Transformer architectures

INFO 4940: Advanced NLP for Humanities Research

Timeline of neural methods in NLP

1958	The perceptron (not NLP, not a deep network)
1982	Modern SGD by backprop introduced (not NLP)
1990s	Modern kernelized SVMs, token n-grams
2001+	Deep MLP (FFNNs, RNNs) for language modeling
2013	Word2vec (static embeddings)
2014	Sequence-to-sequence models (enc-dec, translation)
2015	Attention (efficient representation of context)
2017	Transformer (encoder-decoder, drop RNN)
2018	BERT (encoder), GPT (usw., decoder)

The main idea

In all of these models (sequence-to-sequence and newer), the high-level process is:

1. Encode your inputs (words) to embedding vectors
2. Use embeddings as inputs to a multilayer encoding that captures contextual information
3. Use contextual encodings for a downstream task, often to generate new text

Training

When training, we learn model weights that best predict the next word in a sequence, the next sentence (from selected candidates), or masked words.

This creates a **pretrained** model. These models are expensive. You will not create them, but you will use them.

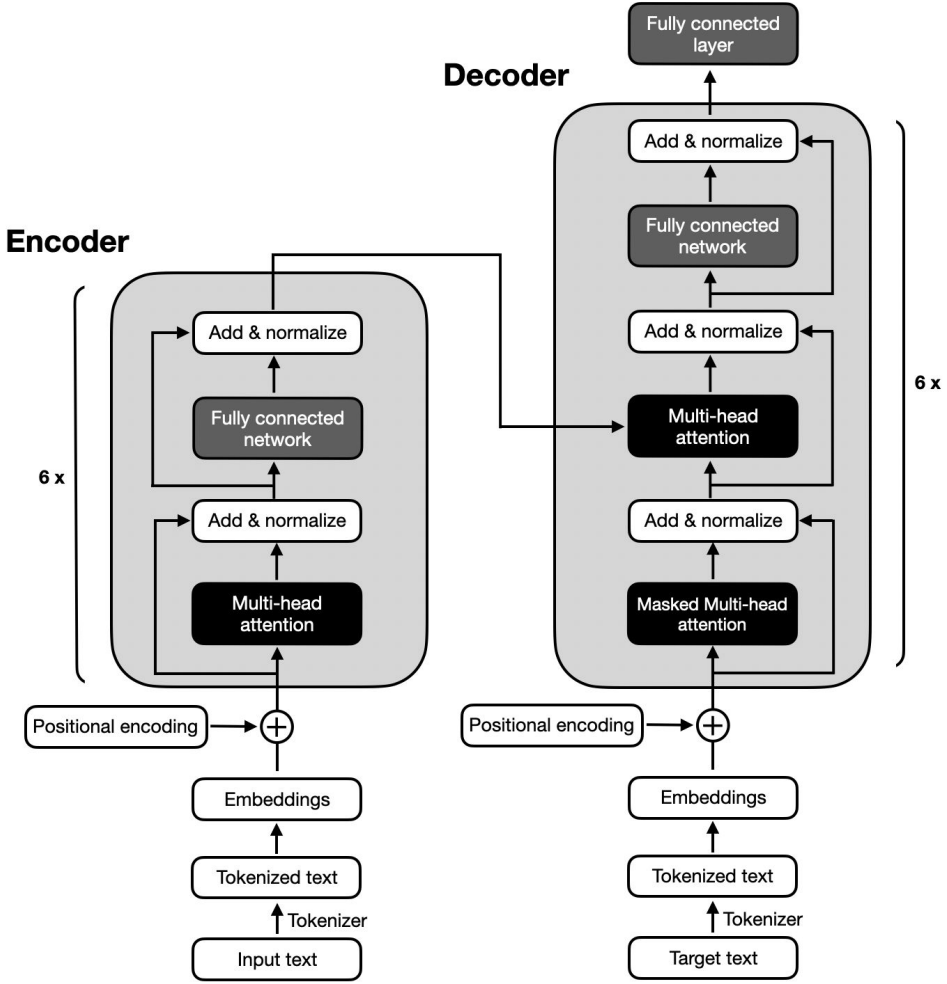
The learned weights are what we use at inference time to construct contextual representations, predict next words, and as the basis for fine-tuning for specific tasks (like classification).

Transformer models

Encoder-decoder architecture

Initial input embeddings are from static embeddings like word2vec.

[Figure from “Attention is all you need” (2017)]

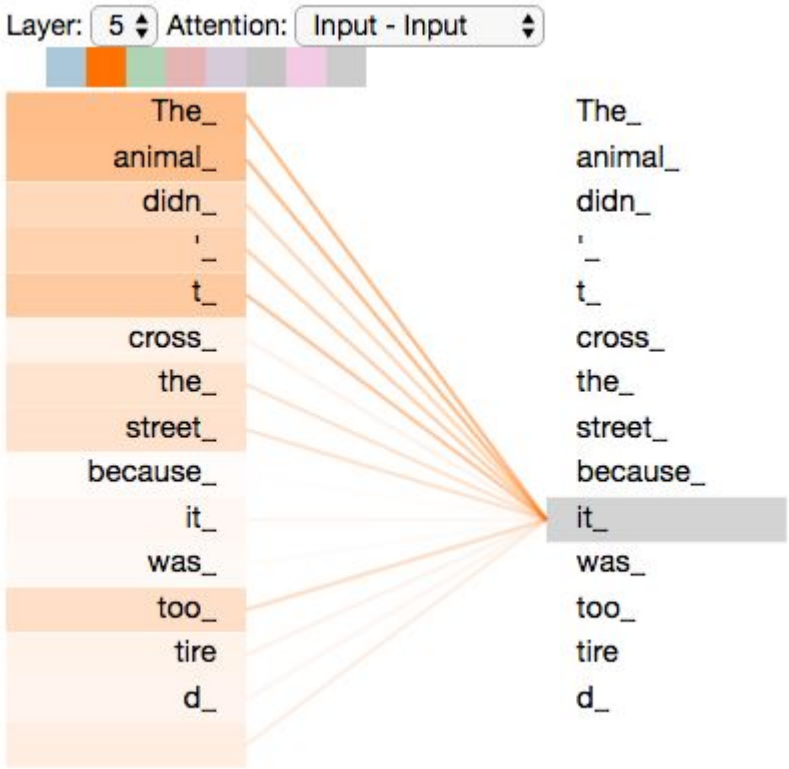


(Self) Attention

Attention

To what does **'it'** refer in this sentence?

"The animal didn't cross the street because it was too tired."



[Figure from Alammar, Illustrated Transformer]

Calculating attention

Queries, keys, and values. At (pre)training time, we learn these three weight matrices.

For each token, multiply our input embedding by the query matrix to produce a query vector.

Multiply each other token vector in our sequence (multiplying by the key matrix) to produce key vectors.

Multiplying query vectors by key vectors gives us token-wise attention scores. We apply softmax to the scores, then multiply by the values matrix to produce the output of the attention layer.

Big picture

The attention layer gives us a representation of connections between words in context.

The layer gives us a representation of the whole document. This representation is aware of how the document's pieces work together to determine its meaning.

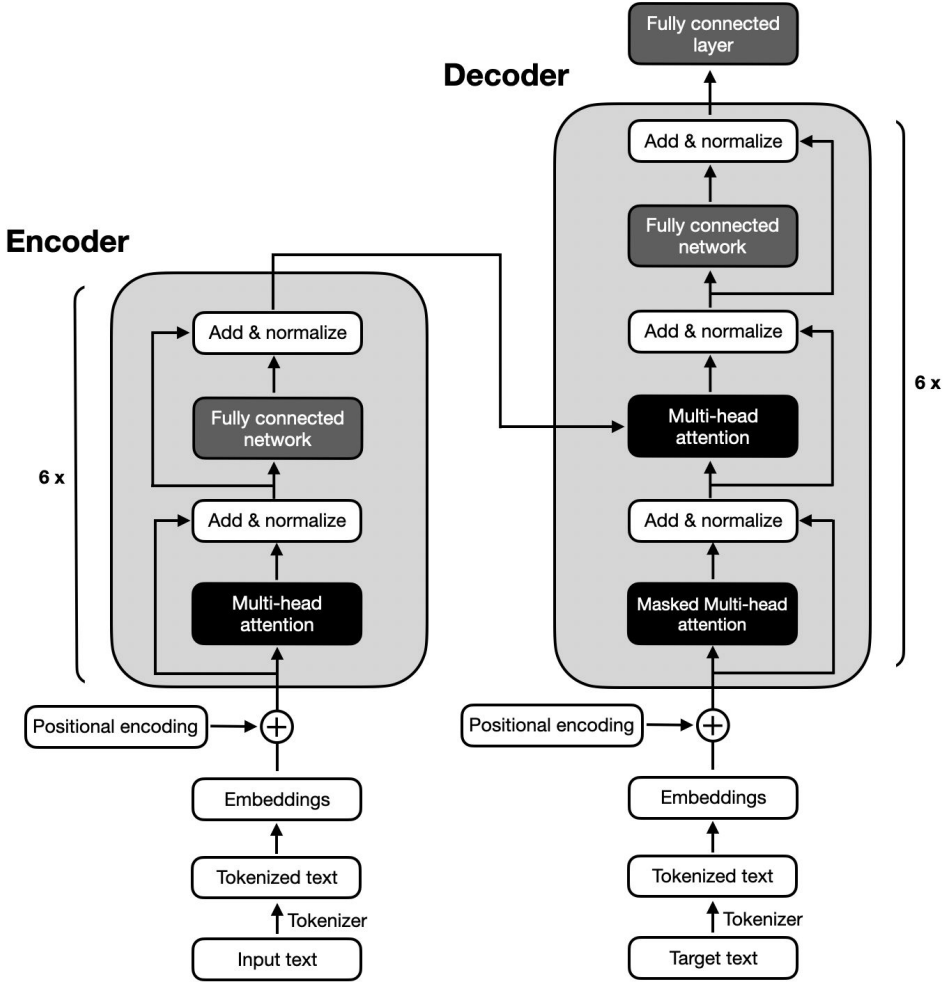
Multiple layers capture different aspects of connection and meaning (syntax, semantics, topical similarity, etc.)

Attention alone is more efficient than LSTMs and other RNNs

Back to the transformer

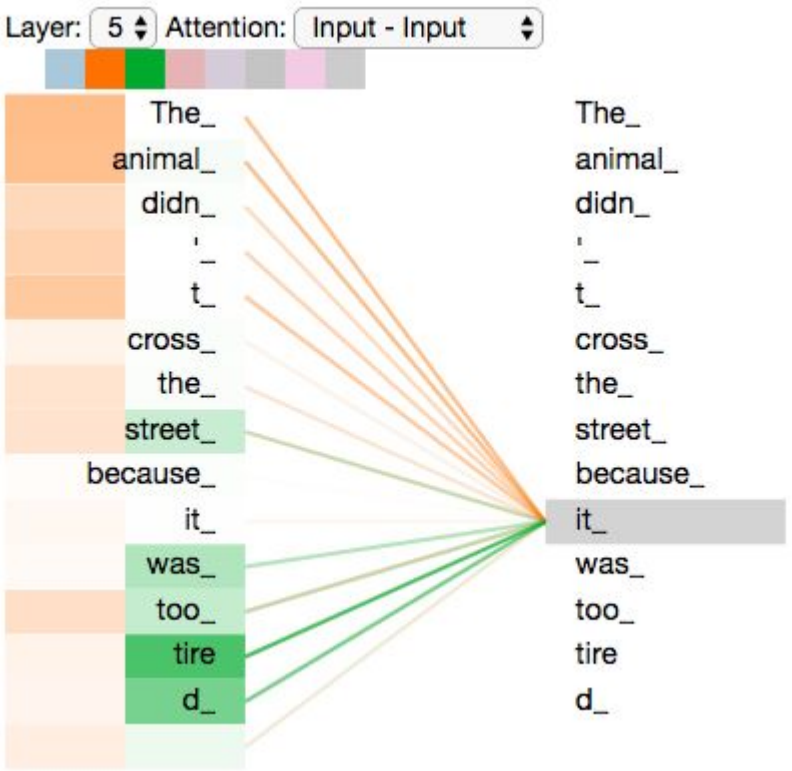
Multi-head attention

“Multi-head attention” means we calculate multiple different attention representations using different learned query, key, and weight matrices …



Two attention heads

… which allows us to capture different attention connections.



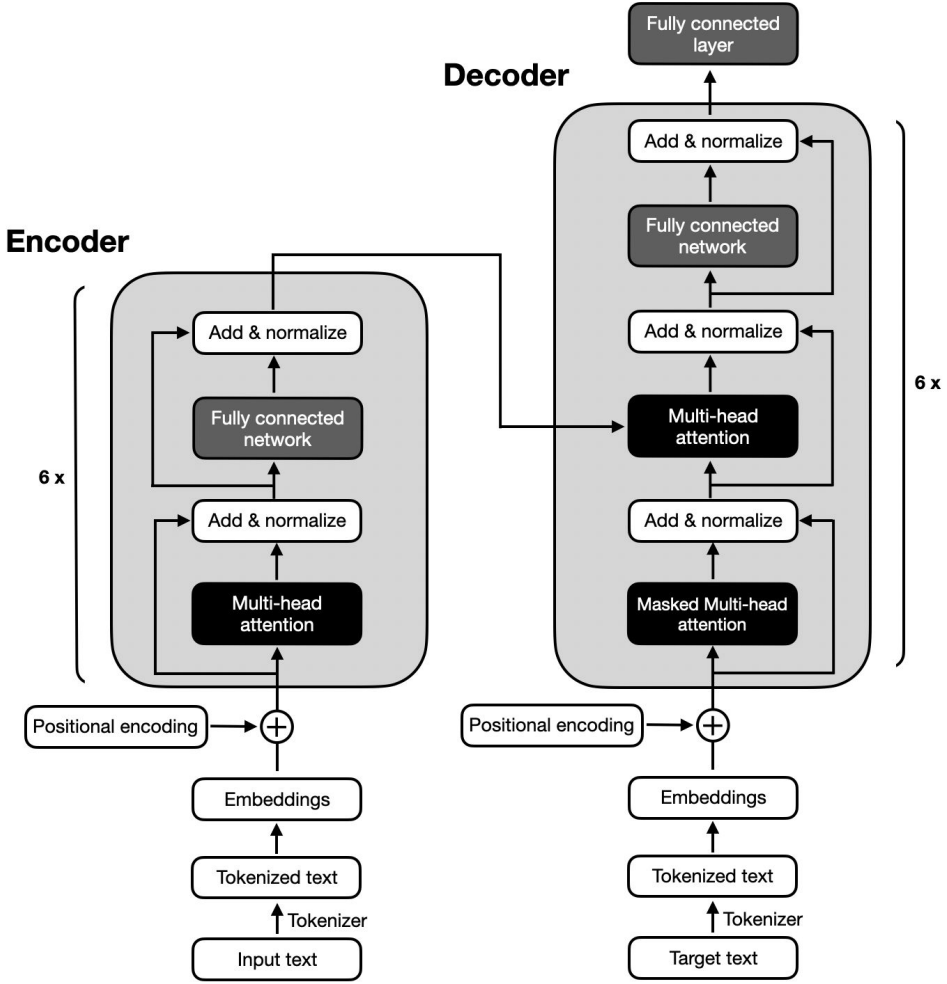
[Figure from Alammar, Illustrated Transformer]

The decoder stack

The final output of the encoder stack is used as input to each decoder. Each decoder *also* receives input from the previous layer in the stack.

The decoder works like the encoder, except that the final, fully connected output layer (logits + softmax) is used to predict the next word.

Initial “target text” is empty; then it’s previous output text



Aside: softmax

The softmax function is the multidimensional version of the logistic function. It converts a vector of *K* real numbers into a probability distribution over *K* possible outcomes.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

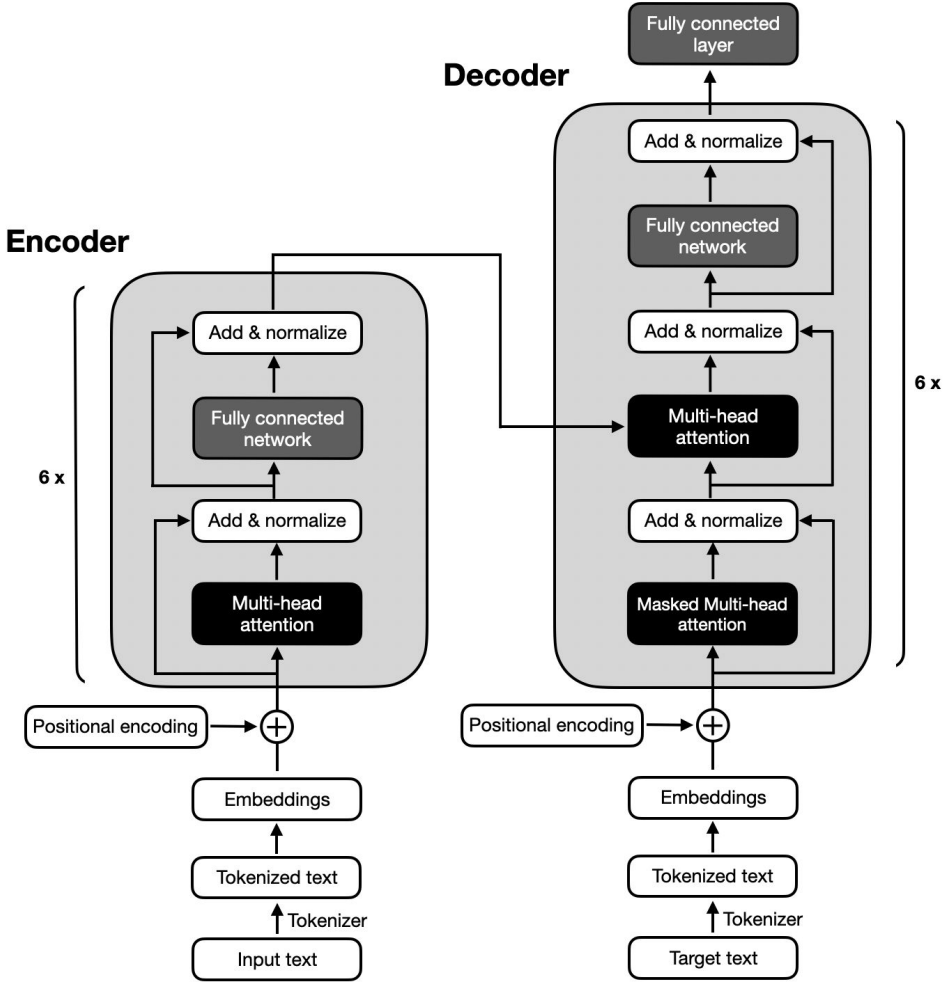
```
softmax([-1,0,3,5]) = [0.002, 0.006, 0.118, 0.873]
```

The decoder stack

The final output of the encoder stack is used as input to each decoder. Each decoder *also* receives input from the previous layer in the stack.

The decoder works like the encoder, except that the final, fully connected output layer (logits + softmax) is used to predict the next word.

Initial “target text” is empty; then it’s previous output text



BERT

Bidirectional Encoder Representations from Transformers

BERT *drops* the decoder stack. It does not generate text via next-word predictions. It is not a generative model.

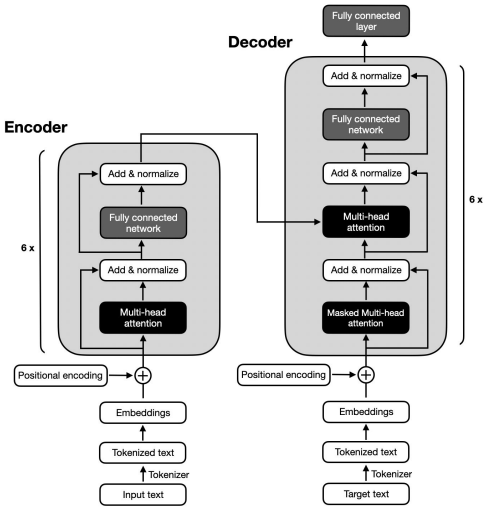
BERT *adds* **bidirectional** encoding. Unlike the transformer, it represents encoding context forward *and backward* from each token.

Bidirectional encoding requires masked self-attention to avoid revealing the target word itself.

Bidirectional Encoder Representations from Transformers

BERT outputs are (very good) contextual embeddings, which you can then use as inputs to another task (often classification via a linear layer and softmax).

Aside: “encoding” and “decoding” in non-encoder-decoder models



Models that lack either an encoder or a decoder obviously *do* still encode things and decode things. They’re not just big stacks of tensors.

But encoder-only models do not use autoregressive generation. That is, they do not learn from their own outputs.

Similarly, decoder-only models lack cross-attention to input encodings.

Finetuning

Finetuning a pretrained model is a method to adapt it to a specific task and dataset.

Finetuning typically involves using embeddings from a pretrained model, adding a new output layer, and training the system on a specific objective.

The finetuned system learns weights for the new layer *and* updates the weights of the pretrained model.

Finetuning

There is evidence to suggest that finetuning typically makes the largest updates to the last (highest) layers of the pretrained model.

Finetuning is much less computationally costly than pretraining. Good performance is often achieved with just a few epochs of additional training on much smaller datasets compared to those used for pretraining.

What’s ahead

Decoders!

Most state-of-the-art LLMs are now decoder(-only) systems.

Decoder systems prioritize general language understanding at the expense of task-specific training.

With sufficiently large models, the advantages are low downstream training cost and high task flexibility

[Figure via Raschka]