2.2.1 Base Architecture

This section is unnecessary for those familiar with the Transformer architecture, but it may serve as a good reference. I have tried to provide some additional intuition and clarity on parts that confused me when reading the original paper for the first time a little more than a year ago.

The classical Transformer has two main parts: the decoder and the encoder, which have similar structures. Each takes as input a sequence of tokens, which can be thought of as matrices with one-hot encodings for each of the tokens. Those tokens are embedded with a linear transformation to make a matrix of width equal to the dimension of the model, d_{model} (a hyperparameter), and height equal to the length of the sequence. Each row in the matrix represents a token embedding, and within the decoder, those rows are slowly morphed, mostly independently, into target token embeddings. Tokens interact with one another inside the "attention blocks" but not elsewhere. Finally, the embeddings that the decoder produces are inverted, and the result is a matrix with height equal to the length of the sequence and width equal to the size of the vocabulary such that the entries in a row can be used to estimate the probability that the row corresponds to the token represented by the column.

The ultimate goal of the encoder is to produce a matrix of embeddings that distill some knowledge about context that the decoder should act upon. The point of dividing the model into an encoder and a decoder is to allow the encoder to have no mask (each token can interact with every other token) while the decoder has a mask (each token can only see the previous tokens). For example, in the translation context, the encoder can take as input the original language while the decoder can take the target language; each place in the

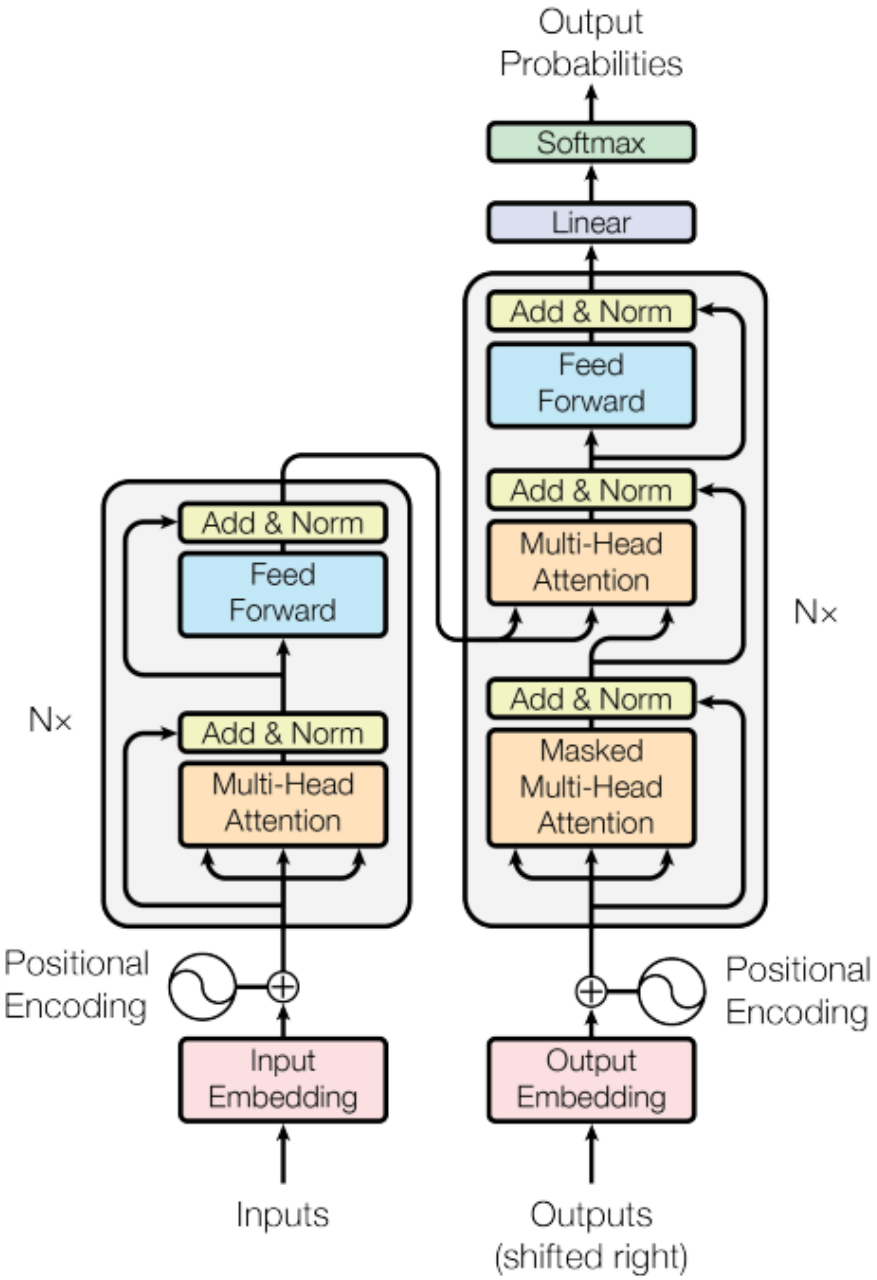

Figure 2.3: The Transformer, as seen in Vaswani et al. 2017

decoder is attempting to predict the next token, so its attention is masked, but there is no need to mask the interactions in the encoder.

It often goes unstated that none of the parameters has a shape dependent on the length of the input sequence; every place in the sequence is acted on by the same weights, though the tokens interact in the attention block. In a CNN, for example, a pattern that is learned using examples that appear in one location of an image are automatically detected everywhere in the image, due to the properties of convolutions. The analogous property of Transformers is that they can learn properties of a token or combination of tokens without seeing those tokens appear in every possible location in a sequence.

Since the model has no inherent understanding of position in a sequence, it is necessary to morph the learned token embeddings so that they carry some information about their place in the sequence. The standard way to do this is by adding a positional encoding to the token embeddings, though there are other possibilities.*

In both decoder and encoder, the embedding matrix passes through a "multi-head self-attention" block. In this block, token embeddings are altered by three learned linear projections into queries, keys, and values.† It can be thought of as "self-attention" since the three projections take the same matrix as input. Each query and key vector (that is, each row of the matrix) has its dot product taken with respect to each of the others, forming an attention matrix that is square with height and width equal to the length of the sequence (that is, its size is N^2, making this procedure quadratic in complexity). Each entry is this matrix can be thought of as an attention score between every pair of tokens. Different learned projections will result in different kinds of attention scores; one set of projections may try

*The reader should refer to the original paper[8] for details as well as experiments with alternative methods.
†Some of these projections can be the same, but they don't have to be.

to find semantically similar words, but another might seek to match nouns with their respective article. There is typically in a model always some projection that simply tries to match each token with the token immediately preceding it. This matrix of attention scores is multiplied by the values[‡] (a learned linear projection of the input embeddings). Finally, this process can be repeated across different "heads" (learned query/key/value projections), the outputs concatenated, and ultimately projected back into a matrix with width equal to d_{model}.

The output of the self-attention layer is then added to the input to the decoder layer in a residual fashion. This result is then normalized using a Layer Normalization.[9] The normalization is necessary if we want the result to have the same scale as the inputs to the residual addition. In an encoder layer, all that remains is to pass the embedding matrix through a position-wise feed-forward neural network; the result is then residually connected and normalized as in the previous step. Each layer in the encoder is stacked on top of one another, and only the final output is sent as input to the decoder.

In the decoder, following the original self-attention, the resulting embedding matrix is passed along with the encoder output into a cross-attention layer. In this layer, the previous self-attention output is projected into values while the encoder output is projected into keys and queries. The same attention operation is performed, but now the token inputs to the decoder are interacting with the outputs of the encoder rather than simply among themselves. The cross-attention output is again residually added and normalized before going through a position-wise feed-forward network, added and normalized again, and

[‡]Actually, the queries and keys are first divided by $\sqrt{d_{queries}}$ and $\sqrt{d_{keys}}$. Note also that queries and keys can exist in a space with different dimension than d_{model} (and similarly for the values). This process reduces computational complexity and promotes numerical stability.

then finally outputted to the next layer.

After the very last layer of the decoder, each token (which is a vector with d_{model} elements) is up-projected into the size of the entire vocabulary (optionally using the transpose of the original embedding matrix). A final Softmax is performed before the resulting probabilities are compared with the empirical target probabilities. The loss is calculated and backpropogated through the entire model during training. During inference, the probability distribution of tokens is sampled.

A key benefit of the Transformer is that all of the tokens in a sequence are processed in parallel. This behavior is not present in RNNs, which generally process tokens sequentially. This property has allowed Transformers to be scaled to hundreds of billions of parameters trained on hundreds of billions of tokens. It might be damning that the attention mechanism is quadratic in the length of the sequence, but the length of the sequence can be small compared to the dimension of the model at scale. Moreover, every significant operation in a Transformer is a highly-optimized matrix multiply.

2.2.2 Architectural Modifications

A recurring challenge during the training of large transformers is to keep the scale of the gradients constant throughout the model. If gradient values grow too large, training can become unstable. In the original paper, the authors used a warmup stage during training: the learning rate for SGD was first raised from a low value before being decreased again. The normalizations are intended to help, but another trick was supposed to be needed in order to remove the need for warmup and make the model more natively stable. It turns out that by moving the first Layer Norm to start of a decoder/encoder layer and the second