

Transformer: Intuition

We try to briefly explain what each the "moving parts" of the Encoder-Decoder style Transformer is doing.

At the highest level: we have the Encoder and the Decoder.

In the Encoder-Decoder architecture

- the Encoder completes before the Decoder starts

Encoder

The role of the Encoder is

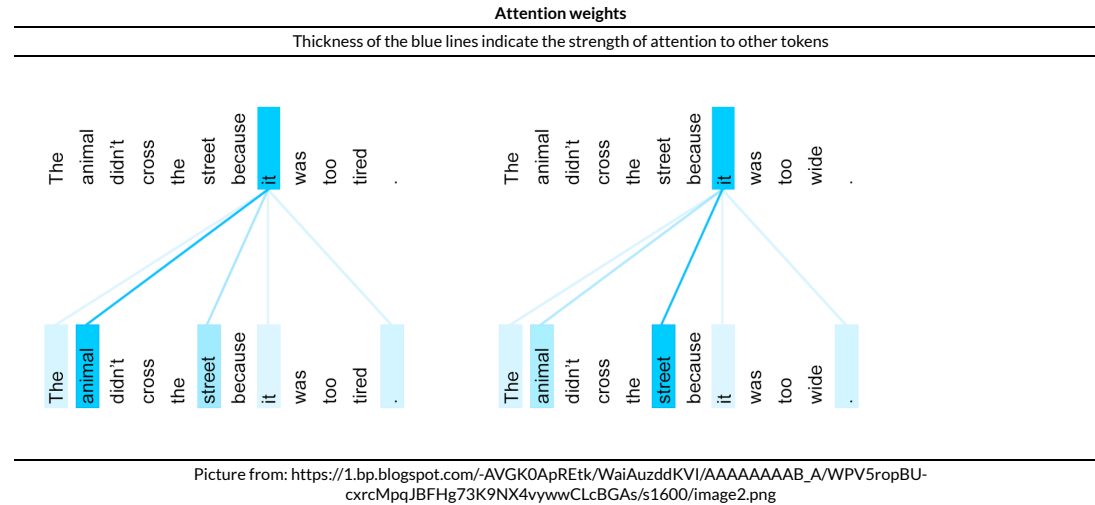
- to create a Context Sensitive Representation

$$\bar{\mathbf{h}}_{(1:\bar{T})}$$

- of each of the Encoder's input tokens

$$\mathbf{x}_{(1:\bar{T})}$$

By making the meaning dependent on the full context, we can disambiguate the meaning of the word "it"



It accomplishes this by the *direct function* approach

- unlike an RNN, it does not process each input token $\mathbf{x}_{(t)}$ sequentially
- it computes $\bar{\mathbf{h}}_{(t)}$ as a function of the entire input $\mathbf{x}_{(1:\bar{T})}$

Encoder Self-Attention is used in the direct function.

Decoder

The Decoder works in *auto-regressive* mode

- predicts one output token at a time
- the current output $\hat{\mathbf{y}}_{(t)}$ token is appended to the input for the next position
 - so the input at time step t is

$$\hat{\mathbf{y}}_{(1 \dots t-1)}$$

It has two inputs at step t

- the previously-generated output tokens t is

$$\hat{\mathbf{y}}_{(1 \dots t-1)}$$

- the Encoder output

$$\bar{\mathbf{h}}_{(1:\bar{T})}$$

Self-attention is used on $\hat{\mathbf{y}}_{(1 \dots t-1)}$

Cross-Attention is used on $\bar{\mathbf{h}}_{(1:\bar{T})}$

At step t , the Decoder

- uses Self-Attention on $\hat{\mathbf{y}}_{(1 \dots t-1)}$
- to create a *query*
- that is used to attend to $\bar{\mathbf{h}}_{(1:\bar{T})}$

We can think of this use of Self-Attention

- as being a replacement for the "latent" state of an RNN
 - rather than using the latent state to record
 - what has already been done
 - what is the next step to perform
 - Self-Attention allows direct access to what has already been done:
 $\hat{\mathbf{y}}_{(1..t-1)}$

The query is used in Cross-Attention

- to attend to the Context Sensitive Representation of the input sequence \mathbf{x}

Whatever is returned by Cross-Attention

- is input into the Feed Forward Network (FFN)

Think of the FFN

- as a repository of "world knowledge" accumulated by processing the training data
- "facts"

The FFN produces an output

- which is processed by a Classifier (Linear layer)
- to produce a token in the vocabulary of tokens

That is

- if the vocabulary has $|V|$ tokens
- the Classifier produces a probability distribution vector \mathbf{p} of length $|V|$)
 - such that \mathbf{p}_j is the probability that the output token should be V_j

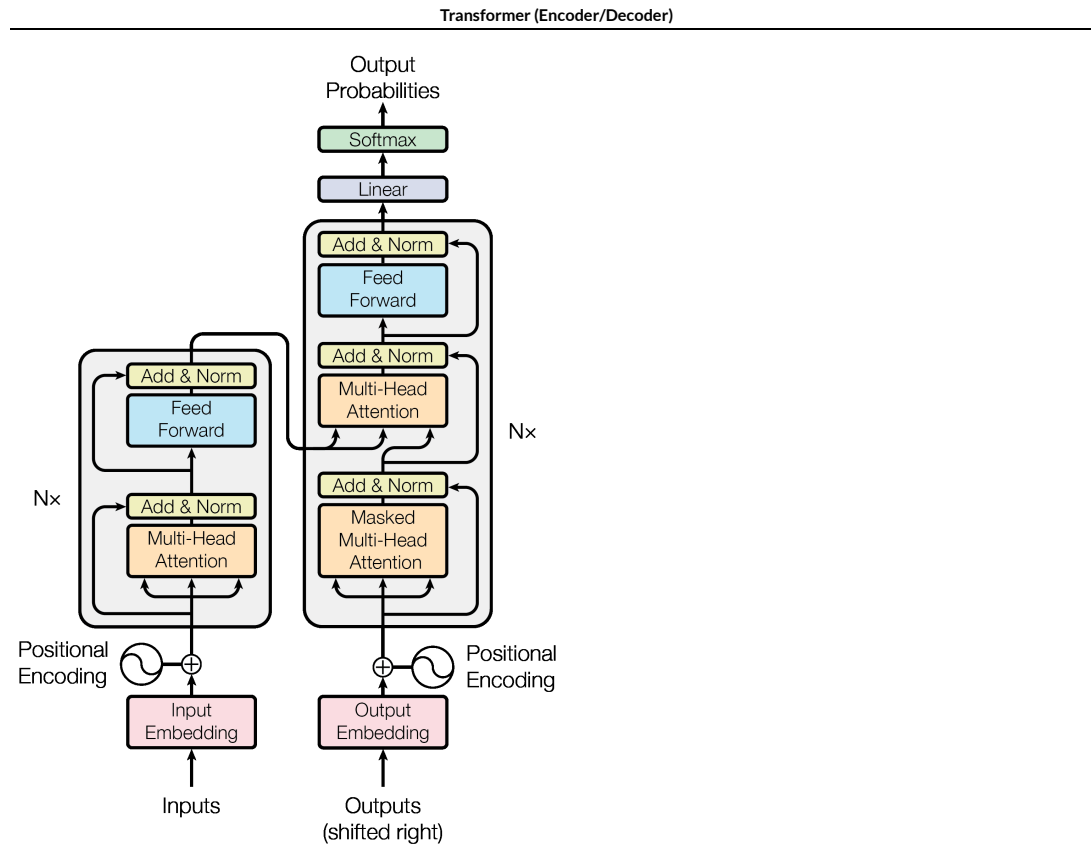
The exact mechanics of this multi-step process

- are controlled by the weights
- that are learned during training

General

Here is the detailed architecture of the Encoder-Decoder Transformer.

We will review each of the pieces.



Each of the paths in the Transformer is a vector of length d_{model}

- sometimes just referred to as d

Having a common length simplifies the architecture

- can stack Transformer blocks (since input and output are same size)
- Self-Attention and Cross-Attention:
 - map a query of size d to an output of size d
- Needed for the Residual Connection (Add and Norm)
 - adding the input of Attention to the output of Attention
 - need to be same length

Residual connections

- [Residual connections from Intro course \(RNN Residual Networks.ipynb\)](#).

Embedding

Words (really: tokens) are *categorical* variables.

Categorical variables are usually encoded as long vectors via One Hot Encoding (OHE)

- very long: number of distinct elements in class
 - e.g., number of words in vocabulary
- *sparse*: only a single non-zero element in the vector
- the similarity (e.g., dot product) of two related words (e.g., "cat", "cats") is zero !
 - same as for two unrelated words (e.g., "cat", "car")

An *Embedding* is a *short* and *dense* vector representation of words (tokens).

In addition to being shorter (and dense: many non-zero elements possible) their construction results in

- the similarity of embeddings for two related words being *non-zero*

This makes Embeddings much more valuable for NLP.

The *Embedding Layer* converts the OHE representation to an Embedding.

See the [module from the Intro course \(NLP Embeddings.ipynb\)](#) for details.

Positional Encoding

The Transformer input is a *sequence*

- there is a total ordering between elements based on absolute position

The Transformer needs to be able to discern

- at least: the *relative* ordering of two elements in different positions in the sequence

The *Positional Encoding* layer

- adds a vector that encodes position
- to the Embedding
- such that the Transformer has a representation with both meaning and positions

This is much more involved than simply using an integer to encode the position.

The fundamental operation of a Neural Network is matrix multiplication

- the positional encoding needs to be preserved as it traverses the layers

The details are not trivial.

See the module on [Positional Embeddings \(Transformer_PositionalEmbedding.ipynb\)](#) if you are interested.

Feed Forward Network (FFN)

Maps the output of the Decoder-Encoder Attention into the "next output token".

- actually: it is still an embedding of the next token, rather than the true next token
 - that way: it can be appended to the already-generated output to become the Decoder input for next position

This acts as a Classifier

- mapping the input
- to a vector of logits
 - one element per possible element of the Output Vocabulary

There is some evidence that

- the parameters of the FFN are where "world knowledge" is stored
 - every "fact" learned during training

Linear

This layer is append *only* to the final block in the stacked Transformer blocks.

It acts as a typical Classifier

- "classifies" the final block's output of length d
- returning a vector
 - whose length is equal to number of elements of the Vocabulary
 - each element is a logit
 - to be converted into probability distribution over elements of the Vocabulary

Softmax

Converts the logit for each possible element of the Vocabulary

- into Probability that the element is the next Decoder Output

In [2]: `print("Done")`

Done

