COLX-531: Neural Machine Translation

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

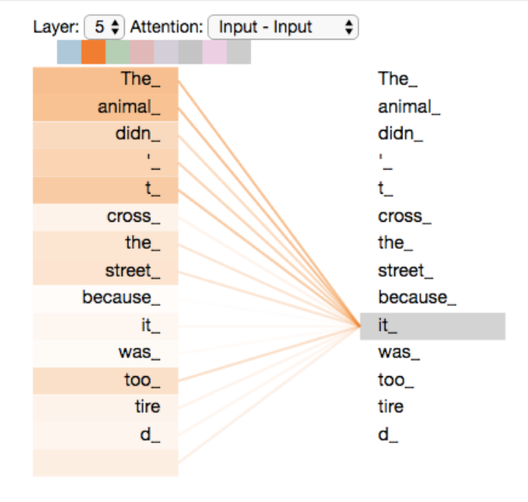


Figure: Self-attention. (Source: `http://jalammar.github.io`).

Transformer Architecture

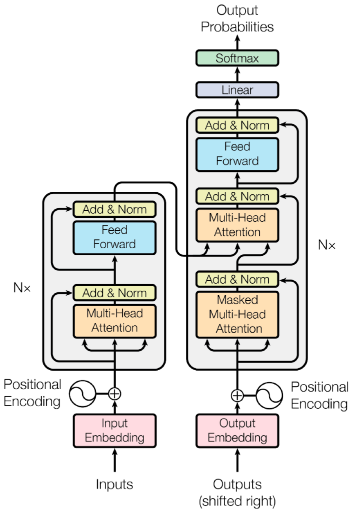


Figure: Transformer: Model Architecture.

Recall: Basic Encoder Decoder Idea

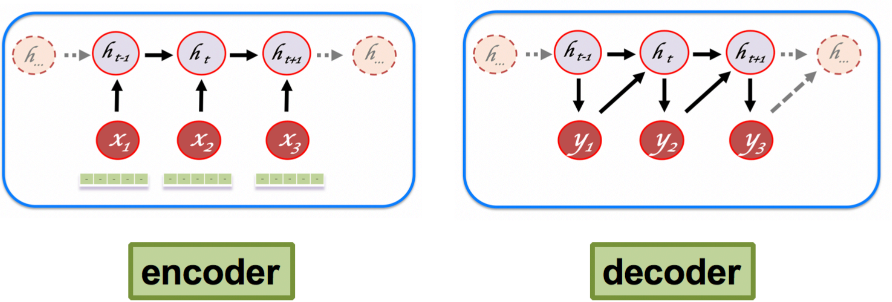


Figure: Transformer has an encoder and a decoder.

Transformer is based on encoder-decoder design

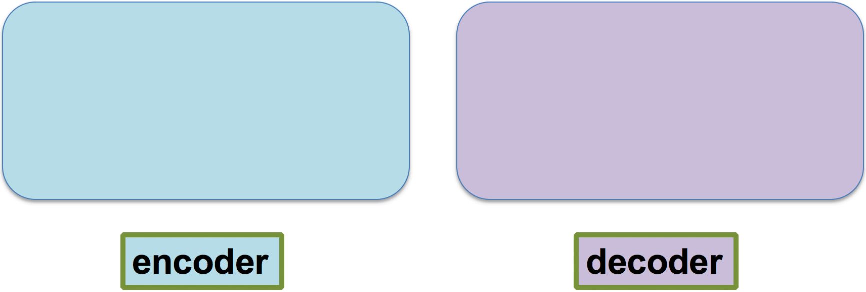


Figure: Transformer has an encoder and a decoder, **each with specialized building blocks**.

Encoder-Decoder

Basic Encoder-Decoder Operations

- **Encoder** maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$.
- Given $\mathbf{z}$, the **decoder** then generates an output sequence $(y_1, \ldots, y_m)$ 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**

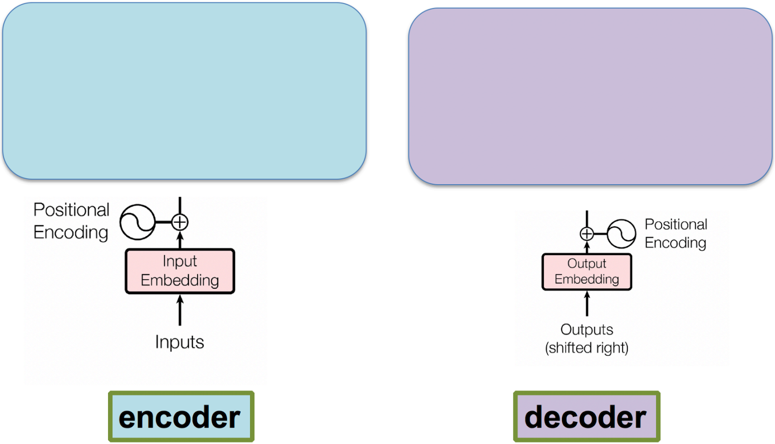


Figure: Embeddings for Encoder and Decoder.

Encoding Word Position

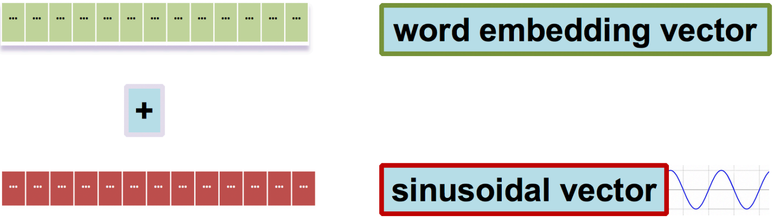


Figure: Simple Addition for encoding word positions

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.

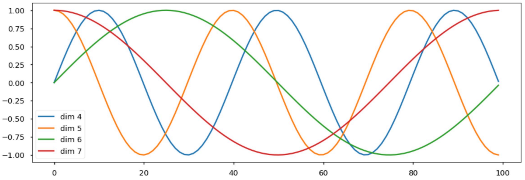


Figure: https://nlp.seas.harvard.edu/2018/04/03/attention.html.

Positional Encoding Example

PE Example

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

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

where the formula for positional encoding is as follows

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

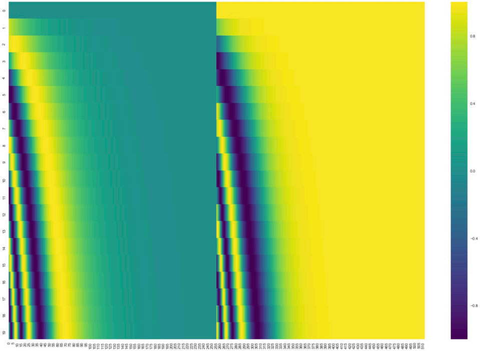
$$\mathrm{PE}(pos, 2i+1) = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right).$$

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.

Positional Encoding Illustrated

In the following figure, each row corresponds the a 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 Word Embeddings

```python
import math
import numpy as np
import torch
import matplotlib.pyplot as plot
#------------------------------
max_seq_len= 4
d_model= 4
d_embed= 4
"""
Let's create a tensor of word embeddings with 4 dimensions for each word (d_embed),
for 4 words (max_seq_len)
"""
word_embeddings= torch.rand( (max_seq_len, d_embed))
print(word_embeddings)
```

```
tensor([[0.6981, 0.3755, 0.7078, 0.0328],
        [0.9458, 0.2585, 0.0134, 0.2634],
        [0.9467, 0.7712, 0.4704, 0.6277],
        [0.2791, 0.9099, 0.8795, 0.9542]])
```

Tensor for Positions

```python
"""
Let's create a tensor of positional encodings, populate by zeros for now.
Each word vector will have a corresponding positional vector within this pe tensor
"""
pe = torch.zeros(max_seq_len, d_embed) # positional encoding
print(pe)
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

```python
"""
Let's change the positional encodings tensor such that each dimension
is changed by sin or cos, as approbriate
"""
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)/d_model))  )
print(pe)
```

```
tensor([[ 0.0000,  1.0000,  0.0000,  1.0000],
        [ 0.8415,  0.5403,  0.0001,  1.0000],
        [ 0.9093, -0.4161,  0.0002,  1.0000],
        [ 0.1411, -0.9900,  0.0003,  1.0000]])
```

Figure: A tensor for sinusoidal vectors at each position *i* **(sin)** and *i+1* **(cos)**.

Adding Embeddings and Positions for **PE**

```
"""
Let's retrieve positional encoding at index 3
and also embedding vector for index 3.
"""
pe3= pe[2] # Positional encoding at index 3.
w3_embed = word_embeddings[2] # Embedding for word 3
#----------------------------------------------------------------
print("\n-- Positional encoding for w in position 3:\n\n", pe3)
print("\n-- Embedding vector for w in position 3:\n\n", w3_embed)
```

```
-- Positional encoding for w in position 3:

 tensor([ 0.9093, -0.4161,  0.0002,  1.0000])

-- Embedding vector for w in position 3:

 tensor([0.9467, 0.7712, 0.4704, 0.6277])
```

```
# 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([1.7087, 0.0290, 0.2876, 1.9235], dtype=torch.float64)
```

Figure: Encoding for w3 is by adding its embed and sinusoidal pos vectors.

Encoder Blocks (6 Identical Blocks)

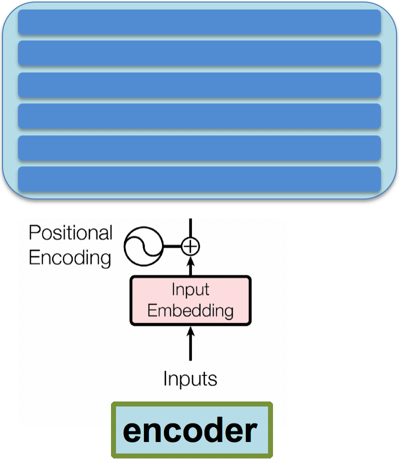
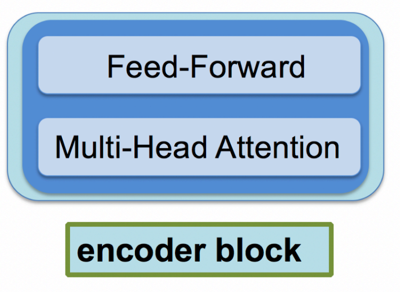


Figure: Encoder's 6 identical blocks.

An Encoder Block



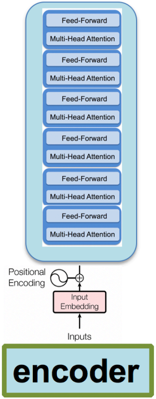


Figure: Each encoder block has two sub-layers, a multi-head attention and a feed-forward.

Self-Attention

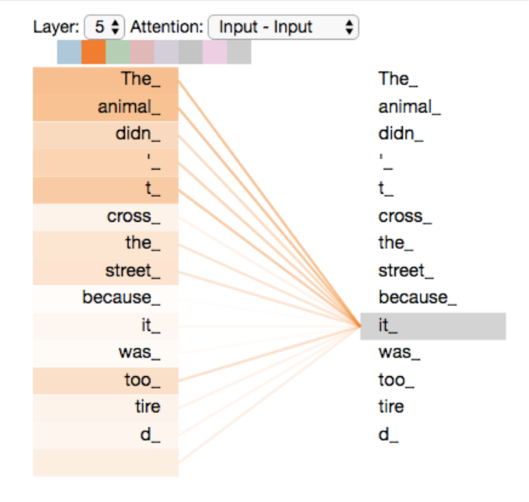
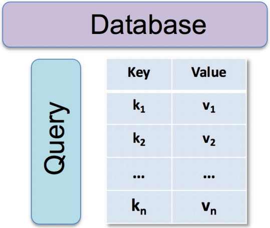


Figure: Self-attention. (Source: http://jalammar.github.io).

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.



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

MT Example of Query, Key, and Value

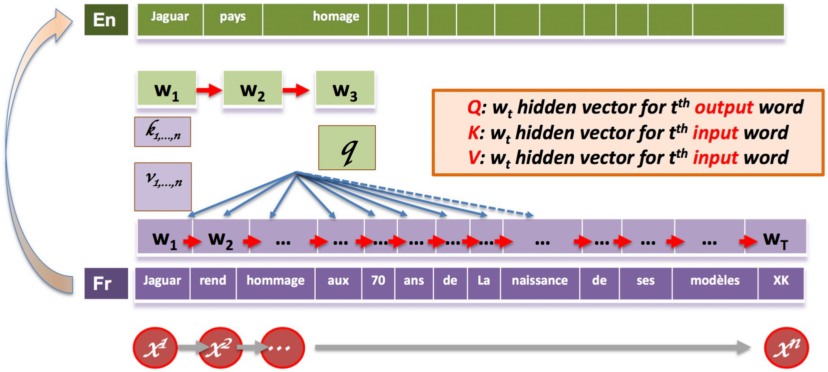


Figure: q, k, and v in MT.

Self-Attention in Transformer with q, k, v

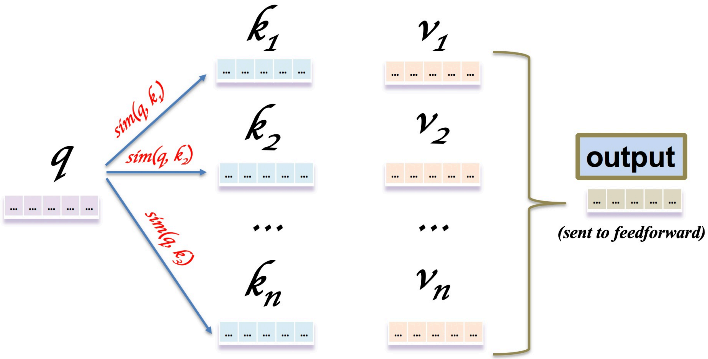


Figure: Mapping a **query** and a set of **key-value** pairs to an output. **Similarity s** will be based on **dot product**. Well, *scaled dot product*. . .

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

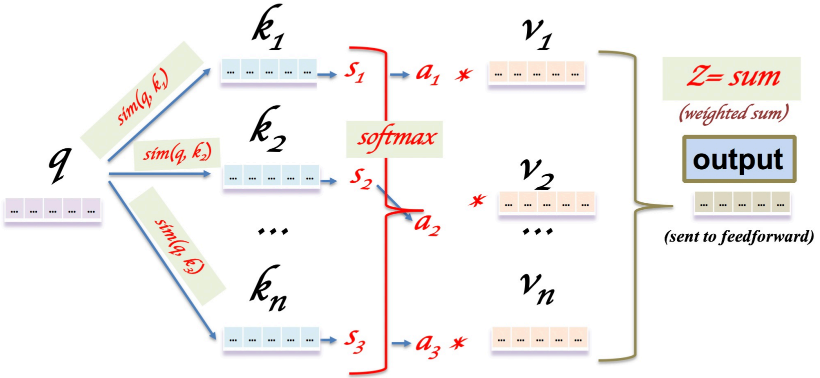


Figure: s=sim(q, k_i). $\alpha = softmax(s)$. $z = \sum_i a_i v_i$

Self-Attention Illustrated: Query "the"

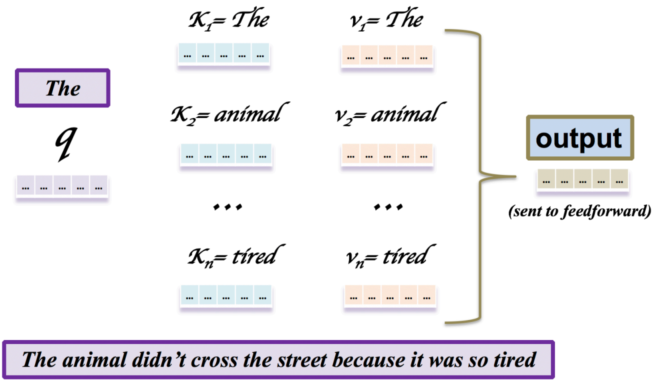


Figure: Mapping the **query** "the" and a set of **key-value** pairs to an output.

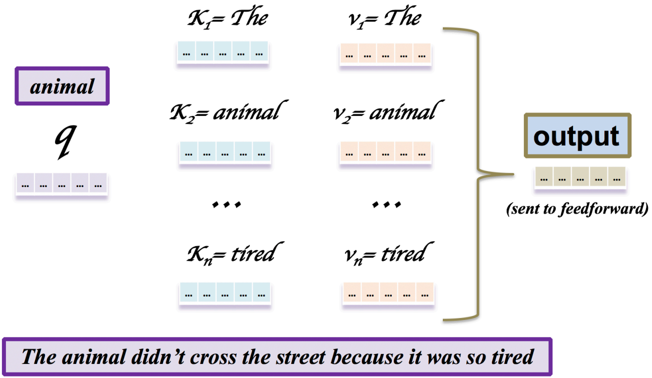


Figure: Mapping the **query** "animal" and a set of **key-value** pairs to an output.

Self-Attention Illustrated: Query "tired"

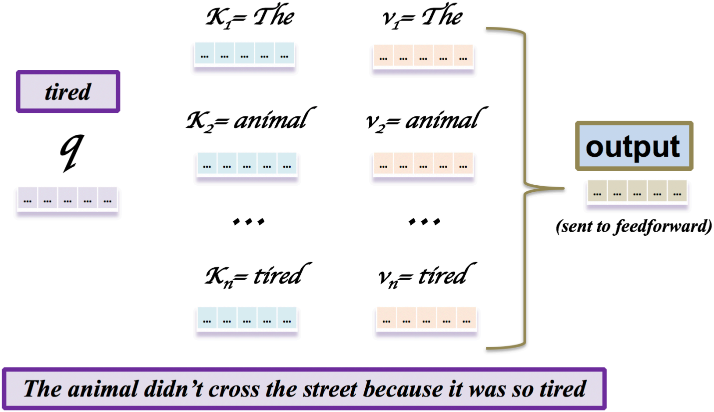


Figure: Mapping the **query** "tired" and a set of **key-value** pairs to an output.

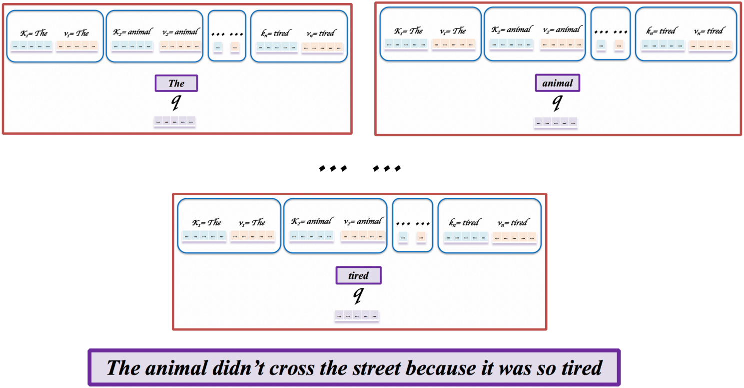


Figure: Mapping all **queries** and a set of **key-value** pairs to an output.

Scaled Dot-Product Attention

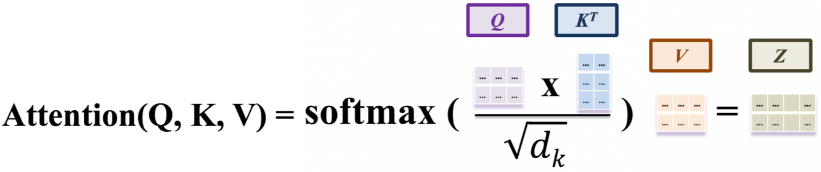


Figure: **Scaled dot-product attention** in the Transformer.

Scaled Dot-Product Attention: Paper Illustration

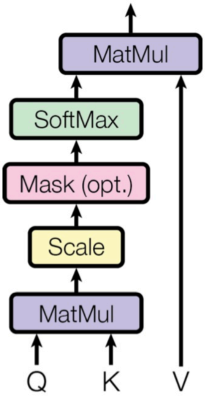


Figure: **Scaled dot-product attention**

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

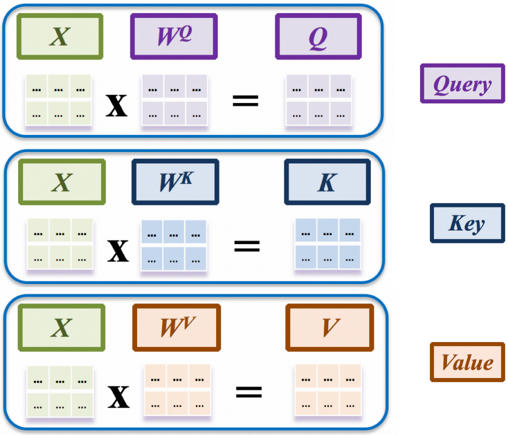


Figure: Linear projections of Q, K, and V. **W** matrixes are learned linear projections.

Multi-Head Attention

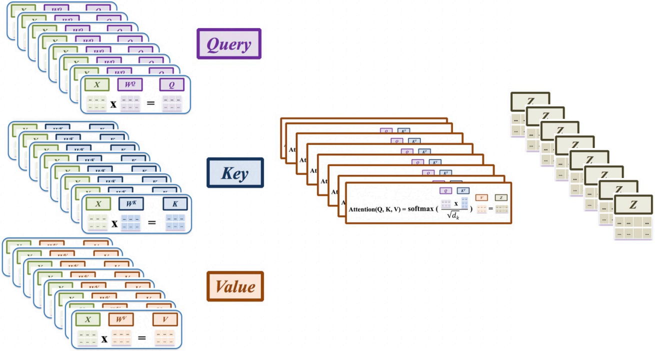


Figure: Multi-Head Attention with 8 attention heads.

Multi-Head Attention in Paper

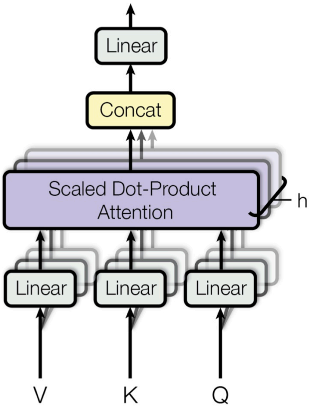
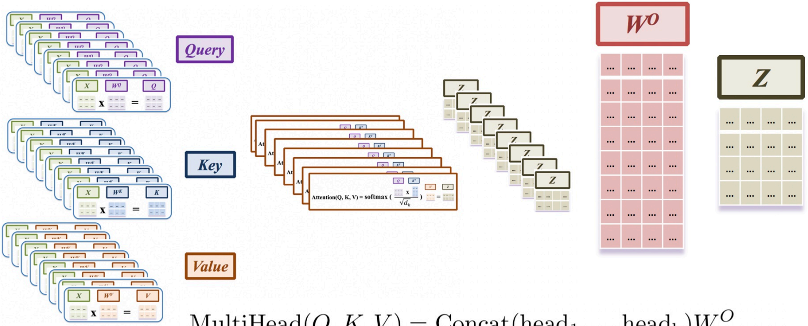


Figure: **Multi-Head Attention**

Multi-Head Attention: Output Concatenation

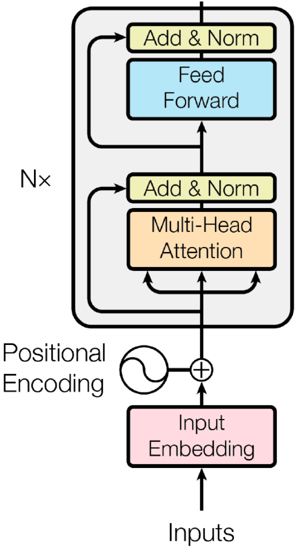


$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},...,\mathrm{head_h})W^O$$

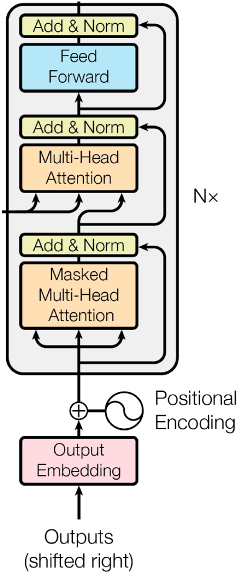
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

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

Detailed Encoder Block



Detailed Decoder Block



Masked Attention

1: Attention

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

2: Masked Attention

$$\color{red}{Masked}\,Attention(Q, K, V) = softmax(\frac{QK^T + \color{red}{M}}{\sqrt{d_k}})V$$

Masking Matrix **M**

M is a matrix of $\color{red}{0}$'s (in lower triangular) and $\color{red}{-\infty}$ (in upper triangular)

Masked Attention II

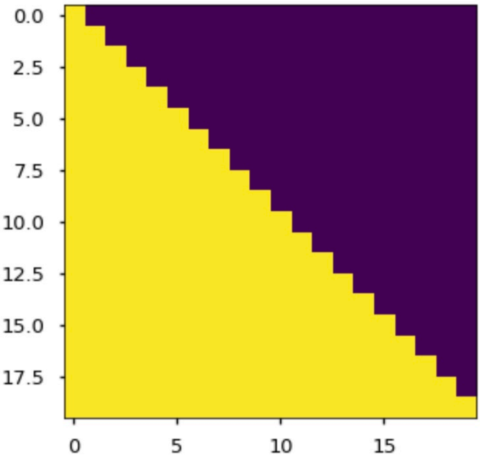


Figure: Masking in Transformer. (Source: https://nlp.seas.harvard.edu/2018/04/03/attention.html).

Residual Learning

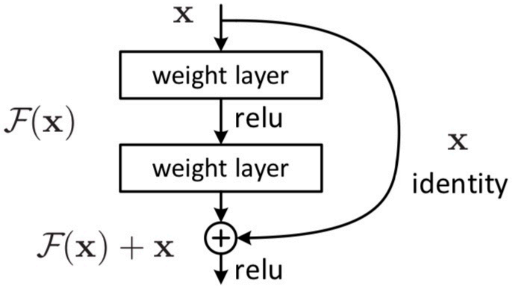


Figure 2. Residual learning: a building block.

Figure: Source: He et al. ("Deep Residual Learning for Image Recognition", 2016)
http://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

Whole Transformer Architecture

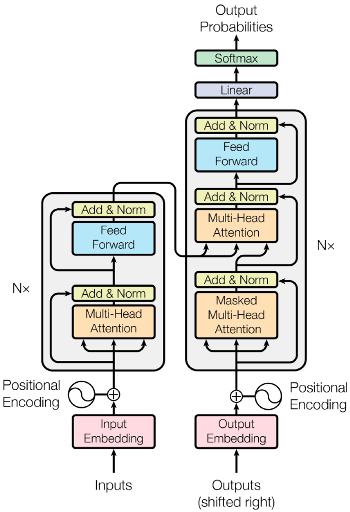


Figure: Transformer: Model Architecture.

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.