On the Music Transformer

Aditya Gomatam
March 24, 2021

1 Introduction

The Music Transformer or Transformer with Relative Self-Attention[1] is an autoregressive sequence model that builds on the Transformer[2] to consider the relative distances between different elements of the sequence rather than / along with their absolute positions in the sequence. This consideration was designed to model music, recognizing that music relies heavily on repetition to construct meaning and maintain long-term structure. The Music Transformer, in my estimation, appears to be the first neural network capable of reasonably capturing any sort of meaning of a piece. Google achieved extremely compelling results that demonstrate this capability of the Transformer architecture with their Piano Transformer. This essay presents my thoughts on how the Self-Attention and Relative Self-Attention algorithms work, in order to be so effective.

2 Self-Attention

Self-Attention is exactly what it sounds like - having a sequence pay attention to itself. The Transformer achieves this by comparing each element in the sequence with the entire sequence so as to determine the amount of "attention" each element pays to every other element, as a method of capturing internal connections and thus the overall structure of the sequence. As such, unlike convolutional or recurrent neural networks, the Transformer is able to select which portions of the input sequence to "attend to", in order to best predict future tokens.

So, how does it work? Vaswani et al. 2017 describe an attention function as, "mapping a query and a set of key-value pairs to an output"[2]. I did not learn much from that. The way I see it, attention is easier to understand by exploring the math behind it.

2.1 The Idea

Suppose we want to compute Self-Attention for a single sequence of L events in a vocabulary. At input to the Transformer, we embed the sequence[1] to make it of shape (L, d_{model}), where d_{model} is the embedding size. For the attention calculation and for residual connections, most tensors handled by the Transformer will have a dimension of length d_{model}, which is why it is so named. Next, (after adding positional encoding and scaling), we pass the embedded input X through the stack of Transformer layers. Each Transformer layer, whether in the Encoder or Decoder, consists fundamentally of an Attention block followed by a fully-connected feedforward network (FFN), with a LayerNorm either preceding the Attention block (pre-LayerNorm) or succeeding the FFN block (post-LayerNorm), and with residual connections in between.

Now, as aforementioned, we want to determine the attention each element in the sequence pays to *every* other element, in order to capture information about the internal connections and hence overall structure of the sequence. Suppose we could calculate the attention an element of the sequence pays to one other as a positive scalar "attention weight" whose magnitude represents the amount of "attention" paid. Then, we can construct a matrix M of shape (L, L) in which each entry M_{ij} is the just mentioned numerical attention X_i pays to X_j. However, we cannot use this matrix by itself - it holds information only about the attention relations between exactly two elements in the sequence in each entry, whereas we want to extract the attention each element pays to the sequence as a whole.

The matrix product MX would in principle achieve this goal, as each $(MX)_i$ would be a weighted sum over X with the weights as the attentions X_i pays to X_j:

$$(MX)_i = \sum_j M_{ij} X_j$$

Those X_j representing elements important to X_i will correspond to high M_{ij} and will thus contribute more to the weighted sum, whereas those X_j representing elements unimportant to X_i will correspond to low M_{ij} and will thus contribute close to nothing. And so, $(MX)_i$ will be a vector in the embedding space pointing closest to the elements of X that X_i attends to the most. The weighted sum will therefore capture the connections between X_i and the entire sequence, as well as store this information in a vector of the same shape as X ($MX \sim (L, L) \times (L, d_{model}) \to (L, d_{model})$). That is, MX is a new representation of input sequence whose every element contains information about the internal structure of the entire sequence. Furthermore, as it preserves the shape of X, outputting this representation allows us to chain Attention blocks (with FFNs and LayerNorms in between) to extract ever purer such encapsulations of the internal structure of the sequence.

Well, it's not quite so simple. But this is the idea.

[1] Create learnable, variable vector representations for each event in the vocabulary, then replace each element in the sequence with the corresponding embedding.

2.2 Scaled Dot-Product Self-Attention

In computing Self-Attention in an actual Transformer, we don't directly multiply the attentions matrix M by X. Instead we multiply it by the *values* tensor, which is another representation of the input sequence obtained by a simple fully-connected layer on X:

$$V = XW^V + B^V$$

The parameter tensors W^V and B^V are learned by gradient descent. It is convenient to preserve the shape of X, which is why W^V is usually shaped (d_{model}, d_{model}), and B^V is usually shaped $(d_{model},)$. However, one can imagine that if these parameter tensors were properly optimized by gradient descent, V would contain refined features of X that allow the attention calculation to more accurately determine the internal connections within the sequence.

But how do we calculate the attentions weights in M? There have been many answers to this question. In fact, while weighted summation over the elements of X to determine which elements of the sequence a particular element is most connected to underlies many formulations of attention, the calculation of M varies significantly.

A few deep learning researchers, such as the developers of the Transformer, interpreted the selection of the values to which to attend as similar to retrieving those values from (i.e., querying) a database. Consequently, the Transformer attention equation can be interpreted as matching a *query* against a set of *keys*, where each key corresponds to a different *value*, so that those values whose keys match best with the query are retrieved as output. That is, mapping a query and a set of key-value pairs to an output.

And so, attention weights in the Transformer are calculated by matching a queries tensor Q with a keys tensor K. Even in this database-querying framework, there are many ways to perform this matching[2]. However, as the dot product is a measure of similarity, the attention weights can be simply calculated as a dot product between the queries and the keys, which is the approach the Transformer takes:

$$M = QK^\top$$

But where do the queries and keys come from? Well, you could take the approach of Section 2.1 and naively consider both of them to be X, computing M as the dot product $XX^\top$. This would indeed compute a matrix of shape (L, L) of similarities between every X_i and X_j. But that's not what we want. We need to extract from X a more abstract notion of attentions. Furthermore, in querying from a database, we want the question and the answer to represent different things, which is why we need to extract from them representations that can adequately match them.

[2]Additive attention, dot-product attention, scaled dot-product attention, general dot-product attention, etc.

Extraction of features? That's what neural networks are for! So, we can compute the queries and keys, just like the values, as fully-connected layers on X:

QK^T learns the dependencies between different elements of the sequence and uses that to capture the internal structure of the sequence, to distribute over V

Well, you could do it with X, but as with V, more features can be extracted, furthermore, you want 2 different representations to simulate asking a question.

as well as the reason it is called Scaled Dot-Product Attention. Notice that Q is of shape $(..., L, d_{model})$ and $K^\top$ is of shape $(..., d_{model}, L)$. As a result, the matrix product $QK^\top$ (and the additional relative position encoding S^{rel}) is of shape $(..., L, L)$. Given the goal of the attention mechanism, we would want the i, j element of the output of the compatibility function to represent the amount of attention that the ith element of the input sequence should pay to its jth element. One can imagine that if W^Q and W^K were properly optimized by backpropagation, the representations of the input sequences in Q by XW^Q and in K by XW^K would be such that the matrix product $QK^\top$ achieves this goal.

But we want to do more than that. We want to find vector representations

And what do they really mean? The names queries and keys are just nice ways to think about it, but that's not really what's happening.

dk can vary but in practice it's usually the same as dmodel for convenience.

So how is it performed? By creating two more representations of the input sequence, the queries tensor and the keys tensor to match it against, and taking their dot product. The query-key matching determines which values to focus on, and the output is the sum over those values that we focus on.

sum over j similarity (qi, kj) * vj = attention

so attention really isn't a retrieval process, but it mimics it in a probabilistic way because of the extremely high slope of the softmax. generalization of retrieval from a database for the keys that have a high similarity to the query in the database.

q and k have to be mapped to new spaces to be able to be compared in some meaningful way - we don't know how, it's like answering a question - you want the answer to contain information that was not in the question, but you know that they match.

The weights come from a notion of similarity between the weights and the keys.

We're doing a proportional retrieval according to a probability vector M_i. under softmax, weighted summation becomes weighted average.

In the Transformer, attention weights are calculated by a scaled dot-product - hence the name, Scaled Dot-Product Attention. Dot product between what? Well the developers of the Transformer interpreted the selection of values to attend to as similar to retrieving those values from (i.e., querying) a database. Consequently, they formulated their attention equation as matching a *query* against a set of *keys*, where each key corresponds to a different *value*, so that those values whose keys match

best with the query are retrieved as output. That is, mapping a query and a set of key-value pairs to an output. And as the dot product is a measure of matching, the dot product between a queries tensor and a keys tensor provides our attention weights.

That still doesn't answer the question, what does attention really mean in a computational context?

A transformation to Q would capture that ask for compatibilility.

But what does attention really mean

So, the first step in the Attention block is to transform the input sequence by fully-connected layers into the queries, keys, and values (we'll get to the values later):

$$Q = XW^Q + B^Q$$
$$K = XW^K + B^K$$
$$V = XW^V + B^V$$

where the parameter tensors W^i and B^i are learned by gradient descent. In order to preserve the shape of X, i.e., (L, d_{model}), each W^i must be shaped (d_{model}, d_{model}), and each B^i must be shaped $(d_{model},)$. Thus, each Q_i, K_i, and V_i is a new vector representation of X_i of shape $(d_{model},)$.

Observe that Q is of shape (L, d_{model}) and so is K. Thus, $QK^\top$ is of shape (L, L). Considering Q and K to be the representations of X we seek to compare, one can imagine that if W^Q, Q^K, B^Q, and B^K were optimized properly by gradient descent, Q and K would be such that their dot product, namely $QK^\top$, would be the attention matrix we seek, where $QK^\top_{ij}$ represents the attention X_j pays to X_i as the dot product $Q_i \cdot K_j$.

Under softmax, this sum becomes a weighted average.

But what do we do with this now?. We need to distribute this over the sequence to sum up the importances of each element to every other element, to determine the overall structure of the sequence, the overall importance of one element to all the other elements. How do we do this? Another dot product. We could do this with just X itself, however, that is bad because we could also try to extract an even more abstract, even more pure representation of the information in X by affine transformation, which is what the fully connected layer into the values space achieves. Thus, a chain of matrix multiplication along the following lines:

$$QK^\top V$$

achieves the distribution of the information of the attention of the sequence that we want, captures the structure of the sequence in terms of how important each element is to every other element in the sequence, i.e., it stores in each new element the entire sum of information about how important that element is to the entire sequence.

Furthermore, this type of representation, preserving the original shape of X allows us to chain layers of the Transformer to extract ever purer representations of that attention information.

What we need is a new representation of the sequence with this information embedded into it

V extracts even higher what X means, how X is constructed, and QK^T distributes the attention information through V to determine which values are the most important. Furthermore, this allows for chaining of the Transformer Layers.

What we want to build is extract at an even higher level, the information about which elements in the sequence are more important in constructing the entire sequence. How can we achieve that? Precisely, by distributing the information in $QK^\top$ throughout the sequence.

The greater the match between the query and the key, the more the value corresponding to that key is unlocked. It's like a key value database

What this seeks to achieve is to determine how much a query lines up with a key, and then use that much of the value in creating the next representations of the sequence

Assuming this information about importance is contained within the sequence itself, it can theoretically be learned by backpropagation.

Attention, or more formally, Scaled Dot-Product Attention, is given by:

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

where Q is the queries tensor, K is the keys tensor and V is the values tensor. For those who don't know what softmax is, it's a function that can convert each vector in a tensor, or each matrix in a tensor, or the tensor itself, into a probability distribution. Look it up.

note: positional information is also important for generalizing the model to arbitrary sequence length note: model did significantly better without absolute positional encoding, with only relative attention

References

[1] C. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, C. Hawthorne, A. M. Dai, M. D. Hoffman, and D. Eck, "Music Transformer: Generating music with long-term structure," 2018.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," 2017.