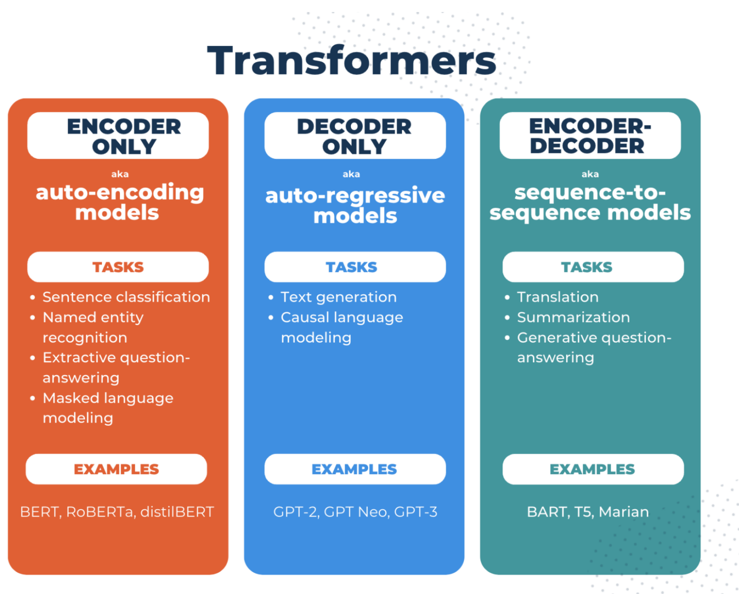
Transformers – Part 2

Summary of Chapter 10 from
Speech and Language Processing,
Jurafsky and Martin, Feb. 3, 2024 draft
Michael Wollowski

Transformers



Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

Review

Query: As *the current focus of attention* when being compared to all of the other preceding inputs.
Key: In its role as *a preceding input* being compared to the current focus of attention.
Value: As a value used to compute the output for the current focus of attention.

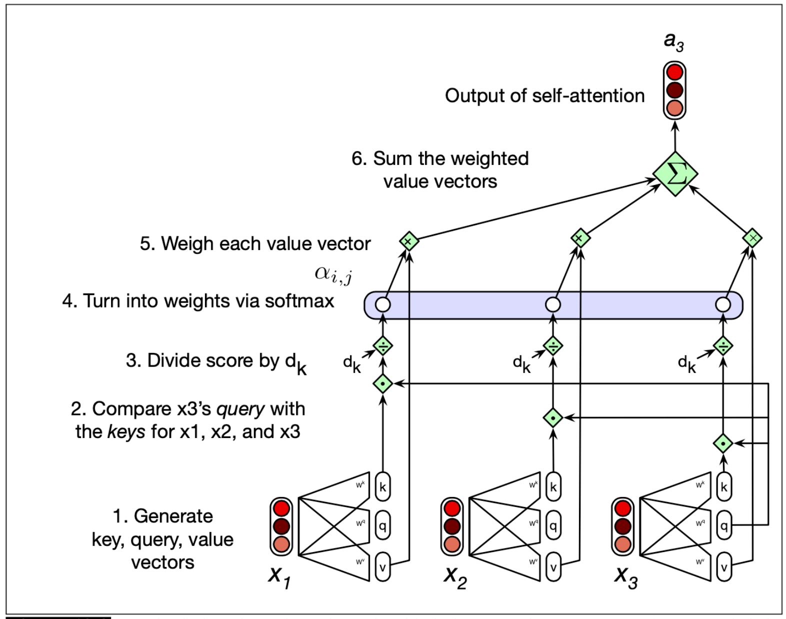
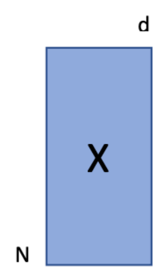


Image source: Speech and Language Processing, Jurafsky and Martin, Feb. 3, 2024 draft

Parallelizing Self-Attention



- So far, we computed a single output at a single time step *i*.
- Each output, y_i, is computed independently.
- The calculation can be parallelized.
- We pack the input embeddings of the *N* tokens of the input sequence into a **single** matrix $\mathbf{X} \in R^{N \times d}$
- Each row of **X** is the embedding of **one** token of the input.
- Transformers for large language models can have an input length *N* = 1024, 2048, or 4096 tokens.
- **X** has between 1K and 4K rows, each of the dimensionality of the embedding *d*.

Parallelizing Self-Attention

- We multiply **X** by the key, query, and value matrices.
- They all are of size *d* x *d*.
- This produces matrices $Q \in R^{N \times d}$, $K \in R^{N \times d}$, and $V \in R^{N \times d}$
- And the query, key, and value vectors:

  $Q=XW^Q$; $K=XW^K$; $V=XW^V$
- Given these matrices we can compute all the requisite query-key comparisons simultaneously by multiplying Q and K^T in a single matrix multiplication.
- The product is of shape $N \times N$.

Masking out the Future

- The self-attention computation has a problem: the calculation in QK^T results in a score for each query value to every key value, *including those that follow the query*.
- This is inappropriate in the setting of language modeling: guessing the next word is pretty simple if you already know it!
- Hence, the upper-triangle portion of the comparisons matrix set to $-\infty$.
- Softmax will turn them into zeros

	q1•k1	−∞	−∞	−∞	−∞
	q2•k1	q2•k2	−∞	−∞	−∞
N	q3•k1	q3•k2	q3•k3	−∞	−∞
	q4•k1	q4•k2	q4•k3	q4•k4	−∞
	q5•k1	q5•k2	q5•k3	q5•k4	q5•k5
			N		

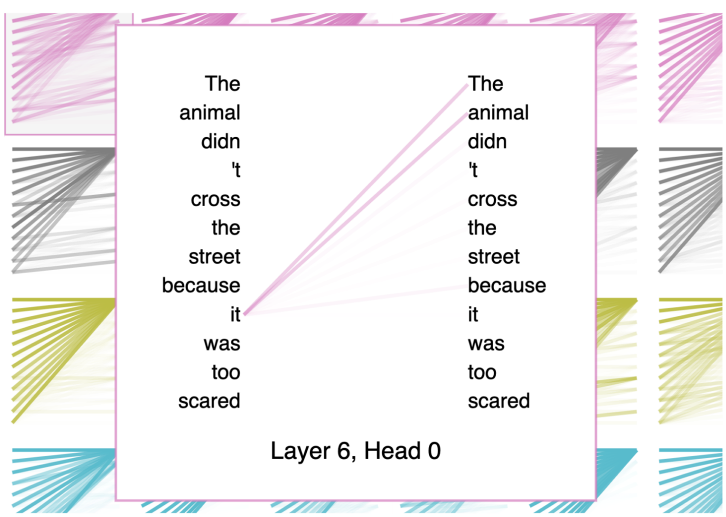
Image source: Speech and Language Processing, Jursafky and Martin, Jan. 12, 2022 draft

Multihead Attention

- Different words in a sentence can relate to each other in many different ways simultaneously.
- For example, distinct syntactic, semantic, and discourse relationships can hold between verbs and their arguments in a sentence.
- It would be difficult for a single self-attention model to learn to capture all of the different kinds of parallel relations among its inputs.
- Hence, transformers have more than one attention head.
- They are computed in parallel at the same depth in a model, each with its own set of parameters.
- This is similar to filters in CNNs.
- By using distinct sets of parameters, each head can learn different aspects of the relationships among inputs.

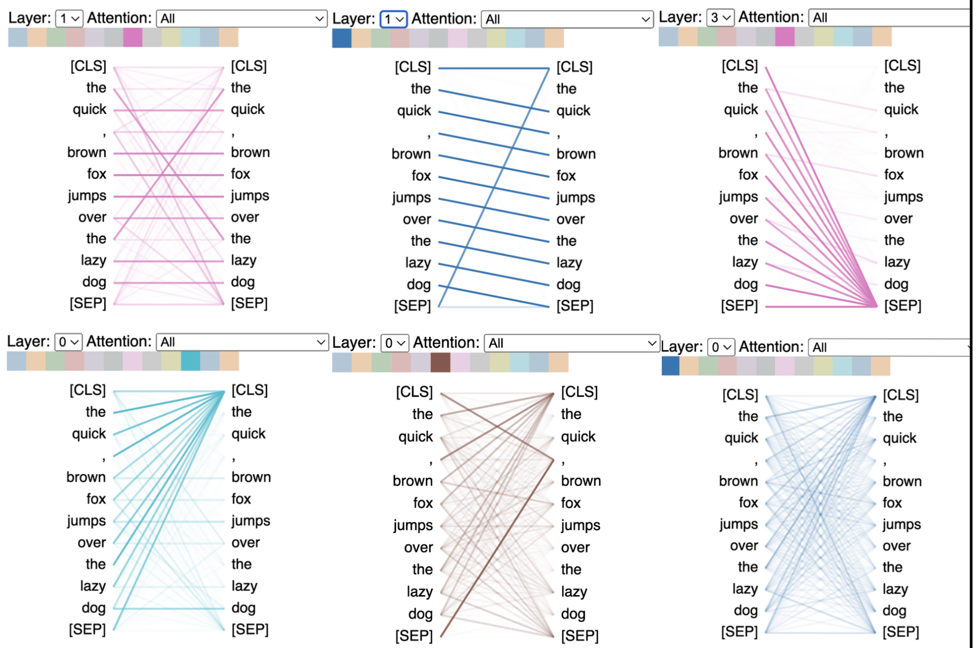
Attention

GPT-2 associated “it” with “the animal”



Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

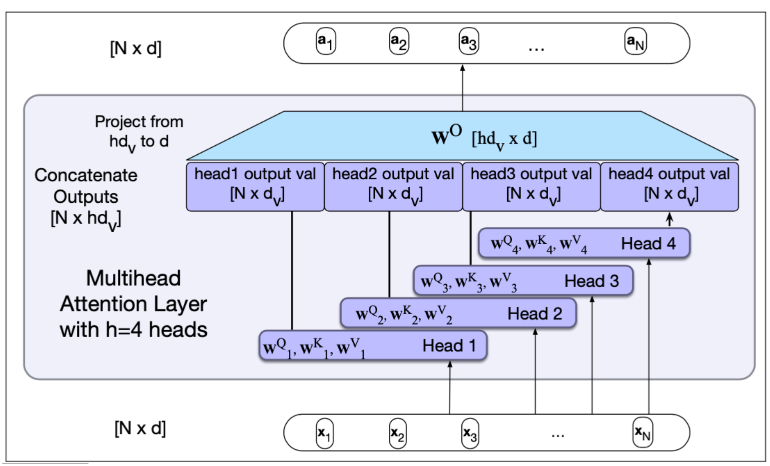
BertViz shows that attention captures various patterns in language, including positional patterns, delimiter patterns, and bag-of-words.



Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

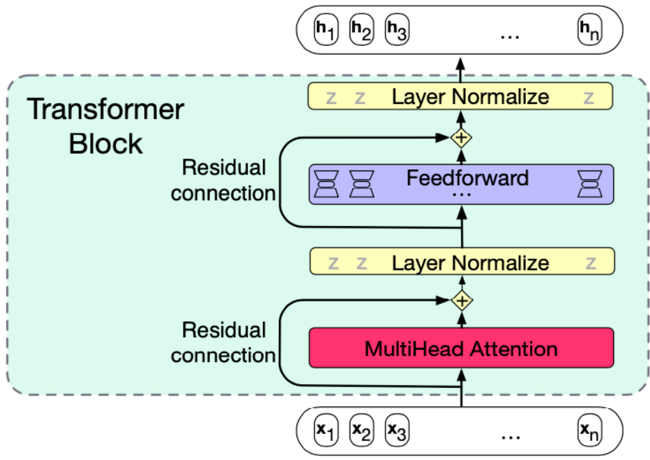
Multihead Attention

- Four self-attention heads.
- Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices.
- The outputs from each of the layers are concatenated.
- They are then projected to *d*.
- Thus producing an output of the same size as the input.
- The attention can be followed by layer norm and feedforward
- Layers can be stacked.

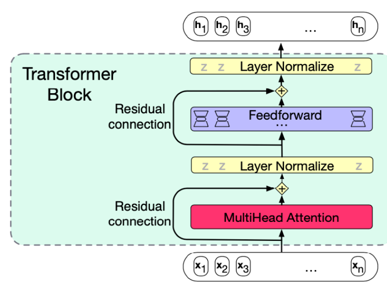


Transformer Blocks

- The self-attention calculation lies at the core of what is called a **transformer block**.
- In addition to the self-attention layer, it includes additional feedforward layers, residual connections, and normalizing layers.

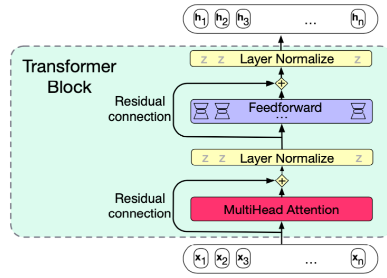


Transformer Blocks



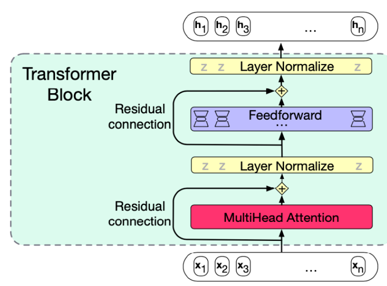
- **Feedforward layer**: It contains *N* position-wise networks, one at each position.
- Each is a fully-connected 2-layer network, i.e., one hidden layer, two weight matrices.
- The weights are the same for each position, but the parameters are different from layer to layer.

Transformer Blocks



- **Residual connections**: They pass information from a lower layer to a higher layer without going through the intermediate layer.
- **Layer normalization (Layer norm)**. Summed vectors are normalized.
- It is used to improve training performance in deep neural networks.
- It keeps the values of a hidden layer in a range that facilitates gradient-based training.
- Layer norm is a variation of the standard score, or z-score, from statistics applied to a single vector in a hidden layer.

Transformer Blocks



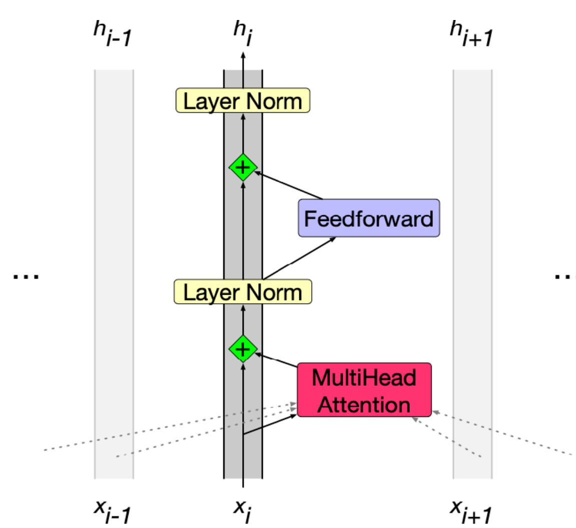
- The input to layer norm is a single vector, for a particular token position *i*, and the output is that vector normalized.
- The first step in layer normalization is to calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized.
- Given these values, the vector components are normalized by subtracting the mean from each and dividing by the standard deviation.
- The result of this computation is a new vector with zero mean and a standard deviation of one.

Transformer Block: Layer Normalization

- Typical: z-score $$z = \frac{x - \mu}{\sigma}$$
- Z = standard score
- X = observed value
- μ = mean of sample
- σ = standard deviation of the sample

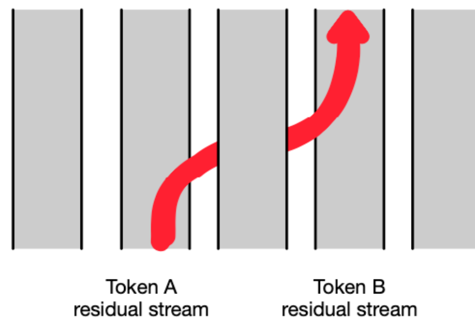
Residual Stream View of the Transformer

- An alternate view of a transformer is to trace the processing of an individual token vector x_i.
- The diagram shows the residual stream for token x_i
- The output of the feedforward and multi-head attention layers are added in, and processed by layer norm, to produce the output of this block, h_i.
- Of all the components, only the Multi Head Attention component reads information from the other residual streams in the context.



Moving Information

- The attention head can move information from token A's residual stream into token B's residual stream.

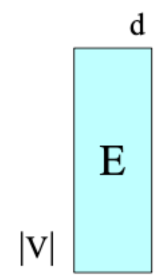


Embeddings and Such

- A token embedding is a vector of dimension *d* that will be the initial representation for the input token.
- As the vector is passed up through the transformer layers in the residual stream, this embedding representation will change and grow, incorporating context and playing a different role depending on the kind of language model we are building

Embeddings and Such

- Given an input token string like *"thanks for all the"* the transformer architecture first convert the tokens into vocabulary indices.
- Let V be the vocabulary and |V| be the size of V.
- Let E be the embedding matrix.
- The representation of "*thanks for all the*" might be
  w = [5, 4000, 10532, 2224].
- We treat the values of w as indices to corresponding rows from E, (row 5, row 4000, row 10532, row 2224).

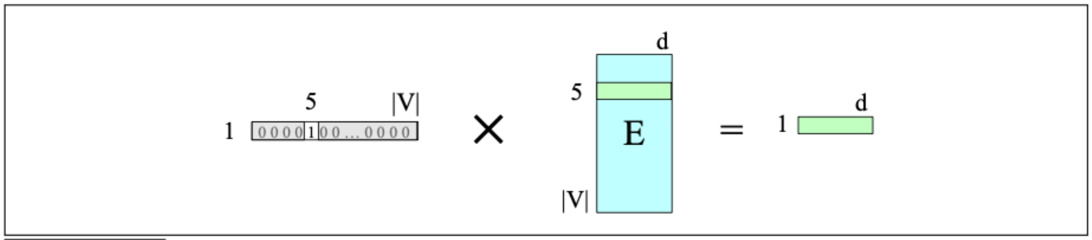


One hot-hot-hot vector

- In a *one-hot* vector all the elements are 0 except for one, the element whose dimension is the word's index in the vocabulary.
- If the word "thanks" has index 5 in the vocabulary, then x_5 =1, and all other x_i =0

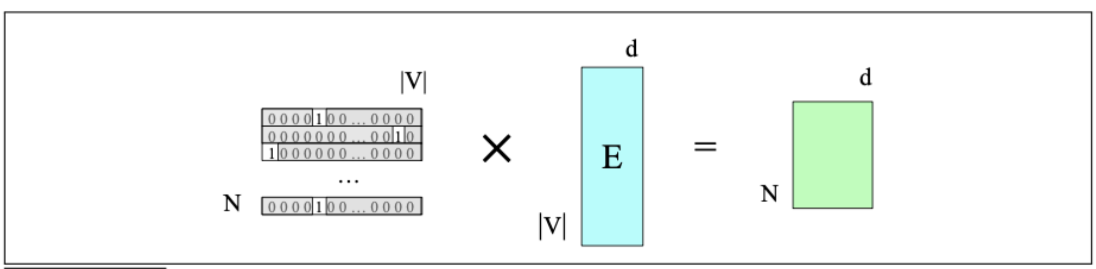
```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...   ... |V|
```

Selecting the token embedding



Multiplying **E** by a one-hot vector that has only one non-zero element x_i = 1 simply selects the relevant row vector for word *i*, resulting in the embedding for word *i*.

Storing all of the N input tokens

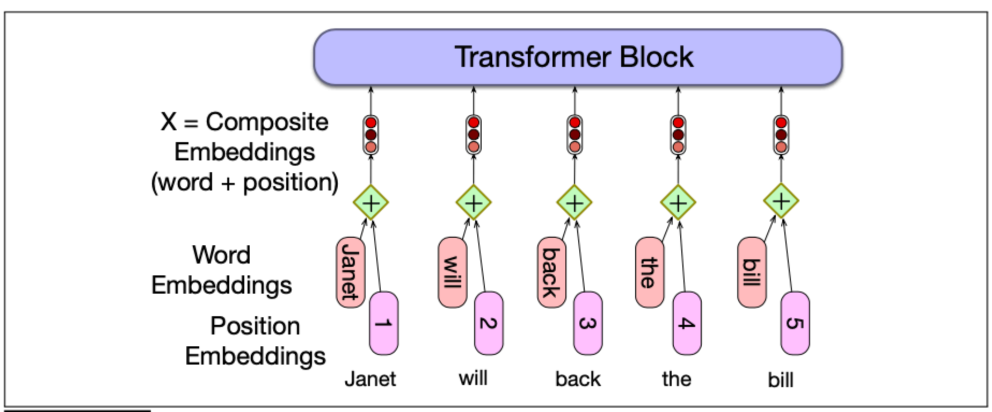


To represent the entire token sequence, we multiply all **N** one-hot vectors with **E**.

Positions

- While the order in which the N tokens are inserted represents word order, this is not sufficient.
- Recall that attention heads can move tokens around.
- We wish to associate with each word the order in which it appeared in the text.
- As such, we combine these token embeddings with positional embeddings specific to each position in an input sequence.

Positions



Combining word embeddings with positions.

Positions

- The positions are absolute.
- However, we do not simply use integers.
- Instead, we start with randomly initialized embeddings corresponding to each possible input position up to some maximum length.
- For example, just as we have an embedding for the word *fish*, we will have an embedding for the position 3.

Positions

- As with word embeddings, these positional embeddings are learned along with other parameters during training.
- We can store them in a matrix E_{pos} of shape [1 x N].
- The individual token and position embeddings are both of size [1×d], so their sum is also [1 × d]
- To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding.
- This new embedding serves as the input for further processing.