“Attention is All You Need”

EXPLORING TRANSFORMER MODELS

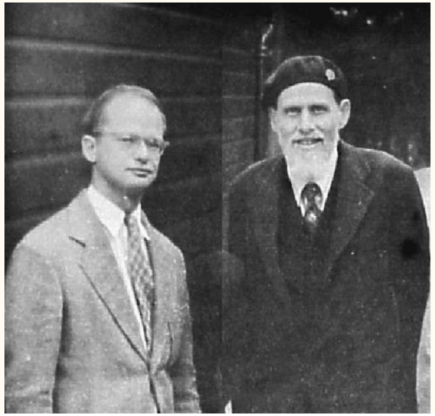
Koerner Gray-Buchta

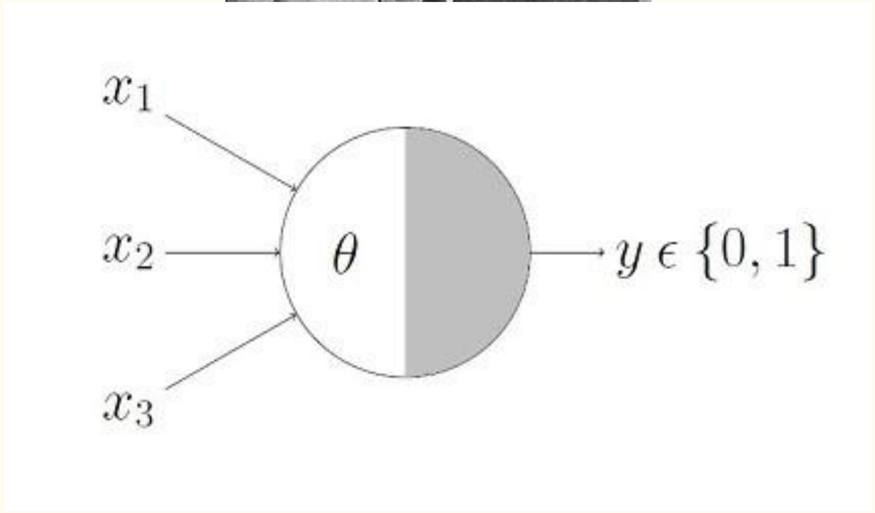
A Brief History of Neural Computing

- Pitts and McCulloch
- Rosenblatt
- Minsky & the AI Winter
- Basic NNs and Deep Learning

Pitts & Mcculloch 1943

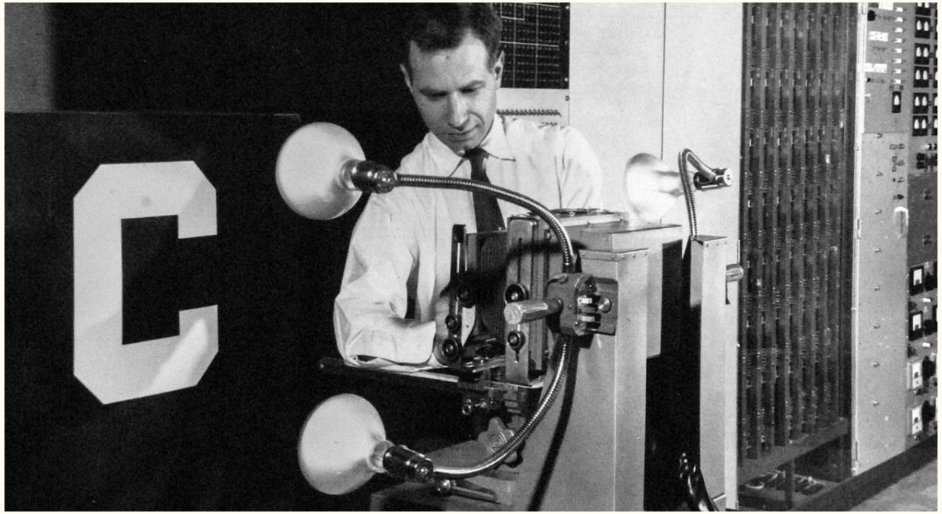
1. First mathematical model of a neuron
2. Theta is the binary threshold
3. Bertrand Russell and Walter Pitts
4. Inspired by Leibniz (“Mind as the universal computer”)
5. Ignored until use by John von Neumann and Norbert Wiener





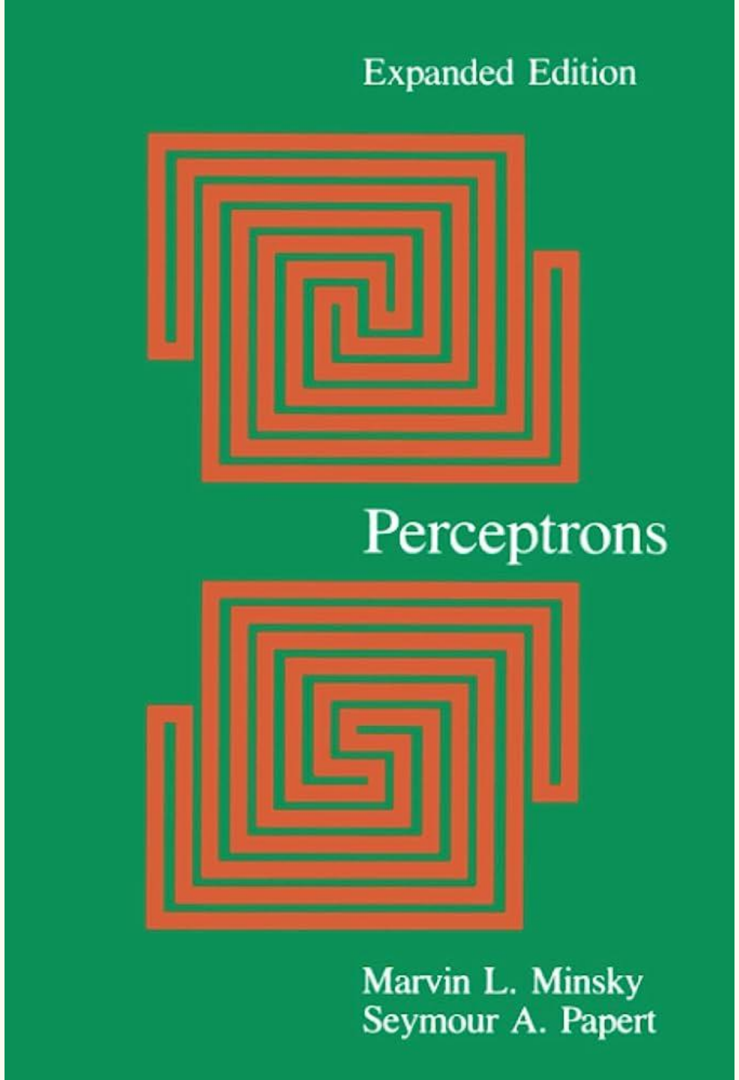
The Perceptron 1958

1. Mechanical hardware implementation based on Pitts-Mcculloch neuron
2. Weights encoded in potentiometer positions driven by motors
3. Essentially a single-layer neural network
4. Used for binary classification, early computer vision



Minsky-Papert Backlash

1. "Perceptrons" written by Marvin Minsky at MIT in 1969
2. Proved that SLPs couldn't solve certain simple problems (XOR, parity)
3. Led to temporary abandonment of neural computing for symbolic, rule-based approaches (leading to AI winter)



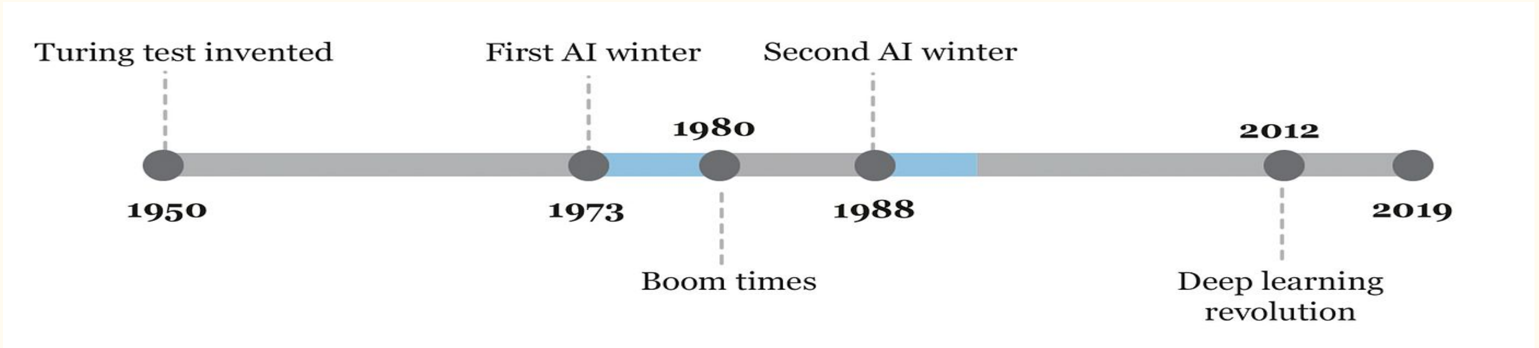
AI Winter (1970-2010)

1. Lighthouse Report, UK 1973, failure of machine translation: over-promising and under-delivering

2. Changes in DARPA funding - no undirected research

3. Academic squabbles over limited computational resources

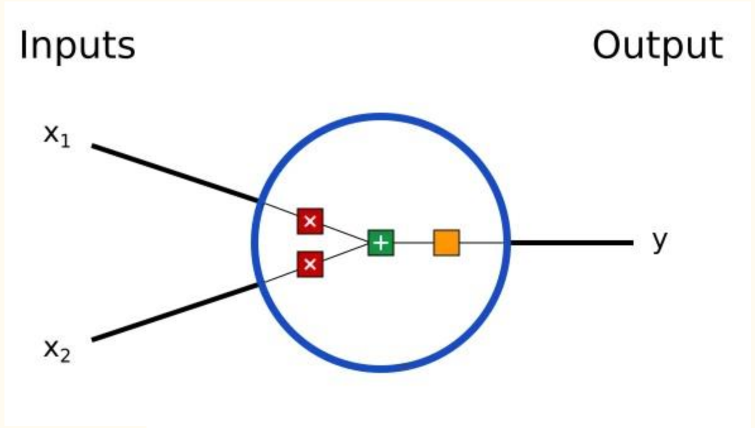
4. *“The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence”* - NYT 1958



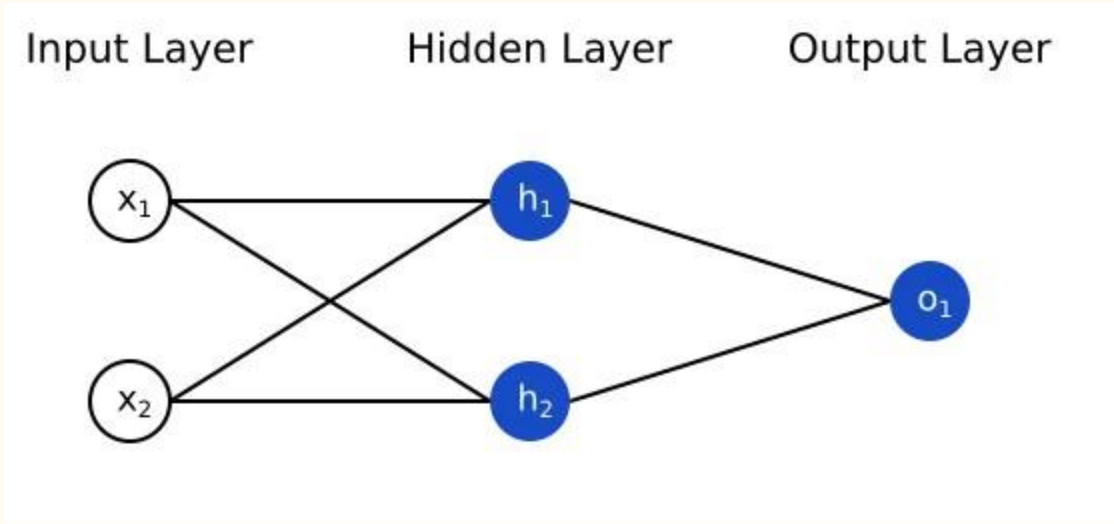
Basic Neural Networks

Weights are calculated during training by defining an error/loss function.

Weights are given a random seed, then gradually nudged according to the magnitude of their contribution to the error (the partial derivative of the weight w/r/t the error)



$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$



Deep Learning Surges

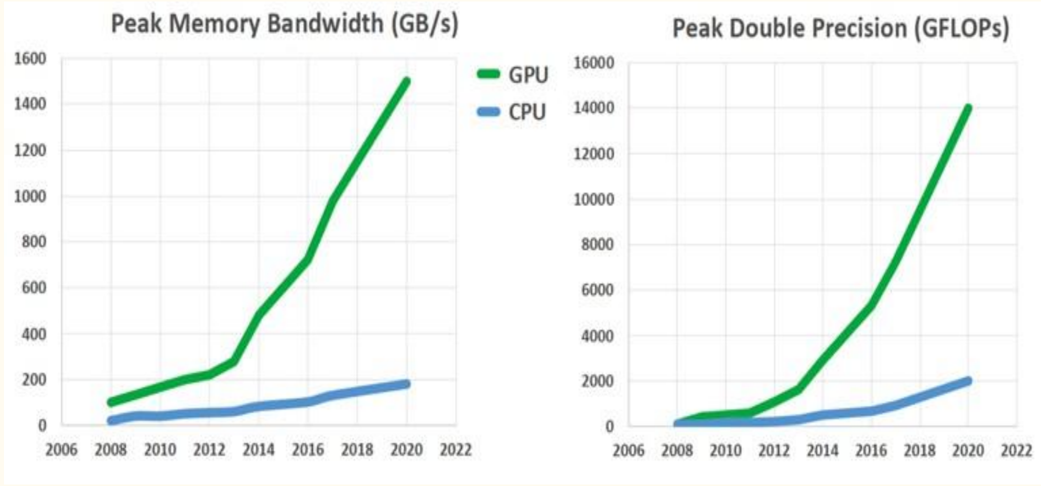
1. Exponential increase in computing power (GPUs, TPUs)

2. Progress in developing new architectures/training methods, depth of hidden layers

3. Big companies with Big Data

4. CNNs and RNNs (image, seq2seq)

5. Robotics, NLP, Vision, Search



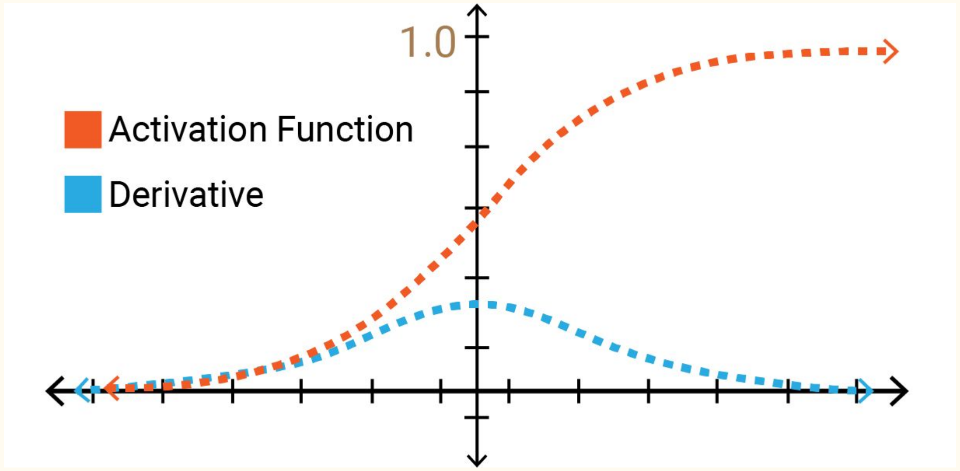
Problems with CNNs/RNNs

1. Vanishing or exploding gradient (float rounded to zero)

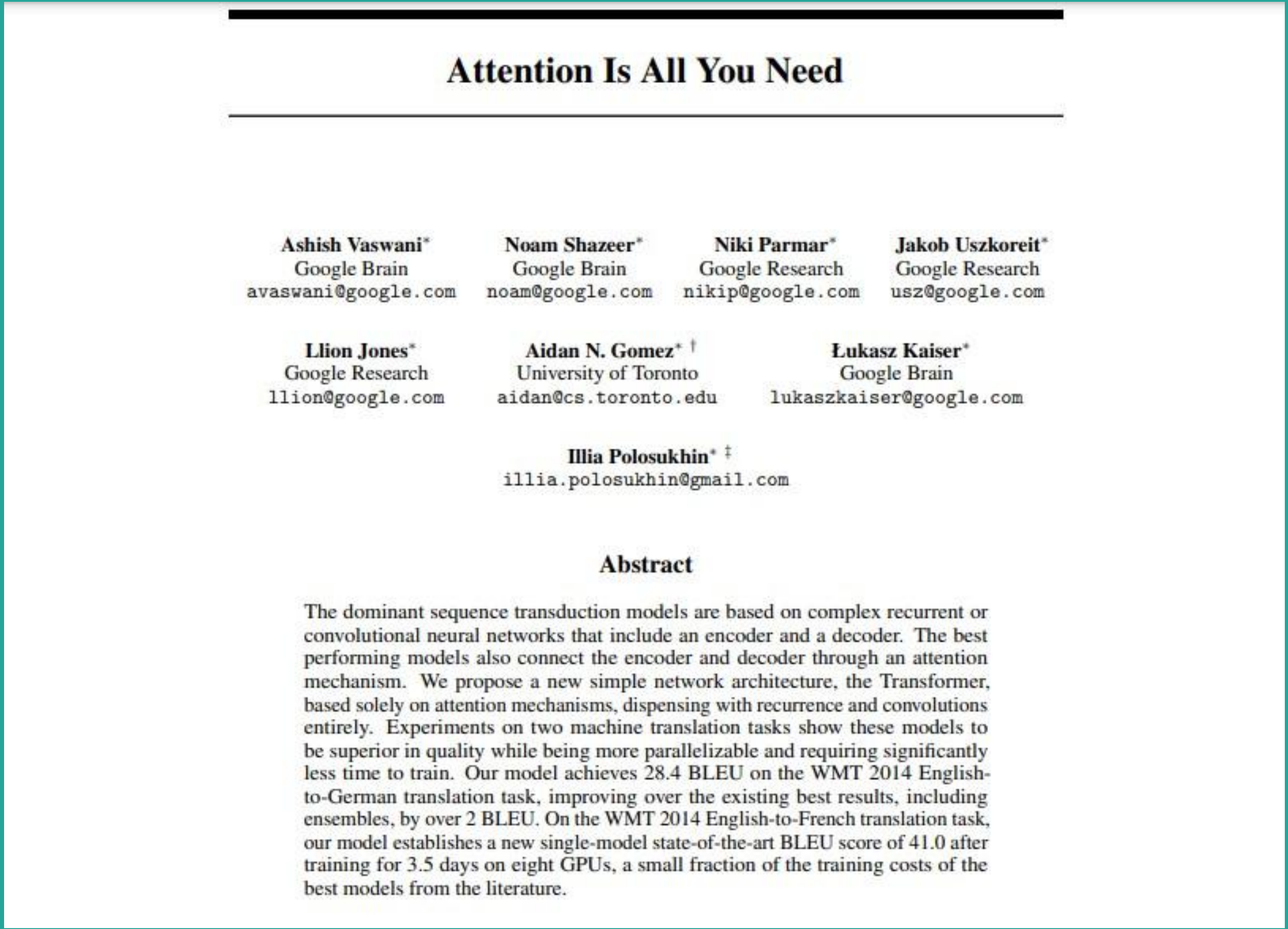
2. Difficulty managing long-term dependencies

3. LSTM is an improvement, but still suffers from exploding gradients

4. Sequential nature prevents parallelization



Enter Transformers!



Attention Is All You Need

Ashish Vaswani[*]
Google Brain
avaswani@google.com

Noam Shazeer[*]
Google Brain
noam@google.com

Niki Parmar[*]
Google Research
nikip@google.com

Jakob Uszkoreit[*]
Google Research
usz@google.com

Llion Jones[*]
Google Research
llion@google.com

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

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

ENCODER STRUCTURE

- Word-embeddings
- Positional encoding
- Self-attention

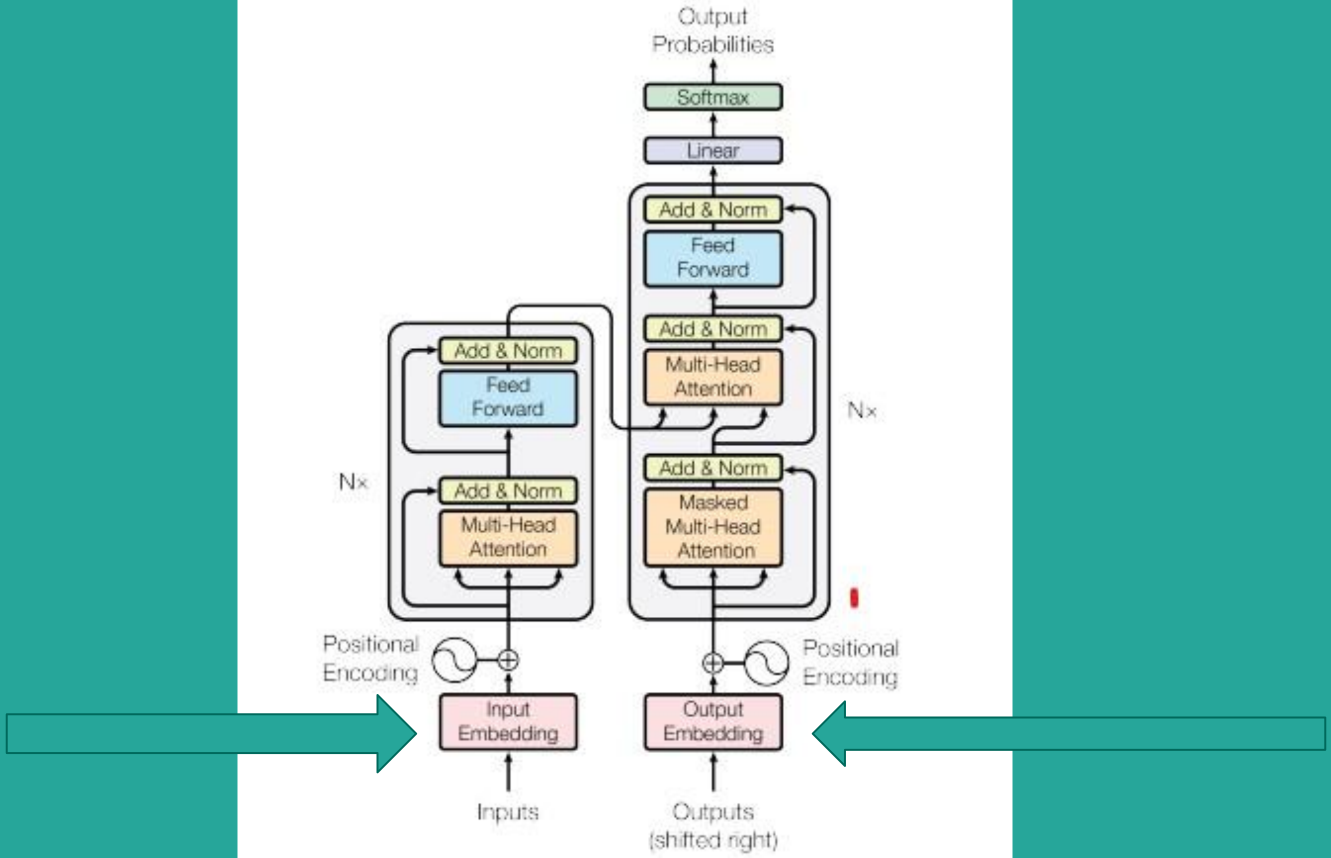


Figure 1: The Transformer - model architecture.

Encoding Words

Problem:

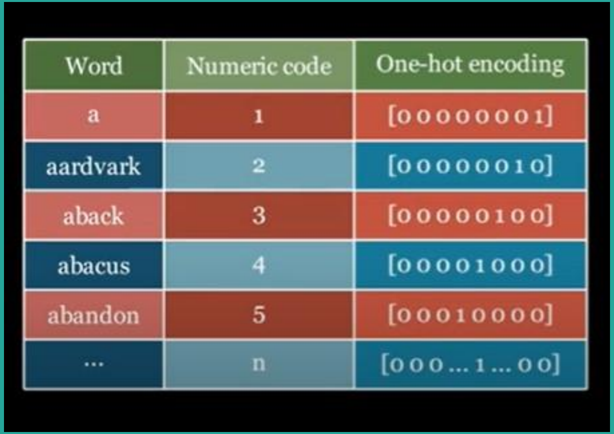
A word is not a number. How do we represent it to a computer?

Naive Answer:

Assign each word a number.

Further problem:

This gives no information about a word’s meaning (semantics). Also, the numbers get too big



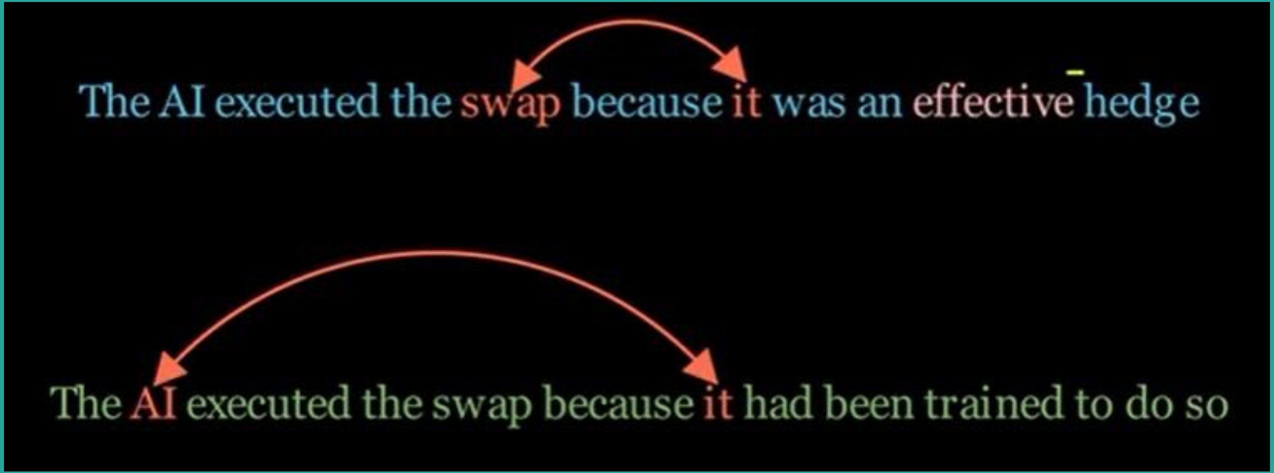
Word	Numeric code	One-hot encoding
a	1	[0 0 0 0 0 0 0 1]
aardvark	2	[0 0 0 0 0 0 1 0]
aback	3	[0 0 0 0 0 1 0 0]
abacus	4	[0 0 0 0 1 0 0 0]
abandon	5	[0 0 0 1 0 0 0 0]
...	n	[0 0 0 ... 1 ... 0 0]

"You shall know a word by the company it keeps!"

- Cambridge Linguist John Firth, 1962

Insight: meaning is relational, rather than absolute (dictionary)

We need an encoding for a word that contains information about how it relates to other words.


The AI executed the swap because it was an effective hedge
The AI executed the swap because it had been trained to do so

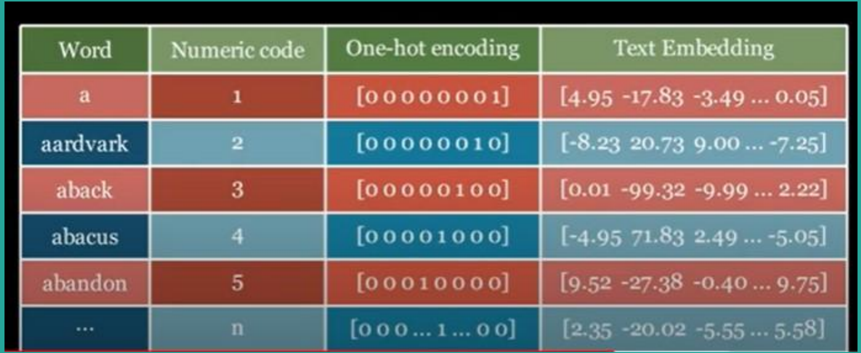
Creating Word Embeddings

Solution: train a network where each word corresponds to a neuron in the input layer (10,000+ neurons/words in vocabulary)

Each neuron in the input layer is connected to every neuron in the hidden layer (300 neurons)

After training, the learned weights (all 300) connecting a word to the hidden layer is unique to that word and semantically meaningful

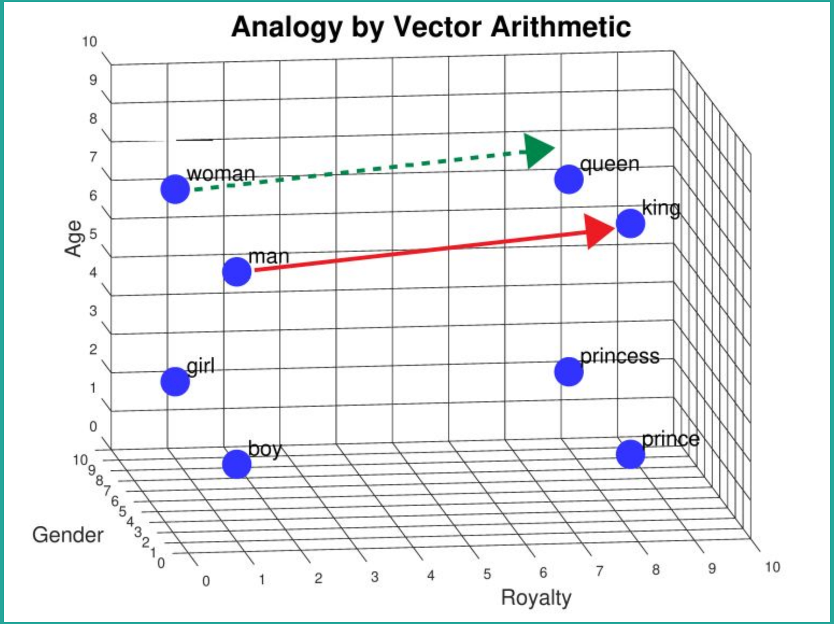
The (pre-trained) word embedding is thus a vector composed of those 300 weight values



Word	Numeric code	One-hot encoding	Text Embedding
a	1	[0 0 0 0 0 0 0 1]	[4.95 -17.83 -3.49 ... 0.05]
aardvark	2	[0 0 0 0 0 0 1 0]	[-8.23 20.73 9.00 ... -7.25]
aback	3	[0 0 0 0 0 1 0 0]	[0.01 -99.32 -9.99 ... 2.22]
abacus	4	[0 0 0 0 1 0 0 0]	[-4.95 71.83 2.49 ... -5.05]
abandon	5	[0 0 0 1 0 0 0 0]	[9.52 -27.38 -0.40 ... 9.75]
...	n	[0 0 0 ... 1 ... 0 0]	[2.35 -20.02 -5.55 ... 5.58]

Utility of Word Embeddings

Allows for vector arithmetic in a (300+) dimension semantic space where words with similar meanings have similar vectors (and thus positions in that space)



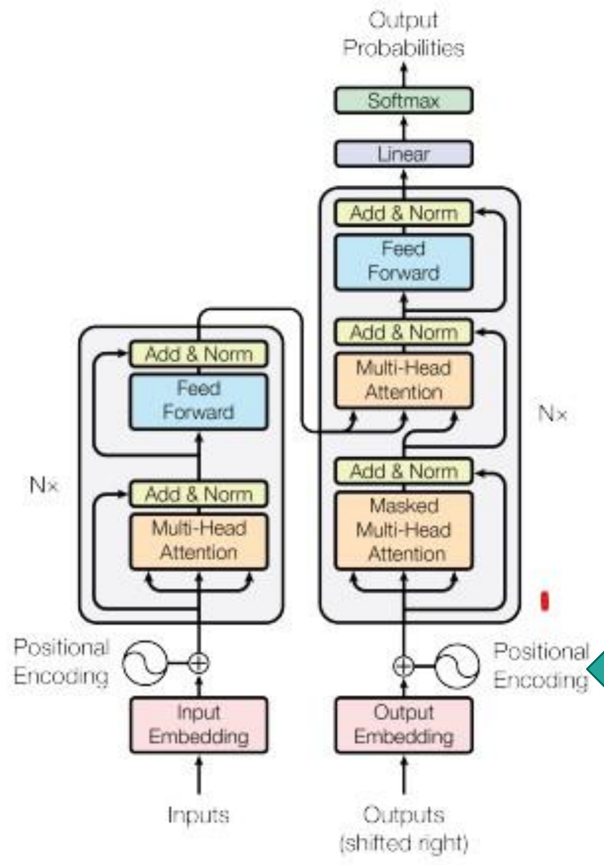


Figure 1: The Transformer - model architecture.

Positional Encoding

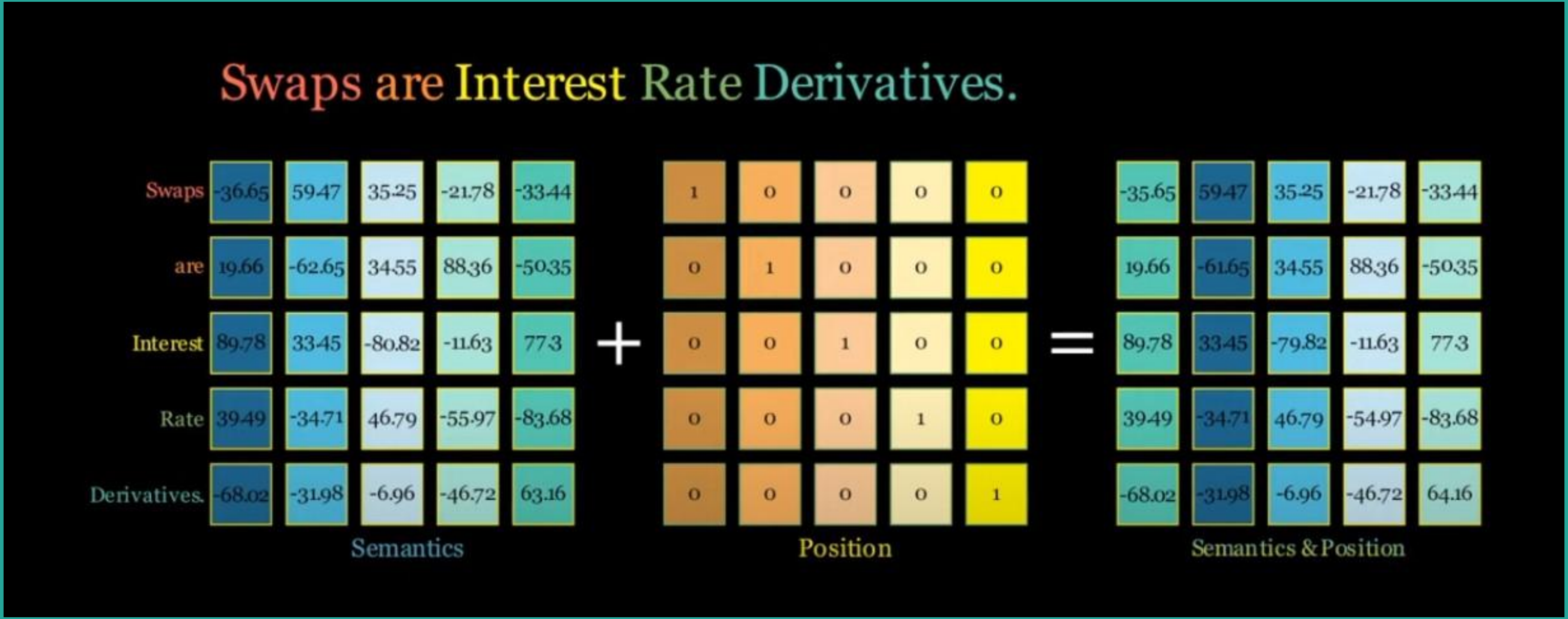
RNNs (including LSTMs) process tokens (words) sequentially, maintaining positional information, but preventing parallelization

Transformers process each token (word) in a sequence in parallel, allowing for faster operations, but this means they lose positional information. To a transformer, the following two sentences would have the same representation:

“Who are you” (Question)
“Who you are” (Statement)

So: how can we encode positional information into each input word-embedding vector?

Naive Solution: One Hot Encoding



Swaps are Interest Rate Derivatives.

Swaps	-36.65	59.47	35.25	-21.78	-33.44
are	19.66	-62.65	34.55	88.36	-50.35
Interest	89.78	33.45	-80.82	-11.63	77.3
Rate	39.49	-34.71	46.79	-55.97	-83.68
Derivatives.	-68.02	-31.98	-6.96	-46.72	63.16

Semantics

+

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Position

=

-35.65	59.47	35.25	-21.78	-33.44
19.66	-61.65	34.55	88.36	-50.35
89.78	33.45	-79.82	-11.63	77.3
39.49	-34.71	46.79	-54.97	-83.68
-68.02	-31.98	-6.96	-46.72	64.16

Semantics & Position

Only encodes absolute position - no relative position information

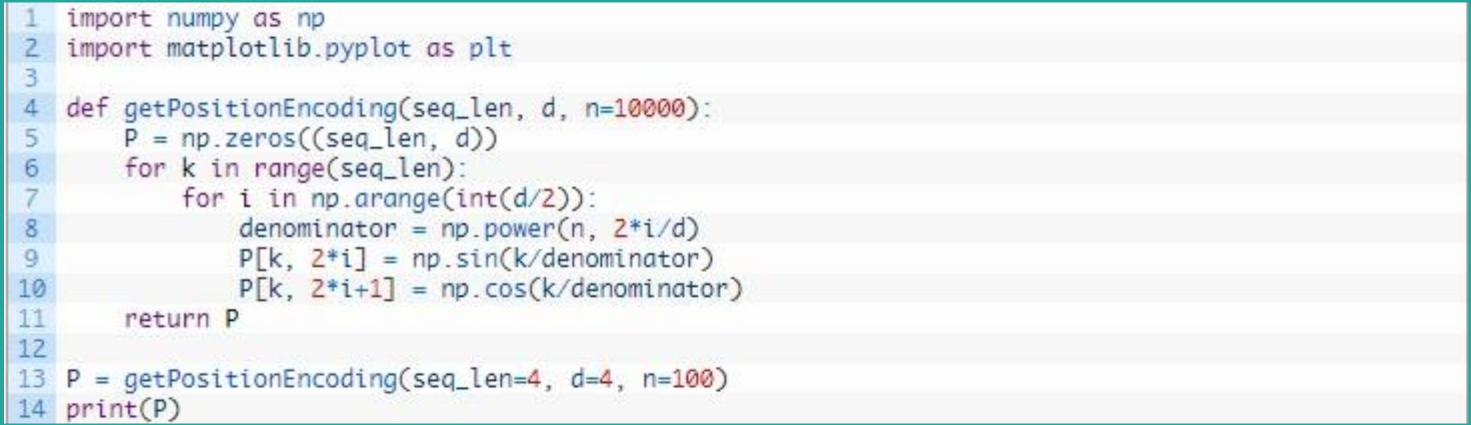
Better Solution: Trig Functions

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

Each dimension of the positional encoding corresponds to a sinusoid

The positional encoding vector and word-embedding vector are summed element-wise

Easy to pick up on by the model, may allow extrapolation


```python
import numpy as np
import matplotlib.pyplot as plt

def getPositionEncoding(seq_len, d, n=10000):
    P = np.zeros((seq_len, d))
    for k in range(seq_len):
        for i in np.arange(int(d/2)):
            denominator = np.power(n, 2*i/d)
            P[k, 2*i] = np.sin(k/denominator)
            P[k, 2*i+1] = np.cos(k/denominator)
    return P

P = getPositionEncoding(seq_len=4, d=4, n=100)
print(P)
```

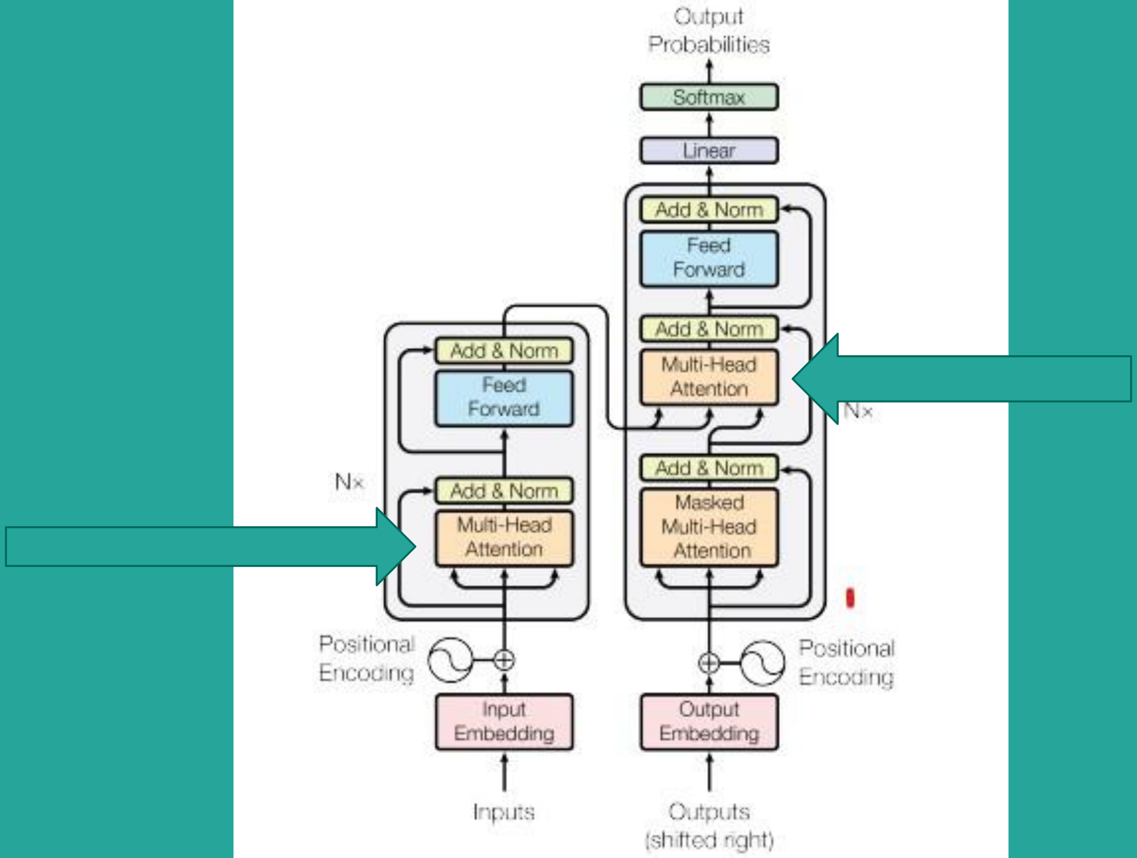
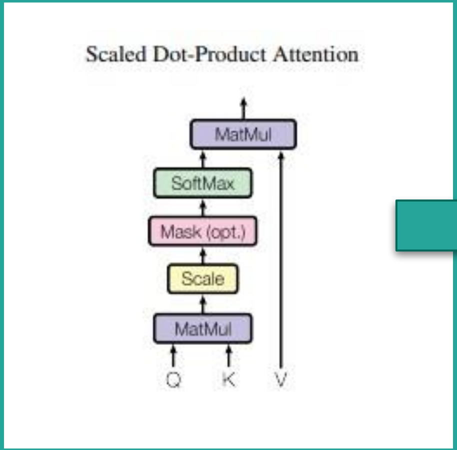


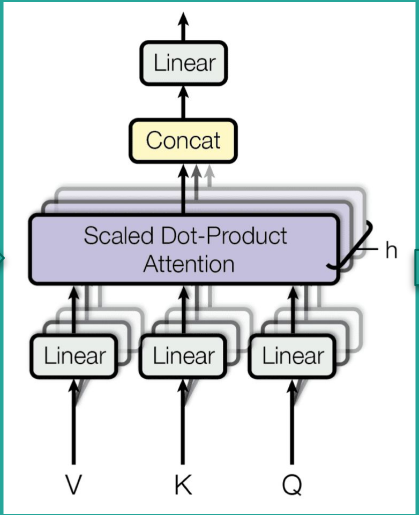
Figure 1: The Transformer - model architecture.

Attention

Attention mechanisms allow us to emphasize ("attend to") certain data-points while de-emphasizing others (cognitive attention)

Words "pay attention" to other words





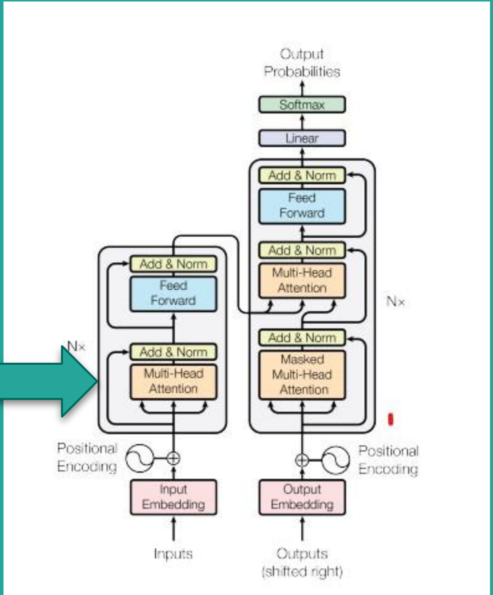
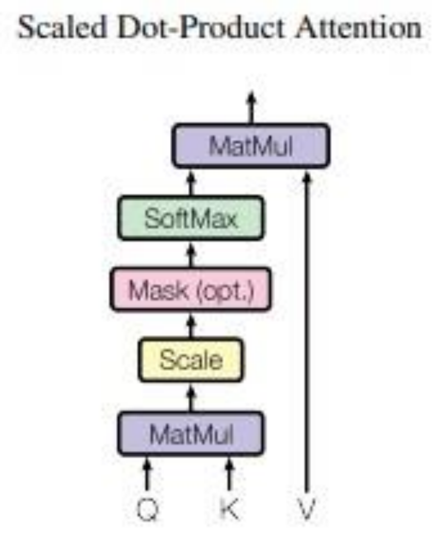


Figure 1: The Transformer - model architecture.

Scaled Dot-Product Attention



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

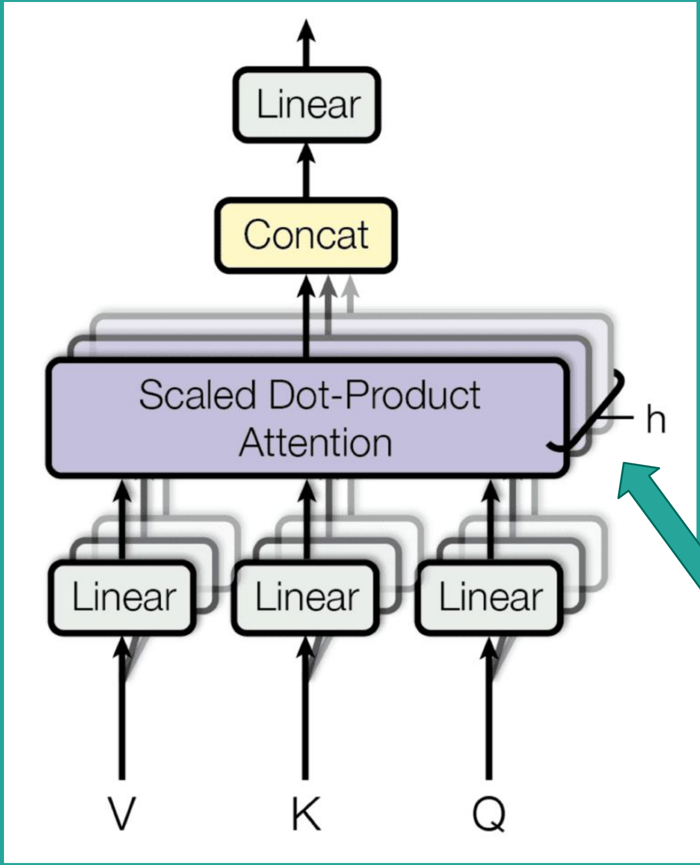
Query vector: represents the unique word we want to compute attention scores relative to
Key vector: identifies the other words in the sequence, the context
Value vector: represents the actual semantic information of the tokens from the key vectors

Left side computes attention weights to be applied to the value vector

Scaling prevents explosion

Product faster than sum

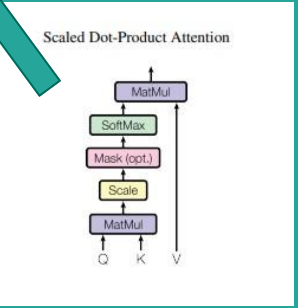
Multi-head Attention



$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},\dots,\mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Each query vector captures a different aspect of the input sequence, so each head processes the entire input sequence

Allows the model to “attend” to different parts of the input sequence, all in parallel



Three Uses of Attention

1. In the **encoder** section: " *Each position in the encoder can attend to all positions in the previous layer of the encoder.*"
2. In the **decoder:** ". . . *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*"
3. At the **junction between encoder and decoder** sections: "*This allows every position in the decoder to attend over all positions in the input sequence*"

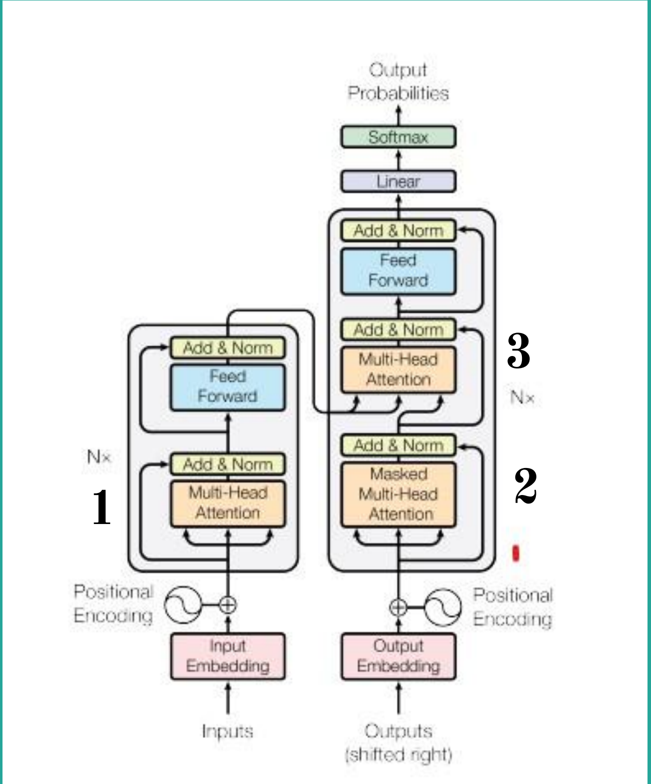


Figure 1: The Transformer - model architecture.

Advantages of Attention

Three considerations:

1. **What's the total computational complexity per layer?** Less complexity, more speed
2. **How many operations can be parallelized?** Parallelization increases speed
3. **What's the path length signals have to travel in the network?** Shorter path lengths mean the model learns long-range dependencies better

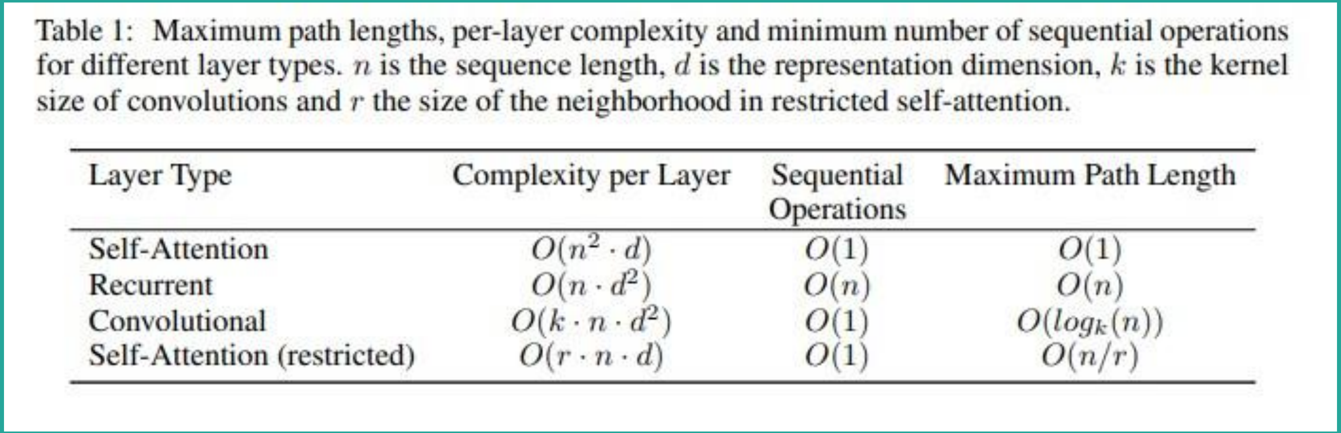


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

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

CONCLUSION

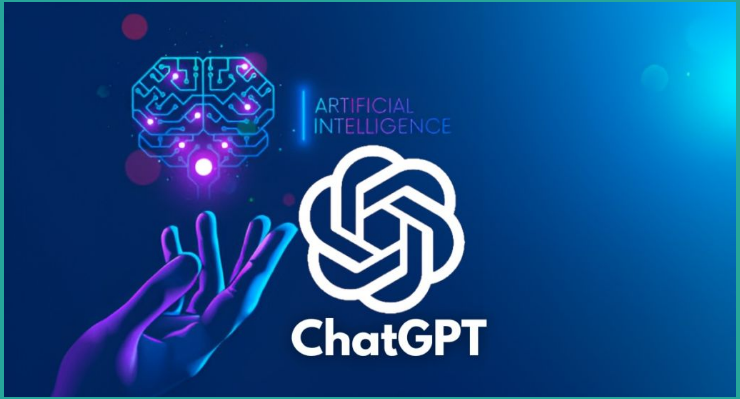
- Takeaways
- Broader Applications
- Colab notebook

Takeaways

Why does maximizing parallelization and minimizing computational complexity matter?

Allows for training on larger datasets that it was infeasible to train sequential architectures on (Wikipedia, Common Crawl)

"For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers. On both WMT 2014 English-to-German and WMT 2014 English-to-French translation tasks, we achieve a new state of the art. In the former task our best model outperforms even all previously reported ensembles."



Broader Applications of Transformers

"Our survey encompasses the identification of the top five application domains for transformer-based models, namely: NLP, Computer Vision, Multi-Modality, Audio and Speech Processing, and Signal Processing." - 'A Comprehensive Survey on Applications of Transformers for Deep Learning Tasks', Islam et al., June 2023

You can play around with a Transformer too!

00-langchain-intro.ipynb - Colaboratory