Lecture 19: 10/27 (Thursday)

Lecturer: Prof. Anant Sahai

Scribes: Nabeel Hingun

1 Agenda

1.1 Attention, ResNet Style Blocks with MLPs and LN

Last time, we talked about different ingredients that go into building a transformer [3]. The basic core ingredient was the attention mechanism, which is just a query-able softmax pooling. We also talked about the multi-headed variant where we can execute many different queries at once into many different tables.

There are two kinds of attention: self-attention and cross-attention. The difference lies in whether we are querying the same table we are sticking stuff into or a different table. Attention by itself has no learn-able parameters associated with it so we need to have learn-able parameters making the queries, keys and values. On top of that, we have ResNet-style blocks to combine something designed to use the attention mechanism with standard building blocks like multi-layer perceptrons and layer normalization to make something we can stack and make deep.

1.2 Position Encoding

The input to the transformer has no sense of ordering. To have some way of imposing order, we use position encoding to encode the positions of input tokens. The one we talked about last time was inspired by the hands of the clock, i.e, sines and cosines. This method is friendly to matrix multiplication and has the nice analogy to advancing the hands of the clock or moving back the hands of the clock. It gives us an absolute encoding of position which is friendly to relative position querying.

2 Variant of Position Encoding

There are many variations of position encoding. In particular, there is a distinction between the one we talked about in last lecture and the one that is used in practice. Last time, we talked about implementing position encoding as concatenating our input to the position encoding (fig. 19.1). Here, we can think of the input to the transformer as having two parts: (1) the data and (2) where that data was in the sequence.

While this 'concatenation' method can be done in practice and is very logical, a more common approach is to have the input and position added together (fig. 19.1). The question that arises is why this even works as it seems less natural.

First, recall that the vector sum is happening in a very high dimensional space (512, 1024, ... dimensions). In such high dimensional spaces, the position encodings are occupying a very limited set of the space which leaves a lot of space for the input data. Then, suppose we want to query for the position only. When we take the inner product, by linearity, we get (1) the inner product of the position oriented query and the input, added to (2) the inner product of the position oriented query

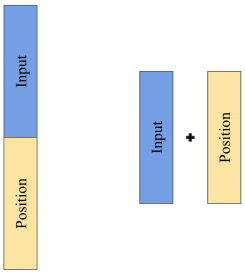

Figure 19.1: Position Embedding: Concatenation (Left) v/s Addition (Right)

with the position encoding. If the position encoding has a high correlation with the query, then (2) will be high whereas the position oriented query and the input will be something completely different. Therefore, the inner product has the ability to be able to pick up positions vs things in the input. Of course, there will be some interference with the query but not that much and so we expect that the sum can still make position encoding work. Thus, in high-dimensional space, adding is also fine for the inner product to be able to pick up positions versus things in the input. Addition is used in practice and usually has better performance.

Question: We want to use complex numbers because matrix multiplications are rotations (of the clock) but we are not actually using complex numbers. By using only sines and cosines, so do we still have some property of rotation?

Answer: The idea of the complex numbers rotating around is used as inspiration for the hands of the clock since we know complex multiplication can result in rotating these relative positions. Yet, if we represent complex numbers using a vector of two numbers, then we can think of complex multiplication as a linear operation represented by a matrix. This means that with matrix multiplication, it is easy to express a rotation, so the same property we had with the multiplication of complex numbers, we can basically inherit from the vector representation using sines and cosines.

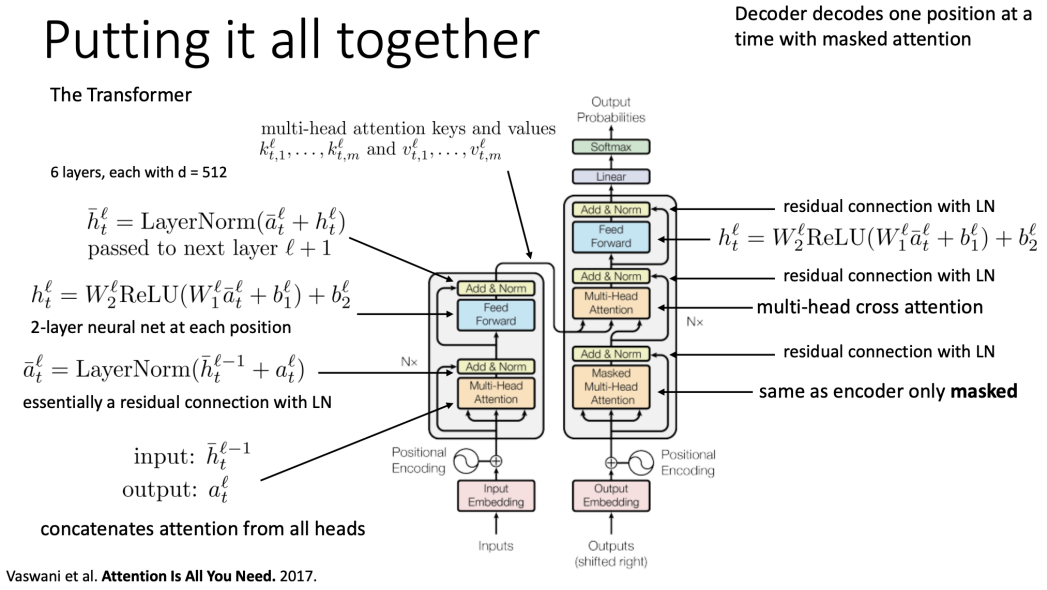

Figure 19.2: On the left, we have an encoder with N_x encoder blocks. The encoder is usually pre-trained with some specific task. On the right, we have a decoder made up of N_x decoder blocks. [1]

Question: If you want to rotate the hands of a clock by some angle, say 10 degrees, then we have to compute the sines and cosines of 10 degrees to populate the transformation matrix. Yet, if we look at the linear and non-linear operations in a transformer that define the entire path of the query or keys, we don't see a sine or cosine as a non-linearity. So how is it that we are actually able to move by 10 degrees?

Answer: The queries are the result of a learned transformations. We don't need to explicitly compute the sine and cosine computation at runtime. The transformation just has to be learn-able during training. Whatever the sines and cosines we actually need will come out during training. It is easy to learn a rotation as a transformation.

Follow Up Question: Does that require a lot of weights if we want to learn all the relevant angles?

Answer: Yes, there are potentially many different relevant angles we need to learn and so indeed, we would have many weights. The hope is that the model has enough expressive power to learn the transformations we need. In fact, transformers are very large models with high memory cost. For a 1000-length sequence, each with hidden dimension 1000, we can will need 1 million parameters. In addition, to compute softmax, we need $O(10^9)$ number of multiplications and additions.

3 Transformer Components

3.1 Attention

The transformer (fig. 19.2) uses a combination of self-attention, cross attention, layer norms and MLPs. But how is that all working together and why do we need these different components? Traditionally, in a decoder block, we start with a self-attention layer (bottom right orange block in fig. 19.2) which is looking at the input and other things in the sequence that might be relevant to understand what it should do next. Then we have cross-attention (center right orange block in fig. 19.2). The cross-attention accesses some table with key-value pairs. Typically, the key-values pairs come from the encoder. Since we have to interpret the embedding of the input token in some way, there is a weight key matrix that generates the keys and another weight matrix that generates the values. These matrices, W_k and W_v have learn-able weights. Gradients from decoder shape W_k and W_v because it is related to the task the decoder is doing (interpret the encoder embedding). For example, in fig. 19.2, W_k and W_v are multiplied by the last output of the encoder to produce $k^l_{t,1}, ..., k^l_{t,m}$ and $v^l_{t,1}, ..., v^l_{t,m}$.

3.2 MLP

Finally, the output is passed through some MLP (fig. 19.2). We need the latter because we need a non-linearity stronger than the non-linearities involved in attention.

3.3 Layer Normalization

There is also a question of why we need layer normalization and what is it doing. Recall that the attention mechanism is computing an inner product. From the query's point of view, the inner product is trying to say, of all the different keys, which is the one which is the most in this direction. Then we can see that layer norm is useful because it gives the query a particular direction.

Question: Do we need to normalize the keys too?

Answer: Not really. The same query is applied to a bunch of keys but for some keys you want them to be able to say, "i'm in this direction, yes, but i'm not that strong. If there is another direction that is stronger, then pick that on". Whereas the query is really picking a direction. If we change the norm of the query, we would still end up with the same ordered sequence of winners of the keys but all that would change is what happens with the softmax, in terms of how well it does the normalization. But that's not what we want so we don't want to impose a normalization constraint on the keys.

3.4 Query Standardization

Input standardization involves three parts. Take data,

- subtract mean
- divide by standard deviation
- possibly shift the mean and standard deviation by something that's learn-able.

These steps involve two sides: normalizing the size of everything and moving the means around. When it comes to preventing exploding gradients or vanishing gradients, the means are not relevant whereas sizes are. In the context of a transformer, when it comes to what we want from the query, we want it to have a norm that is something reasonable and doesn't change a lot between queries. Then, the query doesn't need biases and only needs scaling. Hence, one of the modifications to the Transformers that is often done is to remove the bias and stick to the scaling.

Question: When we are looking at a particular encoder layer, we have a self attention block inside which is looking into a table. Are the contents of the table the same for different input positions?

Answer: Yes. They are only different if we enforce some sort of sequentiality. On the decoder side, the sequentiality is often required.

We usually have two different design choices when it comes to the inputs of the attention layers. For instance, in an arbitrary decoder, which output of the encoder should the attention layer of block 7 (of the decoder) be looking at? One option would be to have every block look at the last output of the encoder. So if the encoder is 20 levels deep, then it is looking at the output of level 20. Every single layer of the decoder here is looking at the output of the last layer, layer 20. This follows the auto-encoder spirit where the output of the encoder is a distilled version of the output. Typically, this is what is done.

Another option is in the style of a U-net where we can look over at different levels of fineness of the encoder. So we can imagine a transformer decoder looking at the middle layers of the encoder for attention.

Question: When we say tables or weights are shared, are they shared horizontally or vertically?

Answer: The standard answer is horizontally. For the same layer, we have the same weights and if we have an attention block, it will have the same table. But at a different layer, there will be a different set of weights and a different table that it looks into. That said, it is often sometimes done to have weight sharing across layers as well.

3.5 Data

Transformers don't have a strong inductive bias so we need a lot of data to train. We get data scraped from the web and hopefully if we get all of this, we can get things to learn the underlying representation of what that language is. That's what we want our model to actually capture and then we can use it for different purposes.

4 Word Embeddings

For us to be able to learn from text data, we need to have a way of representing words such that their representations are meaningful. For example, in computer vision, the pixels of images mean something. They don't mean much, but they mean something. Thus, maybe if we had a more meaningful representation of words, then learning downstream tasks would be easier!

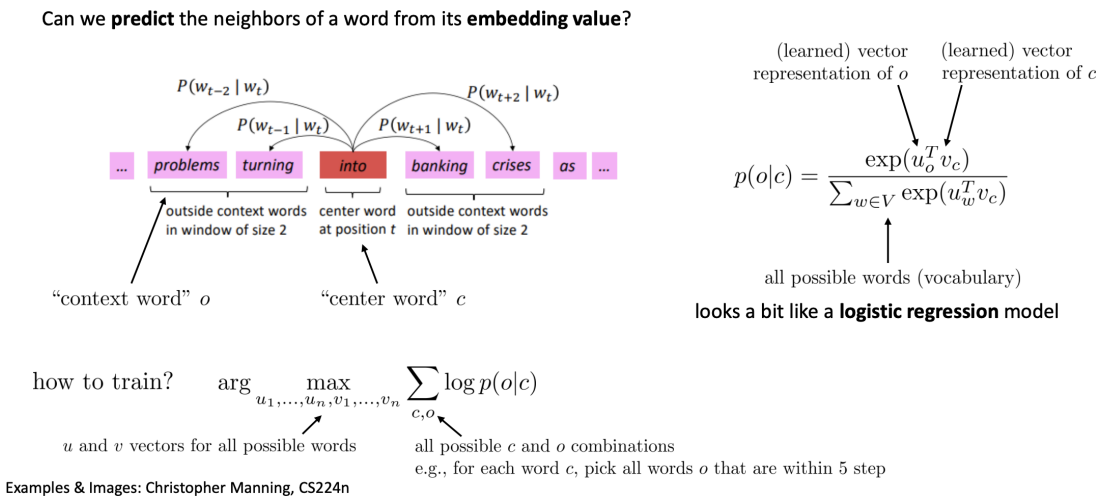

Figure 19.3: How we learn embeddings using word2vec [2]

We represent words with embeddings, but how do we learn these representations? The problem with words is that in the English language alone, you have something of the order of 100,000 words, with 8,000 common words. That's a lot of words. The default way of thinking of categorical variables using one-hot encoding is really annoying for words. We would have lots of words close to each other in meaning but which would not be reflected in this embedding. For example, 'actual' and 'actually' are going to have the same distance from each other as 'actual' and 'banana'. Therefore, this encoding doesn't tell us anything about the meaning of these words.

There also is a question of why we want vectors corresponding to similar words to be close to each other in the embedding space. All of our functions are continuous during training, so we expect to have the basic behaviour of things going in that are close will give things that are close going out. In particular, for losses, during training, we have we want a kind of continuity where words that are close get less words than things that are far. Then, if we have representation with vectors of similar words being close to each other, then hopefully if we make a small mistake during training, we will have something that's also close.

Now that we want our vectors to have this property, the question becomes, how do we learn them? The key idea in word2vec is that words occur in context with other words, i.e, the words that surround a word tells you what a word is like. Therefore, we want to find a representation that reflects this idea of having similar words have close embeddings.

Suppose, we want to learn the embedding for the center word c (fig. 19.3). We create a prediction task, such that, we try to predict the neighbors of the center word. To generate this probability, we use the standard go to way, i.e, we predict the words that are closest to the center word in terms of the inner product. Thus, we can take the embedding for a word o given the context and sum up over all other choices that could be there for the context word.

Question: Intuitively, what we want to do is to have an encoding of embedding of words so that given the neighbors, we can predict a particular word. Here, with word2vec, we are doing what feels like the opposite, i.e, we are predicting the neighbors given a particular word. Why do we do it this way?

Answer: The answer is simply because it is easier to do it this way. If there are multiple inputs and a single output, we need to figure how to combine all this information from the input. It is possible to do it, maybe through a weighted average of the neighbors, but at the end of the day, this is a surrogate task, so we can try the easiest thing first.

Note: There is a subtle distinction that two words should have the same representation if they are roughly interchangeable with each other in a sentence, not that they sit close to each other in a sentence.

References

[1] S. Levine. https://cs182sp21.github.io/static/slides/lec-12.pdf. 2021.

[2] S. Levine. https://cs182sp21.github.io/static/slides/lec-13.pdf. 2021.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. 2017.