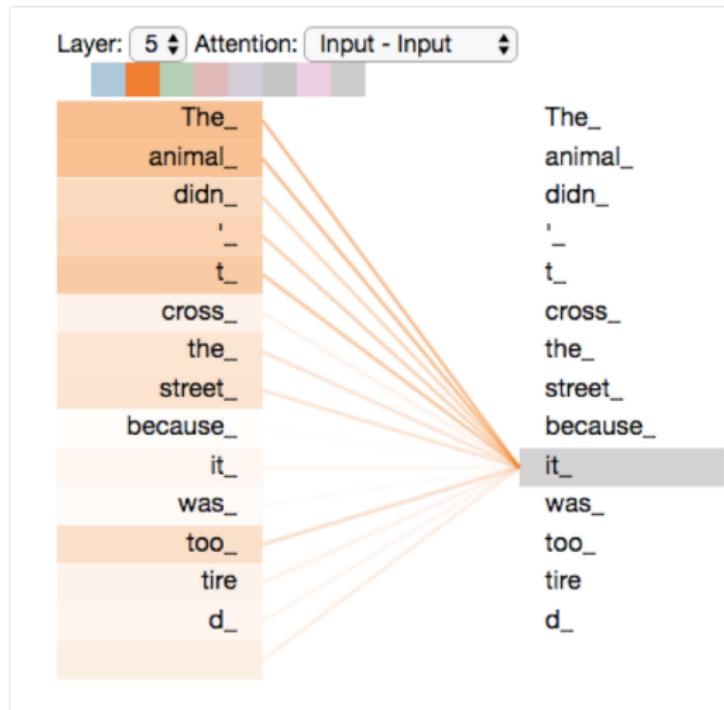
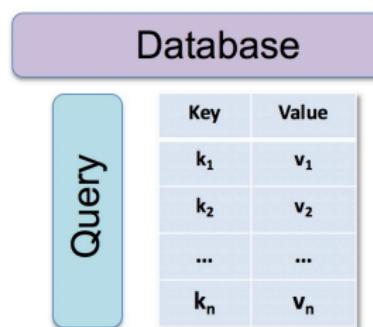


Figure: Self-attention. (Source: <http://jalammar.github.io>).

Query, Key, and Value

Intuition

- Retrieving from a database, we issue a **query (q)** to identify a **key (k)** that has a similarity, based on a **value (v)**, to q.
- To do **backpropagation** on the output, we will not just want one similarity value, but a **similarity distribution** over all keys.



$$\text{Attention}(q, \mathbf{k}, \mathbf{v}) = \sum_i \text{sim}(q, k_i) v_i$$

MT Example of Query, Key, and Value

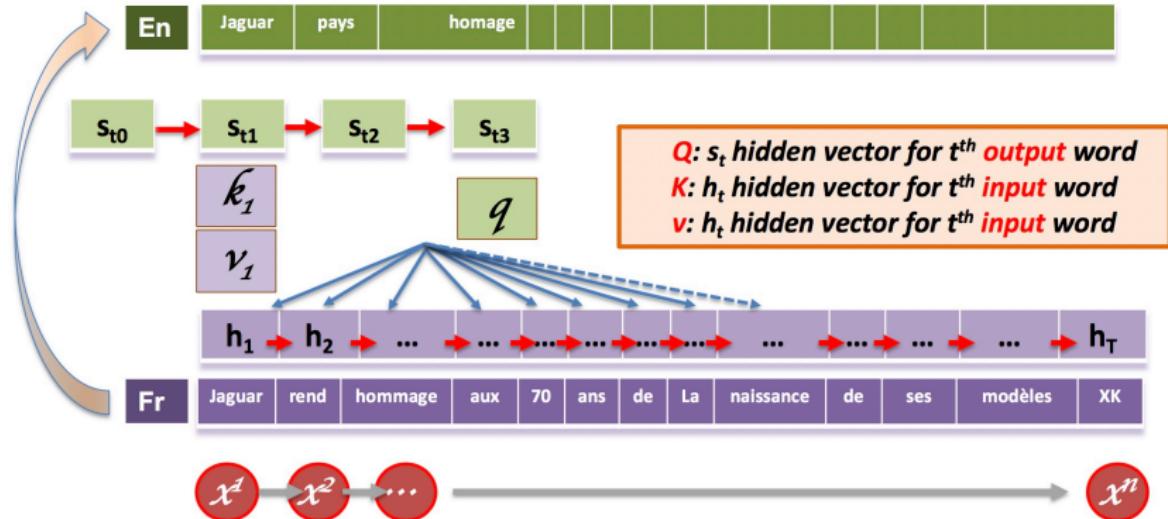


Figure: q, k, and V in MT.

Self-Attention in Transformer with q, k, v

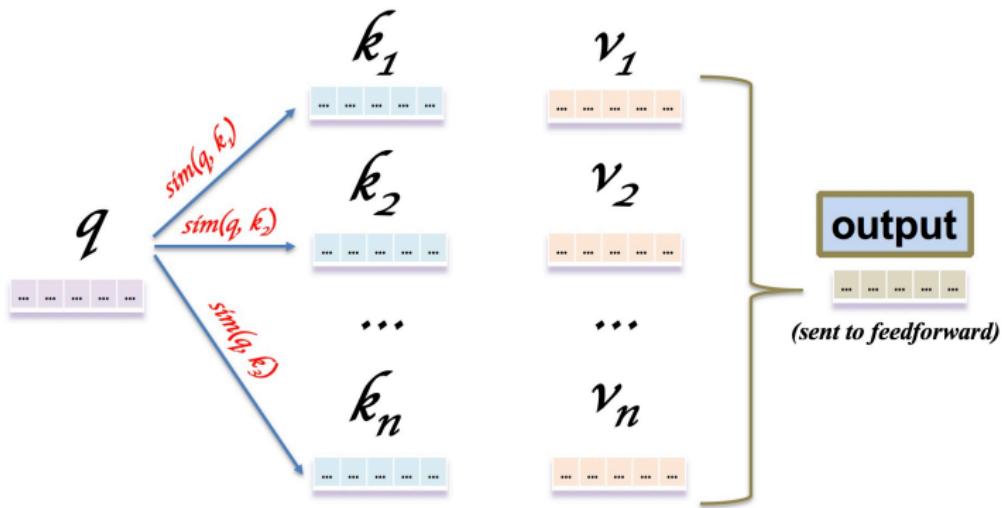


Figure: Mapping a **query** and a set of **key-value** pairs to an output. **Similarity** **s** will be based on **dot product**. Well, **scaled** dot product...

Z weighted sum of $\sum_i \text{softmax}(\text{sim}(q, k_i)) v_i$

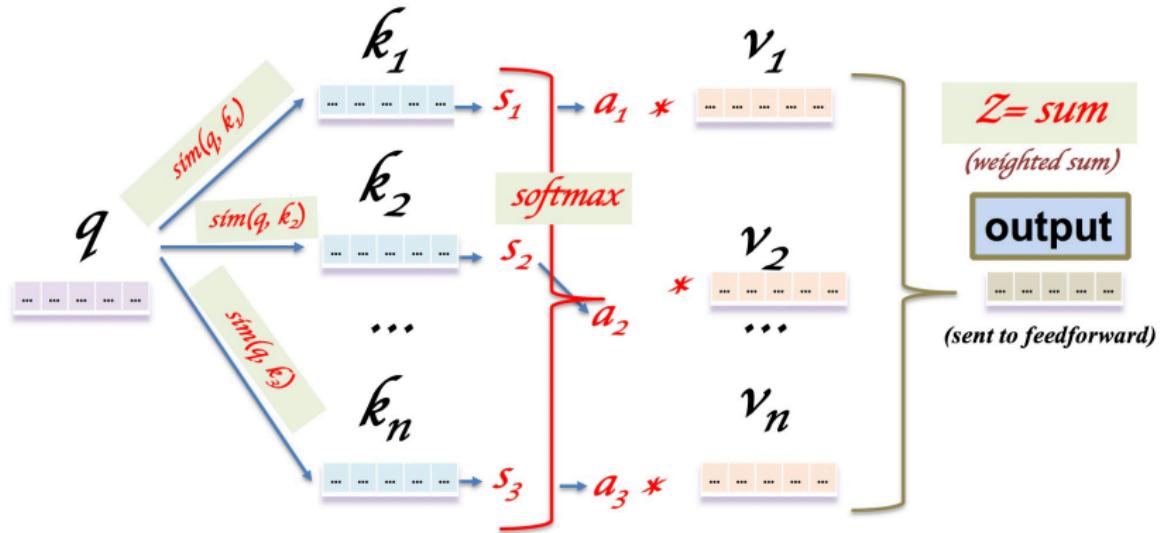


Figure: $s = \text{sim}(q, k_i)$. $\alpha = \text{softmax}(s)$. $Z = \sum_i \alpha_i v_i$

Self-Attention Illustrated: Query “the”

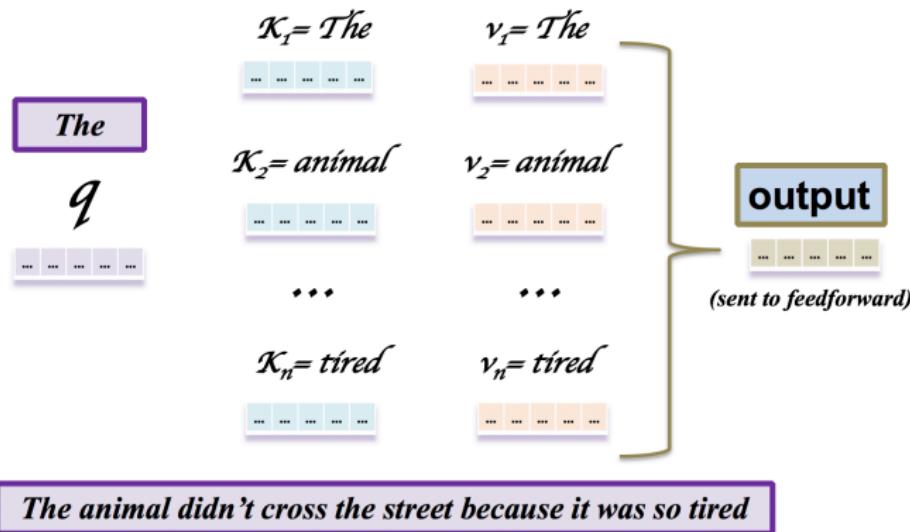


Figure: Mapping the **query** “the” and a set of **key-value** pairs to an output.

Self-Attention Illustrated: Query “animal”

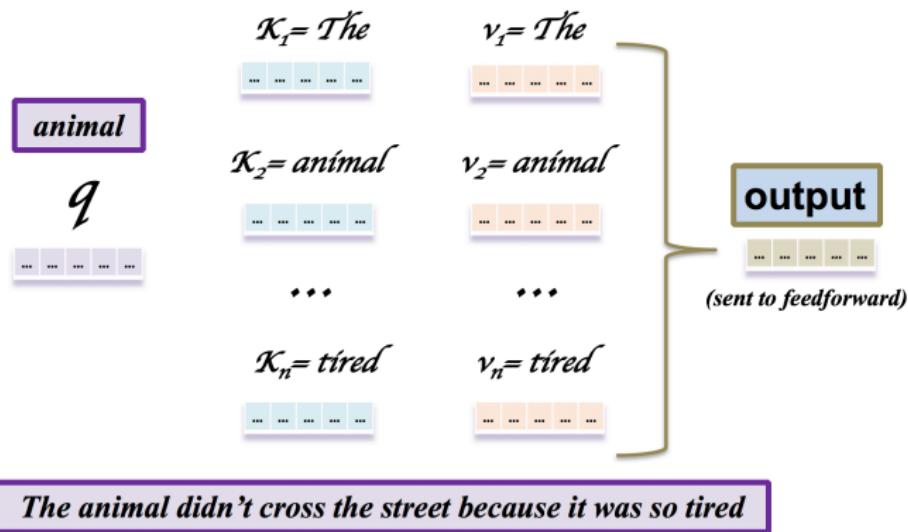
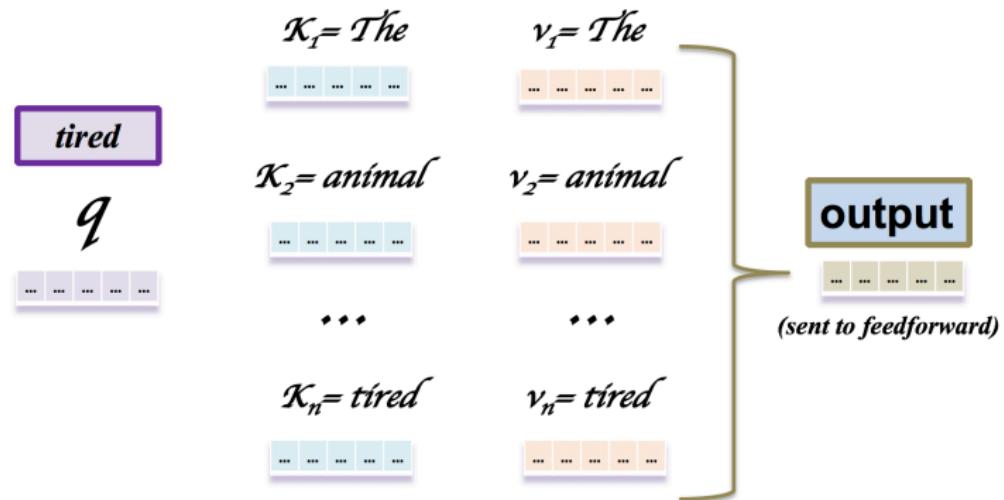


Figure: Mapping the **query** “animal” and a set of **key-value** pairs to an output.

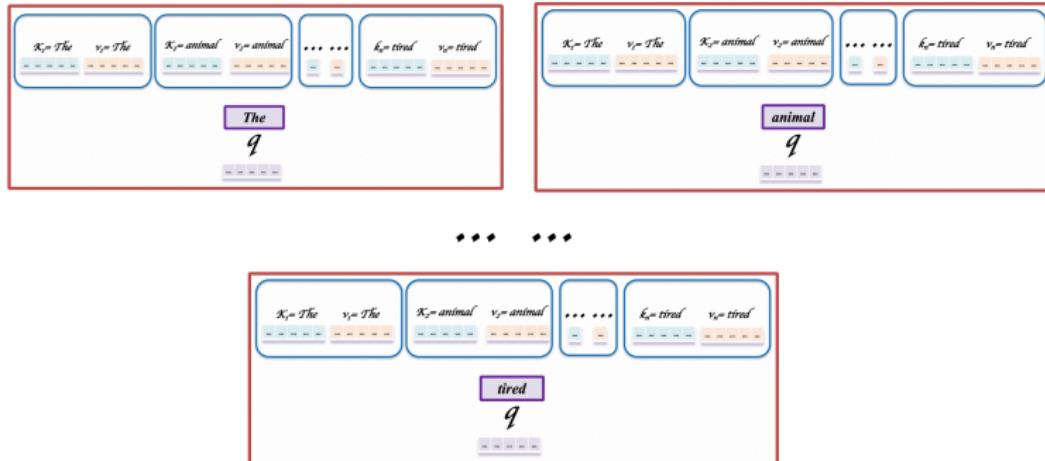
Self-Attention Illustrated: Query “tired”



The animal didn't cross the street because it was so tired

Figure: Mapping the **query** “tired” and a set of **key-value** pairs to an output.

Self-Attention Illustrated: All-Queries Overview'



The animal didn't cross the street because it was so tired

Figure: Mapping all **queries** and a set of **key-value** pairs to an output.

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{\mathbf{X}}{\sqrt{d_k}} \right) \mathbf{V} = \mathbf{Z}$$

The diagram illustrates the components of scaled dot-product attention. At the top, four boxes labeled Q , K^T , V , and Z are shown. Below them, two input matrices \mathbf{X} and \mathbf{V} are shown. Matrix \mathbf{X} is composed of three smaller square blocks, and matrix \mathbf{V} is composed of three smaller rectangular blocks. The equation below shows the computation of \mathbf{Z} from \mathbf{X} and \mathbf{V} .

Figure: **Scaled dot-product attention** in the Transformer.

Scaled Dot-Product Attention: Paper Illustration

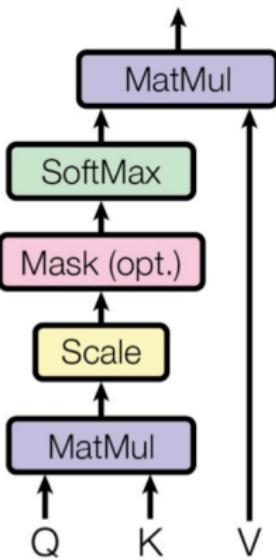


Figure: Scaled dot-product attention

Linear Projections of Query, Key, and Value (for Multi-Head Attention)

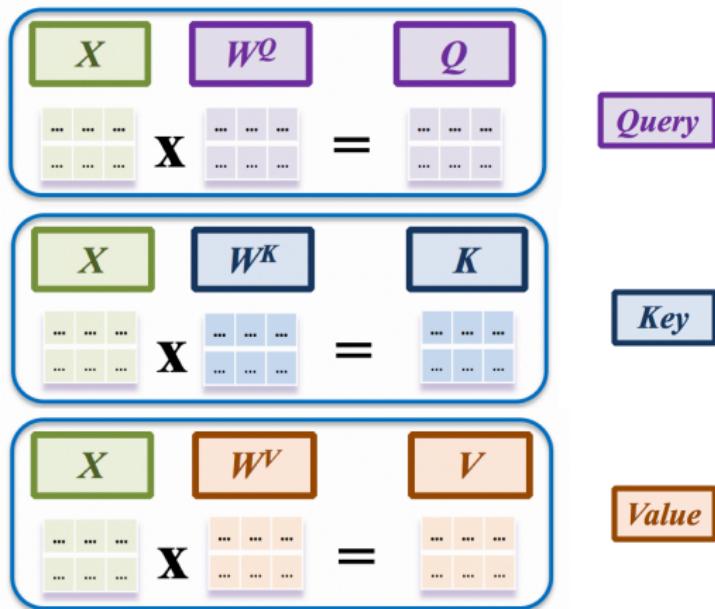


Figure: Linear projections of Q, K, and V. W matrixes are learned linear projections.

Multi-Head Attention

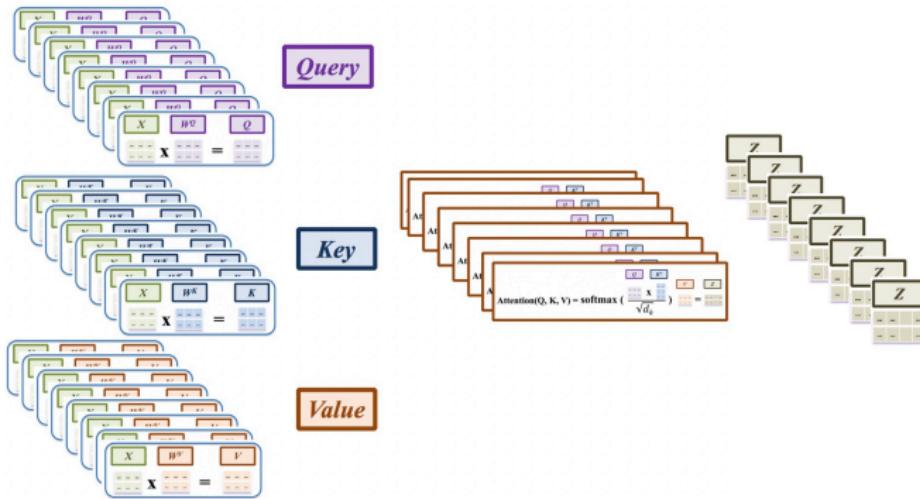


Figure: Multi-Head Attention with 8 attention heads.

Multi-Head Attention in Paper

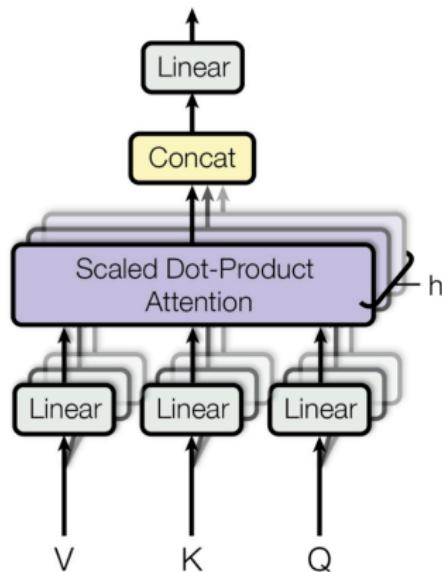


Figure: **Multi-Head Attention**

Multi-Head Attention: Output Concatenation

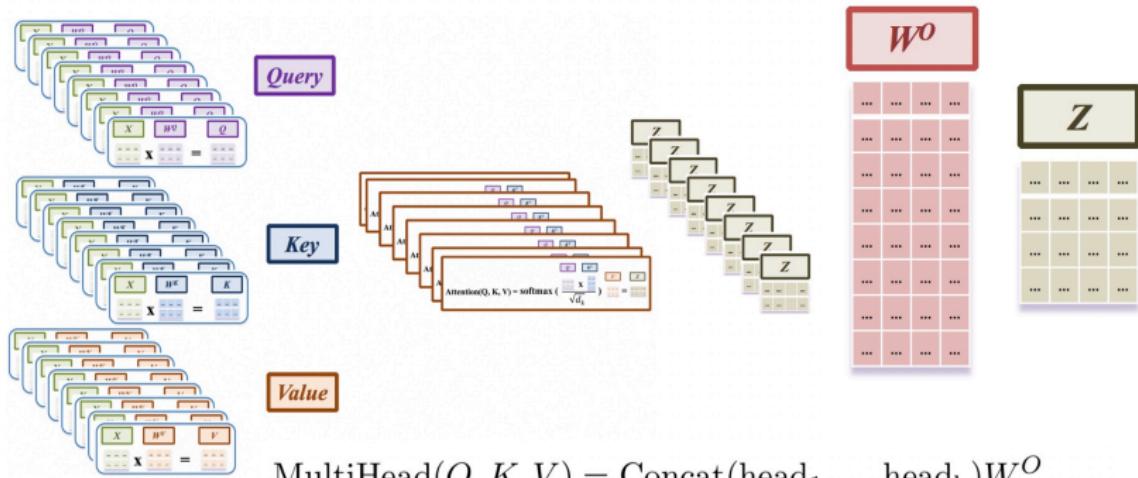
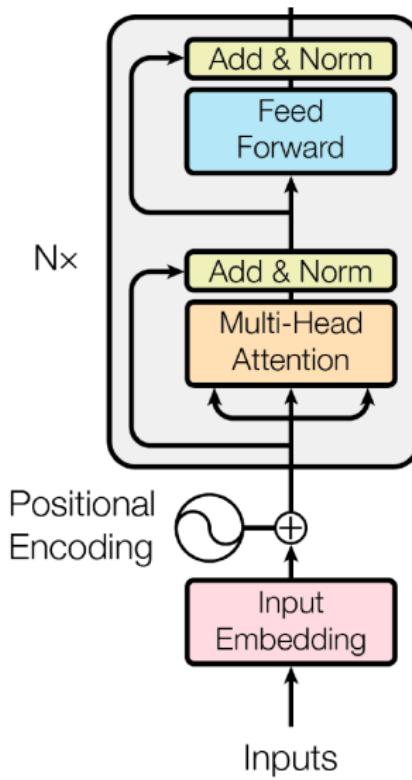


Figure: Output of attention heads are concatenated and projected to a linear layer.

Detailed Encoder Block



Background

- For a **feedforward network** with input x and output y , consider the L^{th} layer with α^L as the summed inputs to the neurons in that layer.
- The summed inputs are computed through a linear projection with the weight matrix W^l and the bottom-up inputs h^l given as follows:

1: Summed Inputs

$$\alpha_i^l = w_i^{lT} h^l$$

$$h_i^{l+1} = f(\alpha_i^l + b_i^l)$$

where $f(.)$ is an element-wise nonlinear function and w_i^l is the incoming weights to the i^{th} hidden units and b_i^l is the scalar bias parameter.

Background

- **Covariate Shift Challenge:** The gradients with respect to the weights in one layer are **dependent on the outputs of the neurons in the previous layer**, especially if these outputs change in a highly correlated way.
- BN normalizes the summed inputs to each hidden unit over the training cases.
- Specifically, for the i^{th} summed input in the L^{th} layer, BN rescales the summed inputs according to their variances under the distribution of the data.

2: BatchNorm

$$\bar{\alpha}_i^l = \frac{g_i^l}{\sigma_i^l} (\alpha_i^l - \mu_i^l)$$

$$\mu_i^l = \mathbb{E}_{x \sim P(x)} [\alpha_i^l]$$

$$\sigma_i^l = \sqrt{\mathbb{E}_{x \sim P(x)} [(\alpha_i^l - \mu_i^l)^2]}$$

- Where $\bar{\alpha}_i^l$ is normalized summed inputs to i^{th} hidden unit in L^{th} layer, g_i is a **gain parameter** scaling the normalized activation **before** the non-linear activation function.
- Note:** μ and σ are estimated using empirical samples from the current mini-batch

Layer Normalization

Stats over units in a single layer

- The layer normalization statistics are computed over all hidden units in the same layer:

3: LayerNorm in RNN

$$\mu^I = \frac{1}{H} \sum_{i=1}^H \alpha_i^I$$

$$\sigma^I = \sqrt{\frac{1}{H} \sum_{i=1}^H (\alpha_i^I - \mu^I)^2}$$

where H is # hidden units in the layer.

Layer Normalization Contd.

- The layer normalized recurrent layer (in RNN) **re-centers and re-scales** its activations using the extra normalization terms similar to below.

4: LayerNorm

$$h^t = f \left[\frac{g}{\sigma^t} \odot (a^t - \mu^t) + b \right]$$

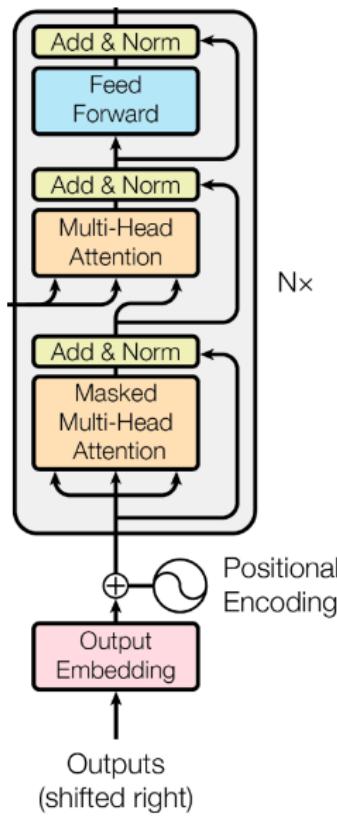
$$\mu^t = \frac{1}{H} \sum_{i=1}^H \alpha_i^t$$

$$\sigma^t = \sqrt{\frac{1}{H} \sum_{i=1}^H (\alpha_i^t - \mu^t)^2}$$

LN vs. BN

- Under layer norm, all the hidden units **in a layer share the same normalization terms μ and σ** , but **different training cases have different normalization terms**.
- Unlike batch normalization, **layer normalization does not impose constraints on the size of a mini-batch** (can be trained in the pure online regime with batch size 1).
- In RNNs, **avg magnitude of summed input to recurrent units tends to either grow or shrink** at every time step, leading to **exploding gradients or vanishing gradients**.
- LN in RNN, the normalization terms make it invariant to re-scaling all of the inputs to a layer, thus leading to **much more stabilize hidden-to-hidden dynamics**

Detailed Decoder Block



Whole Transformer Architecture

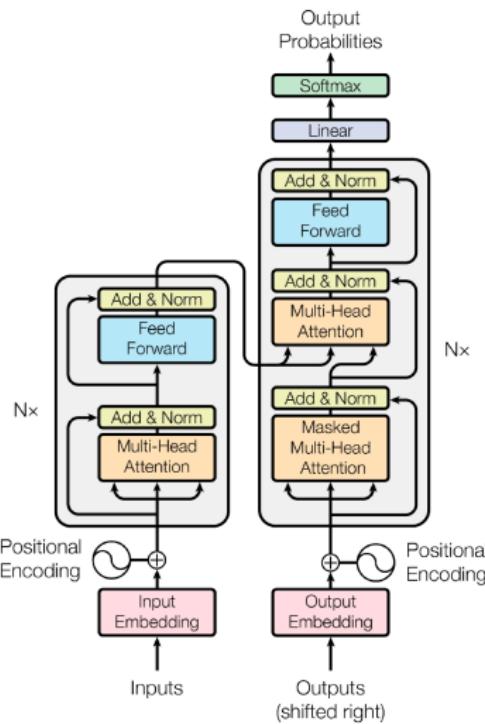


Figure: Transformer: Model Architecture.

Application of Attention in the Transformer

Attention in Transformer

- ① **Encoder 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**.
- ② In “**encoder-decoder attention**” layers: **queries come from the previous decoder layer**, and the **memory keys and values come from the output of the encoder**.
- ③ **Decoder self-attention layers:** Each position in the decoder attends to all positions in the decoder up to and including that position.

Transformer Computation in Comparison

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure: Transformer: Computational Cost

Transformer: Empirical Results

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		$2.3 \cdot 10^{19}$

Figure: Transformer results from paper

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova

Google AI Language

{jacobdevlin, mingweichang, kentonl, kristout}@google.com

Abstract

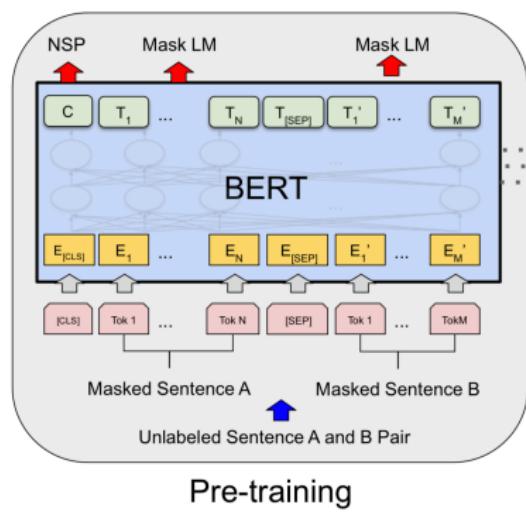
We introduce a new language representation model called **BERT**, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically

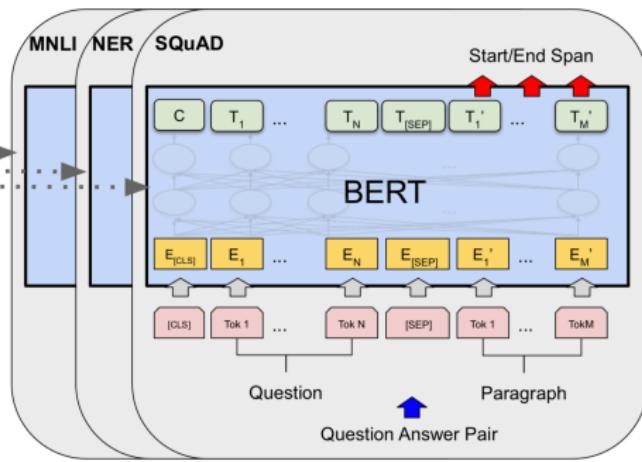
There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

We argue that current techniques restrict the

BERT Illustrated



Pre-training



Fine-Tuning