Transformers: Learning with Purely Attention Based Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Monday 18th April, 2022 22:15

Preamble

So far you have seen two major architectural elements in the neural networks meant for deep learning (DL): convolutional layers and recurrence layers. Until recently, they were the primary reasons for the fame and glory that have been bestowed on DL during recent years.

But now we have another element: attention layers.

That difficult problems could be solved with neural networks through purely attention based logic — that is, without convolutions and recurrence — was first revealed in the paper "Attention is All You Need" by Vaswani et el. that you can access here:

`https://arxiv.org/pdf/1706.03762.pdf`

The goal of this lecture is to explain the basic concepts of attention-based learning with neural networks.

My explanations will be in the context for sequence-to-sequence learning as required for automatic translation. In particular, I will focus on English-to-Spanish translation as a case study.

Preamble (contd.)

In the context of seq2seq learning, attention takes two forms: self-attention and cross-attention.

Self-attention means for a neural network to figure out on its own what parts of a sentence contribute together to the generation of the words in the target language.

To elaborate, consider the following sentence in English:

> *I was talking to my friend about his old car to find out if it was still running reliably.*

For a machine to understand this sentence, it has to figure out that the pronoun "it" is strongly related to the noun "car" occurring earlier in the sentence. A neural network with self-attention would be able to do that and would therefore be able to answer the question:

> *What is the current state of Charlie's old car?*

assuming that system already knows that "my friend" in the sentence is referring to Charlie.

Preamble (contd.)

For another example, consider the following Spanish translation for the above sentence:

> *Yo estaba hablando con mi amigo acerca su viejo coche para averiguar si todavía funcionaba de manera confiable.*

In Spanish-to-English translation, the phrase "*su viejo coche*" could go into "*his old car*", "*her old car*", or "*its old car*". Choosing the correct form would require for the neural-network based translation system to have established the relationship between the phrase "*su viejo coche*" and the phrase "*mi amigo*". A neural network endowed with self-attention that operates at a level higher than a sentence would be able to do that.

While self-attention allows a neural network to establish the sort of intra-sentence word-level and phrase-level relationships mentioned above, a seq2seq translation network also needs what's known as cross-attention.

Cross attention means discovering what parts of a sentence in the source language are relevant to the production of each word in the target language.

Preamble (contd.)

To see the need for cross-attention, consider the fact that in the English-to-Spanish translation example mentioned previously, the Spanish word "averiguar" has several nuances in what it means: it can stand for "to discover", "to figure out", "to find out", etc.

With cross-attention, during the training phase, the neural network would learn that when the context in the English sentence is "friend", it would be appropriate to use "averiguar" for the translation because one of its meanings is "to find out."

Along the same lines, in English-to-Spanish translation, ordinarily the English word "running" would be translated into the gerund "corriendo" in Spanish, however, on account of the context "car" and through the mechanism of cross-attention the neural network would learn that "running" is being used in the context of a "car engine", implying that that a more appropriate Spanish translation would be based on the verb "funcionar".

Preamble (contd.)

In this lecture, I'll be teaching purely-attention based learning with the help of my DLStudio class whose version 2.2.2 comes with the following two slightly different implementations of the transformer architecture:

- TransformerFG
- TransformerPreLN

The suffix "FG" in `TransformerFG` stands for "First Generation". And the suffix "PreLN" inn `TransformerPreLN` stands for "Pre Layer Norm".

The `TransformerFG` implementation is based on the transformers as first envisioned in the seminal paper "Attention is All You Need" by Vaswani et el.:

`https://arxiv.org/pdf/1706.03762.pdf`

The other transformer class, `TransformerPreLN`, incorporates the modifications suggested by "On Layer Normalization in the Transformer Architecture" by Xiong et al.:

`https://arxiv.org/pdf/2002.04745.pdf`

Preamble (contd.)

`TransformerFG` and `TransformerPreLN` are the two inner classes of the `Transformers` co-class of DLStudio. That is, the `Transformers` is the container class for both `TransformerFG` and `TransformerPreLN`. Recall that, in the DLStudio module, a co-class resides at the same level of abstraction as the main DLStudio class.

About the dataset I'll be using to demonstrate transformers, version 2.2.2 of DLStudio comes with the following data archive:

```
en_es_xformer_8_90000.tar.gz
```

In the name of the archive, the number 8 refers to the maximum number of words in a sentence, which translates into sentences with a maximum length of 10 when you include the SOS and EOS tokens at the two ends of a sentence. The number 90,000 is for how many English-Spanish sentence pairs are there in the archive.

These two scripts in the `ExamplesTransformers` directory of the distribution are your main entry points for experimenting with the transformer code in DLStudio:

```
seq2seq_with_transformerFG.py

seq2seq_with_transformerPreLN.py
```

Outline

Outline

On Explaining the Attention in Transformers

- Modern attention networks were developed originally for solving seq2seq learning problems as required for automatic translation from one language to another. Even though more recently attention networks have also been used for solving problems in other domains — for example, problems in computer vision — seq2seq learning still feels like the most natural "domain" for discussing transformers.
- Computation of attention requires representing the basic units of your input domain with a triple of vectors denoted q for Query, k for Key, and v for Value. When all of the input units are considered together in the form of tensors, the same vectors become tensors and are denoted Q for Query, K for Key, and V for value.
- After you have become comfortable with representing the domain information with Query, Key, and Value vectors, the next idea you'd need to conquer is dot-product attention. In this section, I'll introduce these ideas and the develop the notion of Single-Headed Attention.

Expressing Words Through Their (q, k, v) Vectors

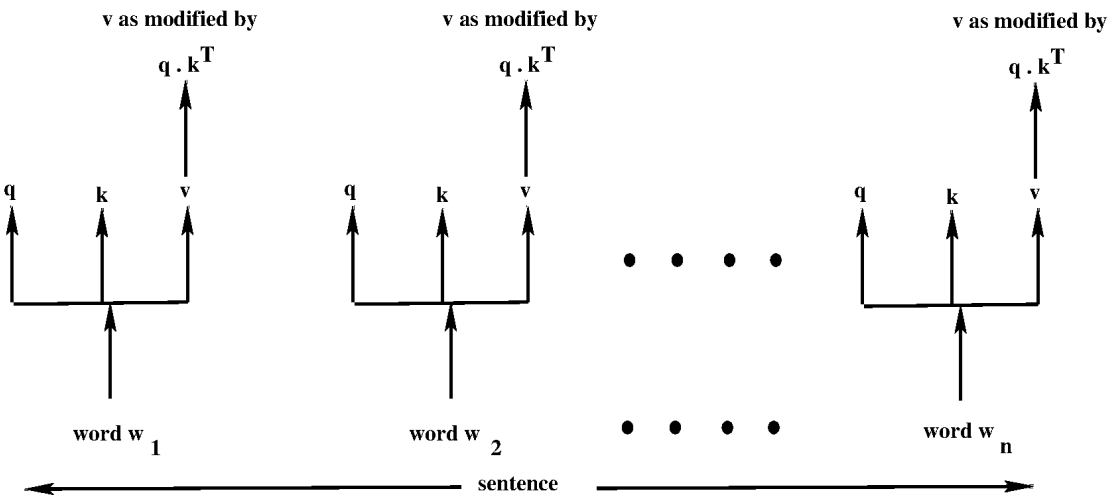
- What makes attention networks unique in deep learning is that the Query, Key, and Value vectors is created **NOT** by convolution, **NOT** by recurrence, but by matrix multiplication.
- For seq2seq learning, we want to express each word w in a sentence through a query vector q, a key vector k, and a value vector v, with these three vectors being obtained through three learnable matrices W_q, W_k, and W_v as follows:

$$q = w \cdot W_q \qquad k = w \cdot W_k \qquad v = w \cdot W_v \quad (1)$$

  where w is a vector of numbers that numerically represents a word in the input sentence.
- You can think of the q, k, and v vectors as a word w's three representatives for assessing the importance of the word in question to every other word in a sentence.

The (q, k, v) Vectors for the Words (contd.)

- Continuing with the thought in the last bullet of the previous slide, a word w_1 would consider a dot product of its own q vector with another word w_2's k vector for estimating its relevance to w_2 and use the result of that dot product to modify its own v vector.
- Obviously, loosely speaking, there is likely to be a certain mutuality and symmetry to how the v vectors for the different words get modified in this manner.
- The figure shown below should help with the visualization of the idea.



The (q, k, v) Vectors for the Words (contd.)

- At this point, one might ask: If the triplet of the vectors (q, k, v) are to be obtained for a word through matrix multiplication of certain learnable matrices with the words, that implies that the words themselves are being represented as vectors. What's nature of those vectors?
- As you would expect on the basis of what you have already seen in the Week 13 lecture on seq2seq learning, we can express the words through their embedding vectors as learned by the `nn.Embedding` class.
- I'll use the symbol M to denote the size of the embedding vectors for the words.
- I'll also use the symbol s_{qkv} to denote the size of the Query, Key, and Value vectors for each word.
- Therefore, the matrices W_q, W_k, and W_v must each be of shape $M \times s_{qkv}$.

From Word-Based (q, k, v) Vectors to Sentence-Based (Q, K, V) Tensors

- In the explanation so far, I considered each word separately because my goal was to convey the basic idea of what is meant by the dot-product attention. In practice, one packs all the words in a sentence in a tensor of two axes, with one axis representing the individual words of the input sentence and other axis standing for the embedding vectors for the words. In what follows, I'll use X to denote the input sentence tensor. (NOTE that, for a moment, I am ignoring the fact that X will also have a batch axis.)

- With all the words of a sentence packed into the tensor X, we can set things up so that the network learns all of the matrices W_q, W_k, and W_v for all the words in a sentence simultaneously. We can therefore visualize a triplet of learnable tensors (W_Q, W_K, W_V) whose different axes would correspond to the individual-word (W_q, W_k, W_v) matrices.

Calculating the (Q, K, V) Tensors

- Calculation of the sentence-level Query, Key, and Value tensors can be expressed compactly as

$$Q = X \cdot W_Q \qquad K = X \cdot W_K \qquad V = X \cdot W_V \quad (2)$$

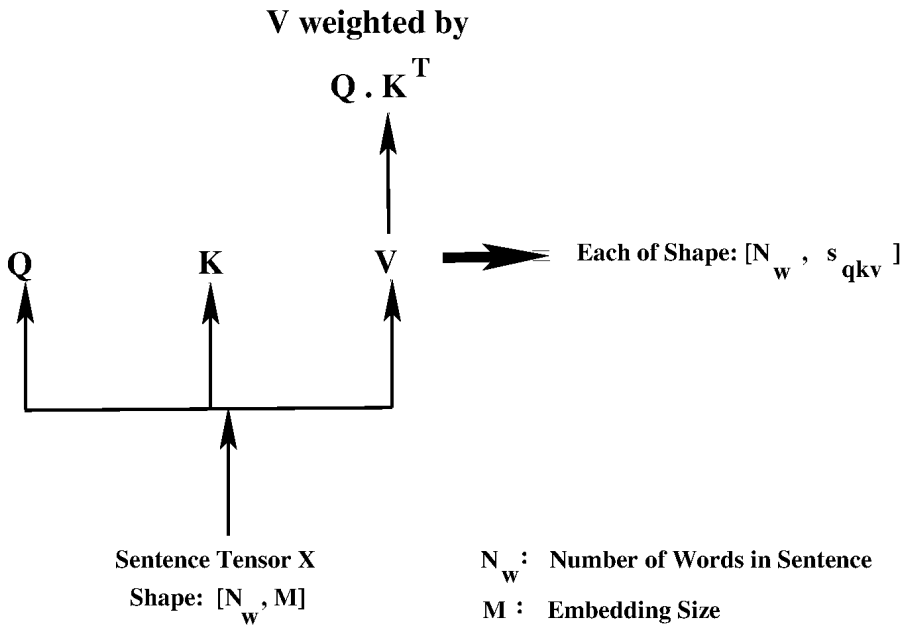
- The tensor Q packs all the word-based query vectors into a single data object. The tensor K does the same for the word-based key vectors, and the tensor V for the value vectors.

- Using N_w to denote the number of words in a sentence, we have $[N_w, M]$ for the shape of the input tensor X; $[N_w, s_{qkv}]$ for the shapes of the output Q, K, and V tensors; and $(N_w * M) \times (N_w * s_{qkv})$ for the shapes of the matrices W_Q, W_K, W_V.

- Using the Q, K, and V tensors, we can express more compactly the calculation of the attention through a modification of the V tensor via the dot-products $Q \cdot K^T$ as shown on the next slide.

Calculating Attention with (Q, K, V) Tensors

- Using Q, K, and V tensors, the visual depiction of the attention calculation shown earlier on Slide 12 can be displayed more compactly as:



- Recall that in the above depiction, N_w is the number of words in a sentence, M the size of the embedding vectors for the words, and s_{qkv} the size of the word-level original q, k, v vectors.

Calculating Attention with (Q, K, V) (contd.)

- In Python, the dot product of the Q and K tensors can be carried out with a statement like

```
QK_dot_prod  =  Q @ K.transpose(2,1)
```

  where `@` is Python's infix operator for matrix multiplication. As you can see, the transpose operator is only applied to the axes indexed 1 and 2. Axis 0 would be for the batch index.

- A tensor-tensor dot-product of Q and K directly carries out all the dot-products at every word position in the input sentence. Since Q and K are each of shape $[N_w, s_{qkv}]$ for an N_w-word sentence, the inner-product $Q \cdot K^T$ is of shape $N_w \times N_w$, whose first N_w-element row contains the values obtained by taking the dot-product of the first-word query vector q_1 with each of the k_1, k_2, k_3, ..., k_{N_w} key vectors for each of the N_w words in the sentence. The second row of $Q \cdot K^T$ will likewise represent the dot product of the second query vector with every key vector, and so on.

Calculating Attention with (Q, K, V) (contd.)

- The dot-product attention is expressed in a probabilistic form through its normalization by `nn.Softmax()` as shown below.
- The following formula shows us calculating the attention — meaning the attention-weighted values for the Value tensor V — using the `nn.Softmax` normalized dot-products:

$$Z = \frac{nn.Softmax(Q \cdot K^T)}{\sqrt{M}} \cdot V \tag{3}$$

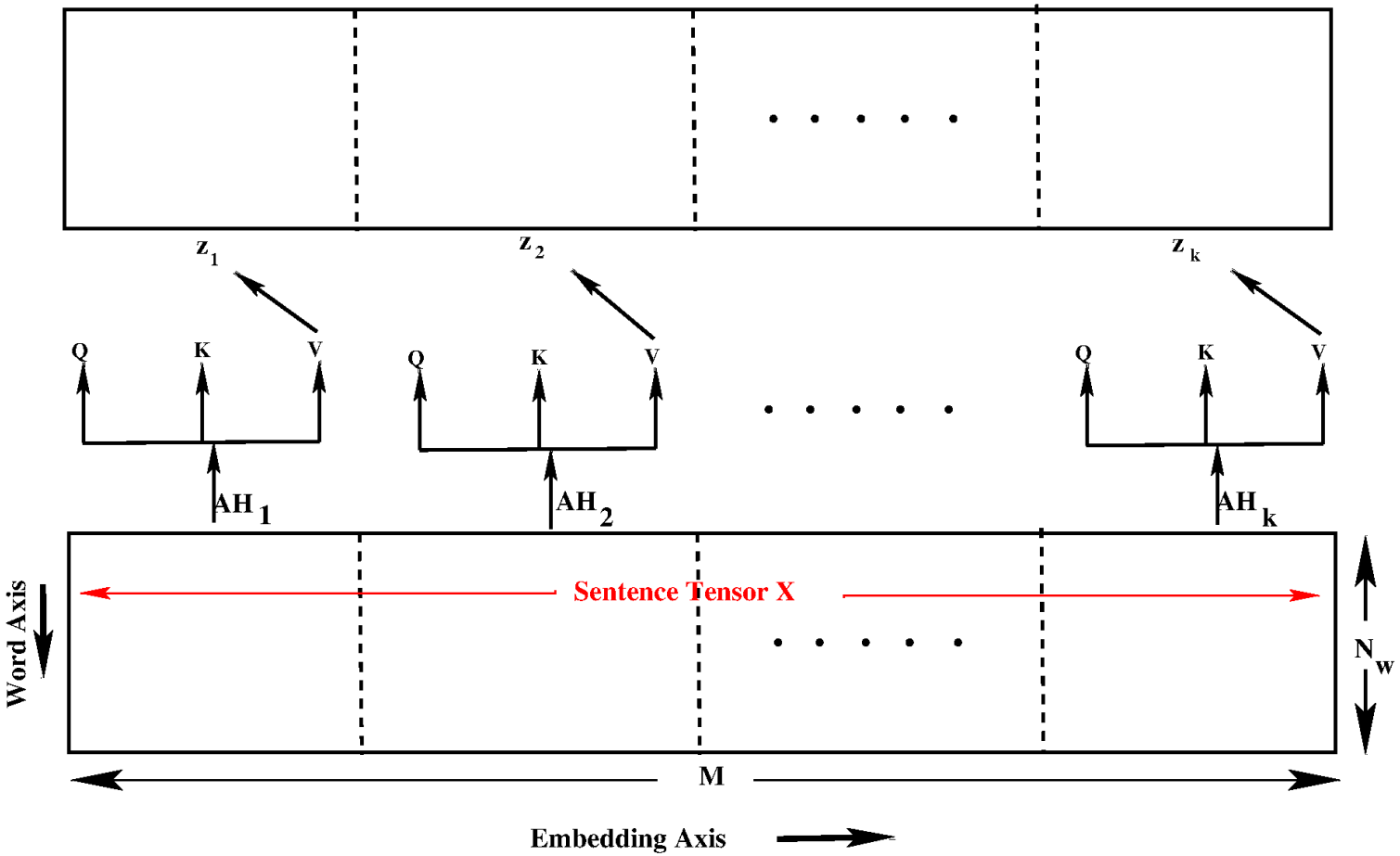
- The additional normalization by $\sqrt{M}$ in the formula shown above is needed to counter the property that large dimensionality for the embedding vectors can result in large variances associated with the output of the dot products.
- The above can be established by the fact that if you assume zero-mean unit-variance independent values for the components $x_i$and y_i in the summation $z = \sum_{i=1}^{N} x_i \cdot y_i$, the output z will also be zero-mean but its variance will equal N.

Outline

Multi-Headed Attention

- What I have described in the previous section is referred to as a Single-Headed Attention. As it turns out, single-headed attention is not sufficiently rich in its representational power for capturing all the needed inter-word dependencies in a sentence.
- Shown on the next slide is an illustration of Multi-Headed Attention. We now partition the input tensor X along its embedding axis into K slices and apply single-headed attention to each slice as shown in the figure.
- That is, each Attention Head gets to focus on a slice along the embedding dimension of the input sentence tensor.
- For reasons that I'll make clear later, I'll denote the size of the embedding slice given to each Attention Head by the same notation s_{qkv} that you saw earlier.

Multi-Headed Attention (contd.)



Multi-Headed Attention (contd.)

- Continuing with the notations used for Multi-Headed Attention, I'll use N_H to denote the number of Attention Heads used. Since s_{qkv} is the size of the embedding slice fed into any single attention head, we have

$$s_{qkv} = \frac{M}{N_H} \tag{4}$$

- Each Attention Head learns its own values for the Q, K, and V tensors with its own matrices for W_Q, W_K, and W_V.
- While each Attention Head receives only a s_{qkv}-sized slice from the embedding axis of the input sentence, the output tensors Q, K, V will still be of shape $[N_w, s_{qkv}]$ for the same reason as described in the previous section.
- Since for each Attention Head, Q and K are shape $[N_w, s_{qkv}]$ for an N_w-word sentence, the inner-product $Q \cdot K^T$ is of the same shape as in the previous section, that is $N_w \times N_w$.

Outline

Attention Head Implementation

- The next slide shows the implementation of the `AttentionHead` class in Version 2.2.2 of DLStudio whose co-class `Transformers` is the keeper of all transformer related code in the module.
- In the code shown on the next slide, all the dot-products mentioned previously are calculated in line (N). Next, as shown in line (O) we apply the nn.Softmax normalization to each row of the $N_w \times N_w$-sized $Q \cdot K^T$ dot-products calculated in line (N).
- The resulting $N_w \times N_w$ matrix is then used to multiply the $N_w \times s_{qkv}$-sized V tensor as shown in line (V). The operations carried out in lines (M) through (Q) of the code shown below can be expressed more compactly as:

$$Z = \frac{nn.Softmax(Q.K^T)}{\sqrt{(M)}} \cdot V$$

At this point, the shape of Z will be $N_w \times s_{qkv}$ — ignoring again the batch axis. This is the shape of the data object returned by each AttentionHead instance.

Attention Head Class in DLStudio's Transformers Class

```
class AttentionHead(nn.Module):

    def __init__(self, dl_studio, max_seq_length, qkv_size, num_atten_heads):
        super(Transformers.TransformerFG.AttentionHead, self).__init__()
        self.dl_studio = dl_studio
        self.qkv_size = qkv_size
        self.max_seq_length = max_seq_length                                                    ## (A)
        self.WQ =  nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )  ## (B)
        self.WK =  nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )  ## (C)
        self.WV =  nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )  ## (D)
        self.softmax = nn.Softmax(dim=1)                                                        ## (E)

    def forward(self, sent_embed_slice):      ## sent_embed_slice == sentence_embedding_slice  ## (F)
        Q = self.WQ( sent_embed_slice.reshape(sent_embed_slice.shape[0],-1).float() )         ## (G)
        K = self.WK( sent_embed_slice.reshape(sent_embed_slice.shape[0],-1).float() )         ## (H)
        V = self.WV( sent_embed_slice.reshape(sent_embed_slice.shape[0],-1).float() )         ## (I)
        Q = Q.view(sent_embed_slice.shape[0], self.max_seq_length, self.qkv_size)             ## (J)
        K = K.view(sent_embed_slice.shape[0], self.max_seq_length, self.qkv_size)             ## (K)
        V = V.view(sent_embed_slice.shape[0], self.max_seq_length, self.qkv_size)             ## (L)
        A = K.transpose(2,1)                                                                  ## (M)
        QK_dot_prod = Q @ A                                                                   ## (N)
        rowwise_softmax_normalizations = self.softmax( QK_dot_prod )                          ## (O)
        Z = rowwise_softmax_normalizations @ V                                                ## (P)
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(self.dl_studio.device)  ## (Q)
        Z = coeff * Z                                                                         ## (R)
        return Z
```

Self-Attention in DLStudio's Transformers Co-Class

- A `SelfAttention` layer concatenates the outputs from all AttentionHead instances and presents that concatenated output as its own output.
- For an input sentence consisting of N_w words and the embedding size denoted by M, the sentence tensor at the input to `forward()` in Line (B) in the `SelfAttention` code on the slide after the next will be of shape $[B, N_w, M]$ where B is the batch size.
- As explained earlier, this tensor is sliced off into *num_atten_heads* sections along the embedding axis and each slice shipped off to a different instance of AttentionHead.
- Therefore, the shape of what is seen by each AttentionHead is $[B, N_w, s_{qkv}]$ where s_{qkv} equals M/num_atten_heads. The slicing of the sentence tensor, shipping off of each slice to an AttentionHead instance, and the concatenation of the results returned by the AttentionHead instances happens in the loop in line (C).

Self Attention (contd.)

- You will add significantly to your understanding of how the attention mechanism works if you realize that the shape of the output tensor produced by a `SelfAttention` layer is exactly the same as the shape of its input. That is, if the shape of the input argument `sentence_tensor` in Line (B) is $[B, N_w, M]$, that will also be the shape of the output produced by layer.
- If you would not mind ignoring the batch axis for a moment, the input/output tensor shapes for a `SelfAttention` layer are both $[N_w, M]$ where N_w is the number of words in the input sentence and M the size of the embedding vector for each word. You could therefore say that the basic purpose of self-attention is to generate attention-enriched versions of the embedding vectors for the words.
- As you will see later, the statement made above also extends to all of the components of a transform.

Self Attention (contd.)

```
class SelfAttention(nn.Module):

    def __init__(self, dls, xformer, num_atten_heads):
        super(Transformers.TransformerFG.SelfAttention, self).__init__()
        self.dl_studio = dls
        self.max_seq_length = xformer.max_seq_length
        self.embedding_size = xformer.embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads
        self.attention_heads_arr = nn.ModuleList([xformer.AttentionHead(dls,
           self.max_seq_length, self.qkv_size, num_atten_heads)  for _ in range(num_atten_heads)])  ## (A)

    def forward(self, sentence_tensor):                                                             ## (B)
        concat_out_from_atten_heads = torch.zeros( sentence_tensor.shape[0],
                             self.max_seq_length, self.num_atten_heads * self.qkv_size).float()
        for i in range(self.num_atten_heads):                                                       ## (C)
            sentence_embed_slice = sentence_tensor[:, :, i * self.qkv_size : (i+1) * self.qkv_size]
            concat_out_from_atten_heads[:, :, i * self.qkv_size : (i+1) * self.qkv_size] =   \
                                                 self.attention_heads_arr[i](sentence_embed_slice)
        return concat_out_from_atten_heads
```

Outline

The Transformer Architecture

- Now that you understand the basics of the attention mechanism in a transformer, it is time to jump to a higher perspective on the overall architecture of a transformer.

- The overall architecture of a transformer is that of an Encoder-Decoder. The job of the Encoder is to create an attention map for the sentences in the source language and the job of the Decoder is to use that attention map for translating the source-language sentence into a target-language sentence.

- During training, the loss calculated at the output of the Decoder propagates backwards through both the Decoder and the Encoder. This process ensures that the attention map produced by the Encoder at its output reflects the intra-word dependencies amongst the source-language sentence that take into account what's needed for achieving the ground-truth translation in the target language.

The Transformer Architecture (contd.)

- While Encoder-Decoder is a simple way to characterize the overall architecture of a transformer, describing the actual architecture is made a bit complicated by the fact that the Encoder is actually a stack of encoders and the Decoder actually a stack of decoders as shown on Slide 34.
- In order to make a distinction between the overall encoder and the encoding elements contained therein, I refer to the overall encoder as the Master Encoder that is implemented by the class `MasterEncoder` in DLStudio's `Transformers` class. I refer to each of the individual encoders insider the MasterEncoder as Basic Encoders that are instances of the class `BasicEncoder`.

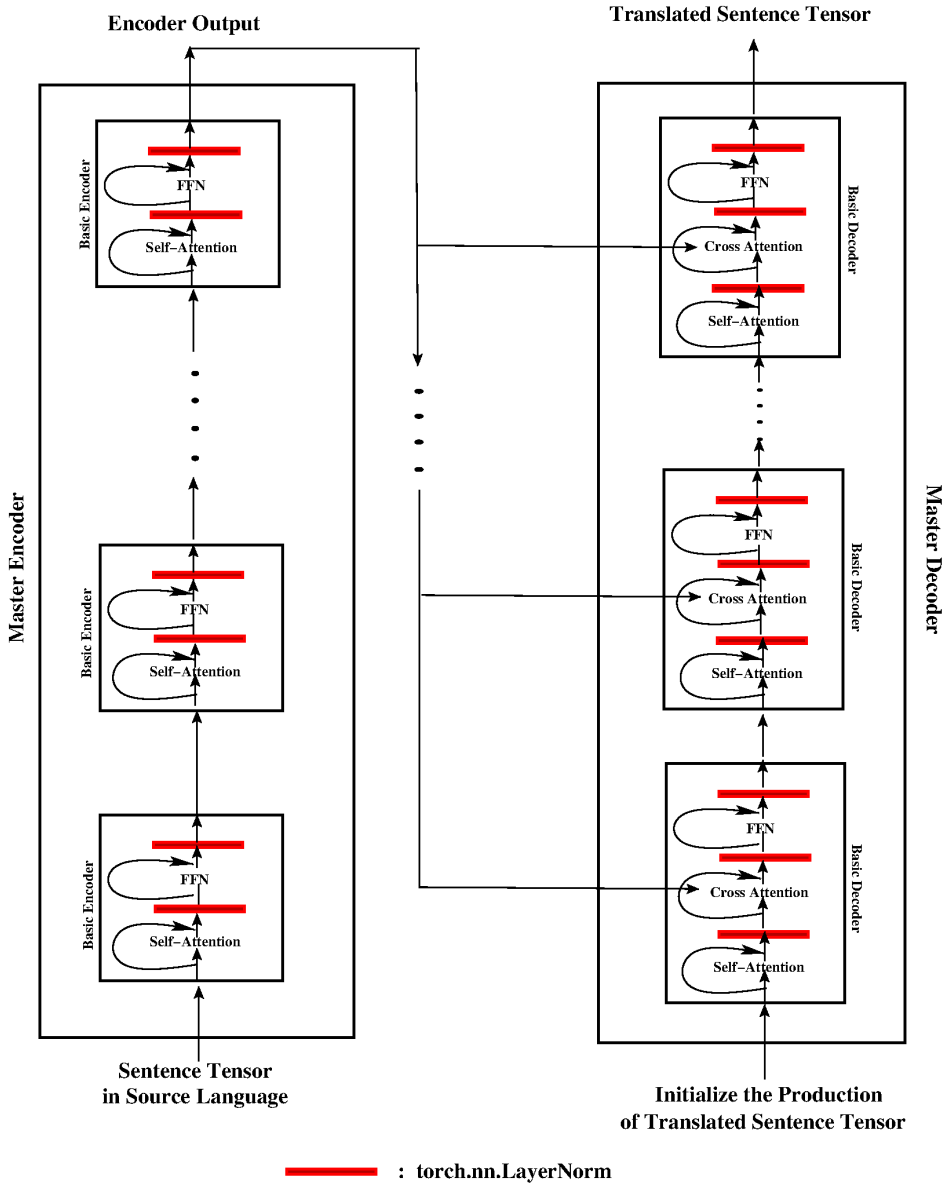
The Transformer Architecture (contd.)

- Similarly, on the decoder side, I refer to the overall decoder as the Master Decoder that is implemented in the class `MasterDecoder`. I refer to the different decoders in the Master Decoder as Basic Decoders that I have implemented with the class `BasicDecoder`.
- The implementation classes mentioned above have been explained in greater detail in the several sections that follow.
- Earlier I mentioned that, ignoring the batch axis, if the sentence tensor at the input to a layer of `SelfAttention` is of shape $[N_w, M]$, that's also the shape of its output.
- As it turns out, that shape constancy applies throughout the processing chains on the encoder and the decoder side. The final output of the Master Encoder will also be of shape $[N_w, M]$, as will be the shape of the input to the Master Decoder and the shape of the output from the Master Decoder.

The Transformer Architecture (contd.)

- The number of words as represented by N_w is the value of the variable `max_seq_length` in the transformer code presented later in this section.
- Therefore, one way of looking at all of the layers in the architecture shown on the next slide is that they are all engaged in using attention to enrich the embedding vectors of the words in order to allow the words to play different roles in different contexts and vis-a-vis what's needed for sequence-to-sequence translation to work correctly.

Encoder-Decoder Architecture for a Transformer



Outline

Master Encoder

- The main purpose of the MasterEncoder is to invoke a stack of `BasicEncoder` instances on a source-language sentence tensor.
- The output of each BasicEncoder is fed as input to the next BasicEncoder in the cascade, as illustrated in the loop in Line (B) below. The stack of BasicEncoder instances is constructed in Line (A).

```
class MasterEncoder(nn.Module):

    def __init__(self, dls, xformer, how_many_basic_encoders, num_atten_heads):
        super(Transformers.TransformerFG.MasterEncoder, self).__init__()
        self.max_seq_length = xformer.max_seq_length
        self.basic_encoder_arr = nn.ModuleList( [xformer.BasicEncoder(dls, xformer,
                                  num_atten_heads) for _ in range(how_many_basic_encoders)] )      ## (A)
    def forward(self, sentence_tensor):
        out_tensor = sentence_tensor
        for i in range(len(self.basic_encoder_arr)):                                             ## (B)
            out_tensor = self.basic_encoder_arr[i](out_tensor)
        return out_tensor
```

Outline

Basic Encoder

- The BasicEncoder in consists of a layer of self-attention (SA) followed by a purely feed-forward layer (FFN). You already know about what is accomplished by SA. The role played by FFN is the same as it does in any neural network — to enhance the discrimination ability of the network.
- The output of SA goes through FFN and the output of FFN becomes the output of the BasicEncoder.
- To mitigate the problem of vanishing gradients, the output of each of the two components — SA and FFN — is subject to Layer Norm. In addition, we use residual connections, one that wraps around the SA layer and the other that wraps around the FFN layer as shown in the figure on Slide 34.
- Deploying a stack of `BasicEncoder` instances becomes easier if the output tensor from a `BasicEncoder` has the same shape as its input tensor.

Basic Encoder (contd.)

- As shown on Slide 28, the `SelfAttention` layer in a Basic Encoder consists of a number of `AttentionHead` instances, with each `AttentionHead` making an independent assessment of what to say about the inter-relationships between the different parts of an input sequence.
- As you also know already, it is the embedding axis that is segmented out into disjoint slices for each `AttentionHead` instance. The calling `SelfAttention` layer concatenates the outputs from all its `AttentionHead` instances and presents the concatenated tensor as its own output.

```
class BasicEncoder(nn.Module):

    def __init__(self, dls, xformer, num_atten_heads):
        super(Transformers.TransformerFG.BasicEncoder, self).__init__()
        self.dls = dls
        self.embedding_size = xformer.embedding_size
        self.max_seq_length = xformer.max_seq_length
        self.num_atten_heads = num_atten_heads
        self.self_attention_layer = xformer.SelfAttention(dls, xformer, num_atten_heads)                          ## (A)
        self.norm1 = nn.LayerNorm(self.embedding_size)                                                            ## (B)
        ## What follows are the linear layers for the FFN (Feed Forward Network) part of a BasicEncoder
        self.W1 =  nn.Linear( self.max_seq_length * self.embedding_size, self.max_seq_length * 2 * self.embedding_size )
        self.W2 =  nn.Linear( self.max_seq_length * 2 * self.embedding_size, self.max_seq_length * self.embedding_size )
        self.norm2 = nn.LayerNorm(self.embedding_size)                                                            ## (C)

    def forward(self, sentence_tensor):
        sentence_tensor = sentence_tensor.float()
        self_atten_out = self.self_attention_layer(sentence_tensor).to(self.dls.device)                           ## (D)
        normed_atten_out = self.norm1(self_atten_out + sentence_tensor)                                           ## (E)
        basic_encoder_out =  nn.ReLU()(self.W1( normed_atten_out.view(sentence_tensor.shape[0],-1) ))             ## (F)
        basic_encoder_out =  self.W2( basic_encoder_out )                                                         ## (G)
        basic_encoder_out = basic_encoder_out.view(sentence_tensor.shape[0], self.max_seq_length, self.embedding_size )
        ## for the residual connection and layer norm for FC layer:
        basic_encoder_out =  self.norm2(basic_encoder_out  + normed_atten_out)                                    ## (H)
        return basic_encoder_out
```

Outline

Cross Attention Class in DLStudio's `Transformers` Class

- Before presenting the decoder side of a transformer network, I must first explain what is meant by Cross Attention and how I have implemented it in DLStudio's transformers.

- Whereas self-attention consists of taking dot products of the Query vectors for the individual words in a sentence with the Key vectors for all the words in order to discover the inter-word relevancies in a sentence, in cross-attention we take the dot products of the Query vectors for the individual words in the *target-language* sentence with the Key vectors at the output of the Master Encoder for a given *source-language* sentence. These dot products than modify the Value vectors supplied by the Master Encoder.

- In what follows, I'll use X_enc represent the tensor at the output of the `MasterEncoder`. Its shape will be the same as that of the source sentence supplied to the `MasterEncoder` instance.

Cross Attention (contd.)

- If N_w is the maximum number of words allowed in a sentence in either language, the X tensor that is input into the `MasterEncoder` will be of shape $[B, N_w, M]$ where B is the batch size, and M the size of the embedding vectors for the words.
- Therefore, the shape of the output of the `MasterEncoder`, X_enc, is also $[B, N_w, M]$. Now let X_target represent the tensor form of the corresponding target language sentences. Its shape will also be $[B, N_w, M]$.
- The idea of CrossAttention is to ship off the embedding-axis slices of the X_enc and X_target tensors to `CrossAttentionHead` instances for the calculation of the dot products and, subsequently, for the output of the dot produces to modify the Value vectors in what was supplied by the `MasterEncoder`.

Cross Attention (contd.)

```
class CrossAttention(nn.Module):

    def __init__(self, dls, xformer, num_atten_heads):
        super(Transformers.TransformerFG.CrossAttention, self).__init__()
        self.dl_studio = dls
        self.max_seq_length = xformer.max_seq_length
        self.embedding_size = xformer.embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads
        self.attention_heads_arr = nn.ModuleList( [xformer.CrossAttentionHead(dls,
              self.max_seq_length, self.qkv_size, num_atten_heads)  for _ in range(num_atten_heads)] )
    def forward(self, basic_decoder_out, final_encoder_out):
        concat_out_from_atten_heads = torch.zeros( basic_decoder_out.shape[0], self.max_seq_length,
                                                   self.num_atten_heads * self.qkv_size).float()
        for i in range(self.num_atten_heads):
            basic_decoder_slice = basic_decoder_out[:, :, i * self.qkv_size : (i+1) * self.qkv_size]
            final_encoder_slice = final_encoder_out[:, :, i * self.qkv_size : (i+1) * self.qkv_size]
            concat_out_from_atten_heads[:, :, i * self.qkv_size : (i+1) * self.qkv_size] =     \
                                 self.attention_heads_arr[i](basic_decoder_slice, final_encoder_slice)
        return concat_out_from_atten_heads
```

The `CrossAttentionHead` Class

- `CrossAttentionHead` works the same as the regular `AttentionHead` described earlier, except that now, in keeping with the explanation for the `CrossAttention` class, the dot products involve the Query vector slices from the target sequence and the Key vector slices from the `MasterEncoder` output for the source sequence.
- The dot products eventually modify the Value vector slices that are also from the `MasterEncoder` output for the source sequence. About the word "slice" here, as mentioned earlier, what each attention head sees is a slice along the embedding axis for the words in a sentence.
- If X_target and X_source represent the embedding-axis slices of the target sentence tensor and the `MasterEncoder` output for the source sentences, each `CrossAttentionHead` will compute the following dot products:

$$Q = X_{target} \cdot W_Q \qquad K = X_{source} \cdot W_K \qquad V = X_{source} \cdot W_V \tag{5}$$

`CrossAttentionHead` Class (contd.)

- Note that the Queries Q are derived from the target sentence, whereas the Keys K and the Values V come from the source sentences.

- The operations carried out in lines (N) through (R) can be described more compactly as:

$$Z_{cross} = \frac{nn.Sofmax(Q_{source} \cdot {K_{target}}^T)}{\sqrt{M}} \cdot V_{source} \tag{6}$$

```
class CrossAttentionHead(nn.Module):

    def __init__(self, dl_studio, max_seq_length, qkv_size, num_atten_heads):
        super(Transformers.TransformerFG.CrossAttentionHead, self).__init__()
        self.dl_studio = dl_studio
        self.qkv_size = qkv_size
        self.max_seq_length = max_seq_length
        self.WQ =  nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )                  ## (B)
        self.WK =  nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )                  ## (C)
        self.WV =  nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )                  ## (D)
        self.softmax = nn.Softmax(dim=1)                                                                        ## (E)

    def forward(self, basic_decoder_slice, final_encoder_slice):                                               ## (F)
        Q = self.WQ( basic_decoder_slice.reshape(final_encoder_slice.shape[0],-1).float() )                     ## (G)
        K = self.WK( final_encoder_slice.reshape(final_encoder_slice.shape[0],-1).float() )                     ## (H)
        V = self.WV( final_encoder_slice.reshape(final_encoder_slice.shape[0],-1).float() )                     ## (I)
        Q = Q.view(final_encoder_slice.shape[0], self.max_seq_length, self.qkv_size)                            ## (J)
        K = K.view(final_encoder_slice.shape[0], self.max_seq_length, self.qkv_size)                            ## (K)
        V = V.view(final_encoder_slice.shape[0], self.max_seq_length, self.qkv_size)                            ## (L)
        A = K.transpose(2,1)                                                                                    ## (M)
        QK_dot_prod = Q @ A                                                                                     ## (N)
        rowwise_softmax_normalizations = self.softmax( QK_dot_prod )                                            ## (O)
        Z = rowwise_softmax_normalizations @ V                                                                  ## (P)
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(self.dl_studio.device)                 ## (Q)
        Z = coeff * Z                                                                                           ## (R)
        return Z
```

Outline

The `BasicDecoderWithMasking` Class

- As with the `BasicEncoder` class, while a Basic Decoder also consists of a layer of `SelfAttention` followed by a Feedforward Network (FFN) layer, but now there is a layer of `CrossAttention` interposed between the two.
- The output from each of these three components of a Basic Decoder instance passes through a `LayerNorm` layer. Additionally, you have a residual connection that wraps around each component as shown in the figure on Slide 34.
- The Basic Decoder class in DLStudio's `Transformers` code is named `BasicDecoderWithMasking` for the reason described below.
- An important feature of the Basic Decoder is the masking of the target sentences during the training phase in order to ensure that each predicted word in the target language depends only on those target words that were seen PRIOR to that point.

The `BasicDecoderWithMasking` Class (contd.)

- This recursive backward dependency is referred to as autoregressive masking. In the implementation shown below, the masking is initiated and its updates established by the `MasterDecoderWithMasking` class to be described in the next section.

```
class BasicDecoderWithMasking(nn.Module):

    def __init__(self, dls, xformer, num_atten_heads):
        super(Transformers.TransformerFG.BasicDecoderWithMasking, self).__init__()
        self.dls = dls
        self.embedding_size = xformer.embedding_size
        self.max_seq_length = xformer.max_seq_length
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads
        self.self_attention_layer = xformer.SelfAttention(dls, xformer, num_atten_heads)
        self.norm1 = nn.LayerNorm(self.embedding_size)
        self.cross_attn_layer = xformer.CrossAttention(dls, xformer, num_atten_heads)
        self.norm2 = nn.LayerNorm(self.embedding_size)
        ## What follows are the linear layers for the FFN (Feed Forward Network) part of a BasicDecoder
        self.W1 =  nn.Linear( self.max_seq_length * self.embedding_size, self.max_seq_length * 2 * self.embedding_size )
        self.W2 =  nn.Linear( self.max_seq_length * 2 * self.embedding_size, self.max_seq_length * self.embedding_size )
        self.norm3 = nn.LayerNorm(self.embedding_size)

    def forward(self, sentence_tensor, final_encoder_out, mask):
        ## self attention
        masked_sentence_tensor = sentence_tensor
        if mask is not None:
            masked_sentence_sentence = self.apply_mask(sentence_tensor, mask, self.max_seq_length, self.embedding_size)
        Z_concatenated = self.self_attention_layer(masked_sentence_tensor).to(self.dls.device)
        Z_out = self.norm1(Z_concatenated + masked_sentence_tensor)
        ## for cross attention
        Z_out2  = self.cross_attn_layer( Z_out, final_encoder_out).to(self.dls.device)
        Z_out2 = self.norm2( Z_out2 )
        ## for FFN:
        basic_decoder_out =  nn.ReLU()(self.W1( Z_out2.view(sentence_tensor.shape[0],-1) ))
        basic_decoder_out =  self.W2( basic_decoder_out )
        basic_decoder_out = basic_decoder_out.view(sentence_tensor.shape[0], self.max_seq_length, self.embedding_size )
        basic_decoder_out =  basic_decoder_out  + Z_out2
        basic_decoder_out = self.norm3( basic_decoder_out )
        return basic_decoder_out

    def apply_mask(self, sentence_tensor, mask, max_seq_length, embedding_size):
        out = torch.zeros(sentence_tensor.shape[0], max_seq_length, embedding_size).float().to(self.dls.device)
        out[:,:,:len(mask)] = sentence_tensor[:,:,:len(mask)]
        return out
```

Outline

Master Decoder

- The primary job of the Master Decoder is to orchestrate the invocation of a stack of `BasicDecoderWithMasking` instances. The number of `BasicDecoderWithMasking` instances used is a user-defined parameter.
- The masking that is used in each `BasicDecoderWithMasking` instance is set here by the Master Decoder.
- In Line (B) on the next slide, we define the `BasicDecoderWithMasking` instances needed. The linear layer in Line (C) is needed because what the decoder side produces must ultimately be mapped as a probability distribution over the entire vocabulary for the target language.
- With regard to the data flow through the network, note how the mask is initialized in Line (D). The mask is supposed to be a vector of one's that grows with the prediction for each output word. We start by setting it equal to just a single-element vector containing a single "1".

`MasterDecoderWithMasking` (contd.)

- Lines (E) and (F) in the code on the next slide declare the tensors that will store the final output of the Master Decoder. This final output consists of two tensors:
    - One tensor holds the integer index to the target-language vocabulary word where the output log-prob is maximum. [This index is needed at inference time to output the words in the translation.]
    - The other tensor holds the log-probs over the target language vocabulary. The log-probs are produced by the nn.LogSoftmax in Line (L).

MasterDecoderWithMasking (contd.)

```
class MasterDecoderWithMasking(nn.Module):

    def __init__(self, dls, xformer, how_many_basic_decoders, num_atten_heads):
        super(Transformers.TransformerFG.MasterDecoderWithMasking, self).__init__()
        self.dls = dls
        self.max_seq_length = xformer.max_seq_length
        self.embedding_size = xformer.embedding_size
        self.target_vocab_size = xformer.vocab_es_size                                                    ## (A)
        self.basic_decoder_arr = nn.ModuleList([xformer.BasicDecoderWithMasking( dls, xformer,
                                                num_atten_heads) for _ in range(how_many_basic_decoders)])  ## (B)
        ## Need the following layer because we want the prediction of each target word to be a probability
        ## distribution over the target vocabulary. The conversion to probs would be done by the criterion
        ## nn.CrossEntropyLoss in the training loop:
        self.out = nn.Linear(self.embedding_size, self.target_vocab_size)                                 ## (C)

    def forward(self, sentence_tensor, final_encoder_out):                                               ## (D)
        ##  This part is for training:
        mask = torch.ones(1, dtype=int)                                                                   ## (E)
        ##  A tensor with two axes, one for the batch instance and the other for storing the predicted
        ##  word ints for that batch instance:
        predicted_word_index_values = torch.ones(sentence_tensor.shape[0], self.max_seq_length,
                                                              dtype=torch.long).to(self.dls.device)       ## (F)
        ##  A tensor with two axes, one for the batch instance and the other for storing the log-prob
        ##  of predictions for that batch instance. The log_probs for each predicted word over the entire
        ##  target vocabulary:
        predicted_word_logprobs  = torch.zeros( sentence_tensor.shape[0], self.max_seq_length,
                                            self.target_vocab_size, dtype=float).to(self.dls.device)      ## (G)
        for mask_index in range(1, sentence_tensor.shape[1]):
            masked_target_sentence = self.apply_mask(sentence_tensor, mask, self.max_seq_length,
                                                                    self.embedding_size)                  ## (H)
            ## out_tensor will start as just the first word, then two first words, etc.
            out_tensor = masked_target_sentence                                                           ## (I)
            for i in range(len(self.basic_decoder_arr)):                                                  ## (J)
                out_tensor = self.basic_decoder_arr[i](out_tensor, final_encoder_out, mask)
            last_word_tensor = out_tensor[:,mask_index]                                                   ## (K)
            last_word_onehot = self.out(last_word_tensor.view(sentence_tensor.shape[0],-1))               ## (L)
            output_word_logprobs = nn.LogSoftmax(dim=1)(last_word_onehot)                                 ## (M)
            _, idx_max = torch.max(output_word_logprobs, 1)                                               ## (N)
            predicted_word_index_values[:,mask_index] = idx_max                                           ## (P)
            predicted_word_logprobs[:,mask_index] = output_word_logprobs                                  ## (Q)
            mask = torch.cat( ( mask, torch.ones(1, dtype=int) ) )                                        ## (R)
        return predicted_word_logprobs, predicted_word_index_values                                       ## (S)

    def apply_mask(self, sentence_tensor, mask, max_seq_length, embedding_size):
        out = torch.zeros_like(sentence_tensor).float().to(self.dls.device)
        out[:,:len(mask),:] = sentence_tensor[:,:len(mask),:]
        return out
```

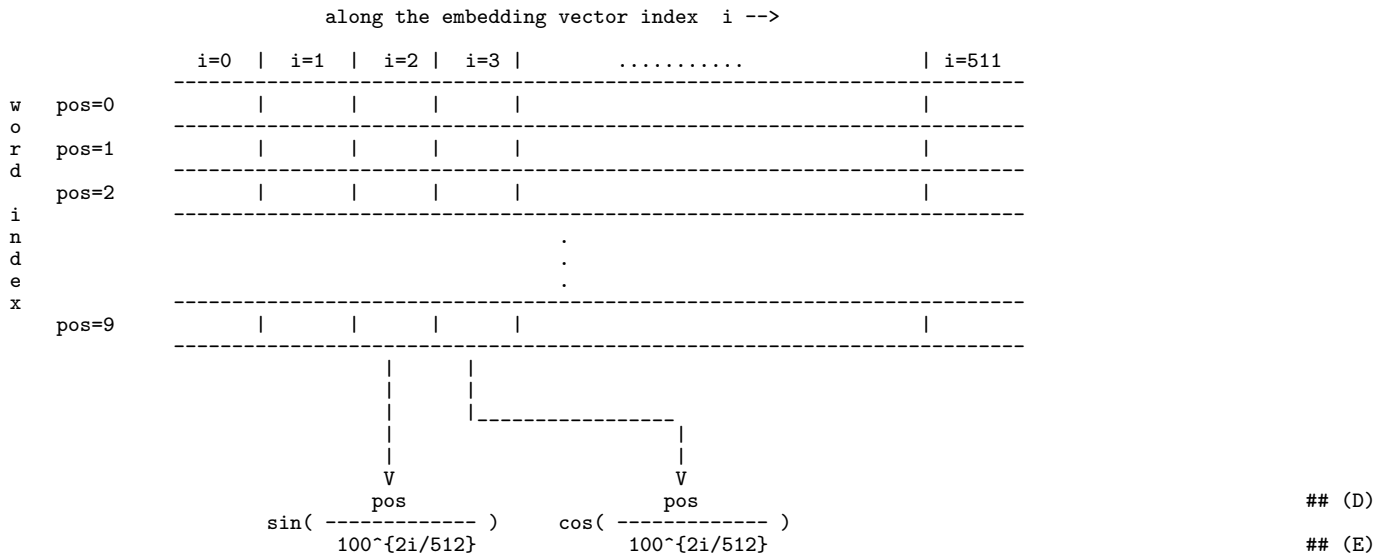
Outline

Positional Encoding for the Words

- The main goal of positional encoding is to sensitize a neural network to the position of each word in a sentence and also to each embedding-vector cell for each word.
- Positional encoding can be achieved by first constructing an array of floating-point values as illustrated on the next slide and then adding that array of numbers to the sentence tensor.
- The alternating columns of the 2D array shown on the next slide are filled using sine and cosine functions whose periodicities vary with the column index in the pattern.
- Note that whereas the periodicities are column-specific, the values of the sine and cosine functions are word-position-specific. In the depiction shown on the next slide, each row is an embedding vector for a specific word.

Positional Encoding (contd.)

- In the pattern shown below to illustrate positional encoding, I am assuming that the size of the word embedding vectors is 512 and that we have a max of 10 words in the input sentence.



- In this case, the sentence tensor is of shape $[10, 512]$. So the array of positional-encoding numbers we need to construct will also be of shape $[10, 512]$. We need to fill the alternating columns of this $[10, 512]$ array with *sin*() and *cos*() values as shown above.

Positional Encoding (contd.)

- To appreciate the significance of the values shown on the previous slide, first note that one period of a sinusoidal function like $sin(pos)$ is $2 * \pi$ with respect to the word index pos. That would amount to only about six words. That is, there would only be roughly six words in one period if we just use $sin(pos)$ for the positional indexing needed for the pattern shown on the previous slide.
- On the other hand, one period of a sinusoidal function like $sin(pos/k)$ is $2 * pi * k$ with respect to the word index pos. So if k=100, we have a periodicity of about 640 word positions along the pos axis.
- The important point is that every individual column in the 2D pattern shown above gets a unique periodicity and that the alternating columns are characterized by sine and cosine functions.
- Shown on the next slide is the function in DLStudio's `Transformers` class that implements positional encoding.

Positional Encoding (contd.)

```
def apply_positional_encoding(self, sentence_tensor):
    position_encodings = torch.zeros_like( sentence_tensor,  dtype=float )
    ## Calling unsqueeze() with arg 1 causes the "row tensor" to turn into a "column tensor"
    ##    which is needed in the products in lines (F) and (G). We create a 2D pattern by
    ##    taking advantage of how PyTorch has overloaded the definition of the infix '*'
    ##    tensor-tensor multiplication operator.  It in effect creates an output-product of
    ##    of what is essentially a column vector with what is essentially a row vector.
    word_positions = torch.arange(0, self.max_seq_length).unsqueeze(1)
    div_term =  1.0 / (100.0 ** ( 2.0 * torch.arange(0,
                                          self.embedding_size, 2) / float(self.embedding_size) ))
    position_encodings[:, :, 0::2] =  torch.sin(word_positions * div_term)
    position_encodings[:, :, 1::2] =  torch.cos(word_positions * div_term)
    return sentence_tensor + position_encodings
```

Outline

The Two Transformer Classes in DLStudio

- Everything I have said so far in this lecture is for the transformers as originally envisioned in the much celebrated Vaswani et el. paper. In DLStudio, my implementation for that architecture is in the class `TransformerFG` where the suffix "FG" stands for "First Generation".

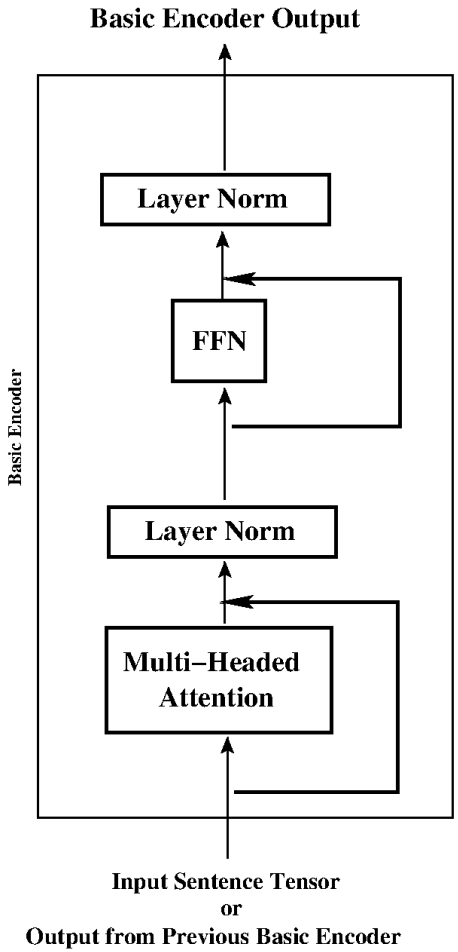
- Authors who followed that original publication observed that the Vaswani et el. architecture was difficult to train and that was the reason why it required a carefully designed "warm-up" phase during training in which the learning-rate was at first increased very slowly and then decreased again.

- In particular, it was observed by by Xiong et al. in their paper "On Layer Normalization in the Transformer Architecture" that using `LayerNorm` *after* each residual connection in the Vaswani et al. design contributed significantly to the stability of the learning process.

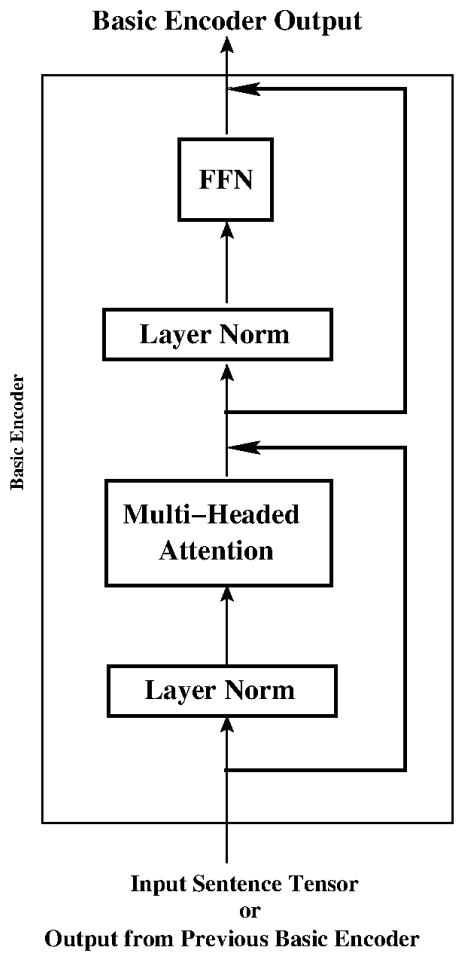
TransformerFG vs. TransformerPreLN

- Xiong et al. advocated changing the point at which the LayerNorm is invoked in the original design. In the two diagrams shown on the next slide, the one at left is for the encoder layout in TransformerFG and the one on right for the same in TransformerPreLN for the design proposed by Xiong et al.
- As you can see in the diagrams, in TransformerFG, each of the two components in the BasicEncoder — Self Attention and FFN — is followed with a residual connection that wraps around the component. That is, in TransformerFG, the residual connection is followed by LayerNorm.
- On the other hand, in TransformerPreLN, the LayerNorm for each component is used prior to the component and the residual connection wraps around both the LayerNorm layer and the component, as shown at right below.

TransformerFG vs. TransformerPreLN (contd.)



TransformerFG



TransformerPreLN

TransformerFG vs. TransformerPreLN (contd.)

- While the the difference between TransformerFG and TransformerPreLN depicted in the diagram on the previous slide specifically addresses the basic encoder, the same difference carries over to the decoder side.
- In TransformerPreLN, inside each Basic Decoder, you will have three invocations of LayerNorm, one before the Self-Attention layer, another one before the call to Cross-Attention and, finally, one more application of LayerNorm prior to the FFN layer.

Outline

Results on the English-Spanish Dataset

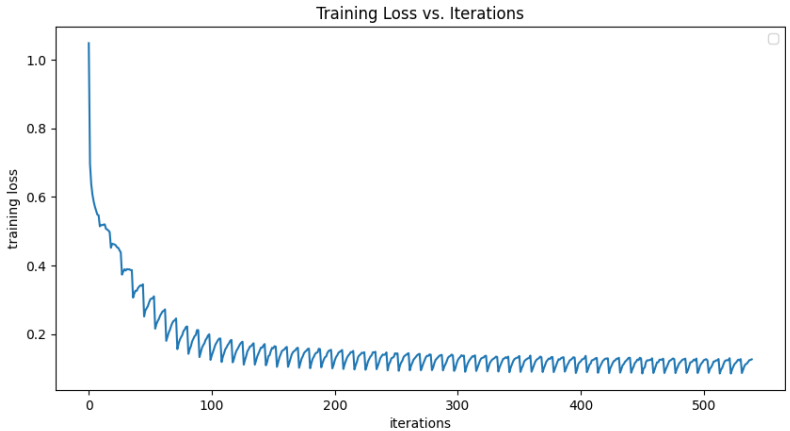


Figure: Training loss vs. iterations for 20 epochs with the TransformerFG class in DLStudio.

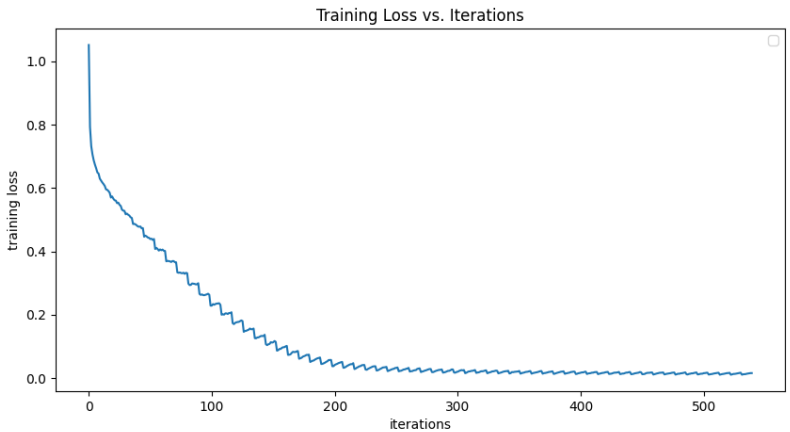


Figure: Training loss vs. iterations for 60 epochs with the TransformerPreLN class in DLStudio.

Translations Produced by `TransformerFG`

- Show below and on the next two slides are the results produced by `TransformerFG` class on 20 randomly selected sentences from the English-Spanish dataset mentioned earlier after 20 epochs of training:

```
Size of the English vocab in the dataset:  11258
Size of the Spanish vocab in the dataset:  21823

The number of learnable parameters in the Master Encoder: 124583936
The number of layers in the Master Encoder: 128

The number of learnable parameters in the Master Decoder: 149886015
The number of layers in the Master Decoder: 234


1:  The input sentence pair:  ['SOS anybody can read it EOS'] ['SOS cualquiera puede leerlo EOS']

    The translation produced by TransformerFG:  EOS cualquiera puede hacerlo EOS EOS EOS EOS EOS EOS            [PARTIALLY CORRECT]


2:  The input sentence pair:  ['SOS is he your teacher EOS'] ['SOS es tu profesor EOS']

    The translation produced by TransformerFG:  EOS él es profesor profesor EOS EOS EOS EOS EOS                [WRONG]


3:  The input sentence pair:  ['SOS i wanted to study french EOS'] ['SOS quería estudiar francés EOS']

    The translation produced by TransformerFG:  EOS quería estudiar francés EOS EOS EOS EOS EOS EOS          [CORRECT]


4:  The input sentence pair:  ['SOS what are you doing next monday EOS'] ['SOS qué vas a hacer el próximo lunes EOS']

    The translation produced by TransformerFG:  EOS qué vas a hacer el próximo lunes EOS EOS                 [CORRECT]


5:  The input sentence pair:  ['SOS it was a beautiful wedding EOS'] ['SOS fue un hermoso casamiento EOS']

    The translation produced by TransformerFG:  EOS fue un hermoso hermosa EOS EOS EOS EOS EOS               [WRONG]
```


(Continued on the next slide ...)

Translations Produced by TransformerFG (contd.)

(..... continued from the previous slide)

6: The input sentence pair: ['SOS there were two glasses under the mirror EOS'] ['SOS bajo el espejo había dos vasos EOS']
The translation produced by TransformerFG: EOS bajo el espejo había dos vasos EOS EOS EOS [CORRECT]

7: The input sentence pair: ['SOS he has a very interesting book EOS'] ['SOS él tiene un libro muy divertido EOS']
The translation produced by TransformerFG: EOS él tiene un libro muy divertido EOS EOS EOS [CORRECT]

8: The input sentence pair: ['SOS i was waiting for tom EOS'] ['SOS estaba esperando a tom EOS']
The translation produced by TransformerFG: EOS estaba esperando a tom EOS EOS EOS EOS EOS [CORRECT]

9: The input sentence pair: ['SOS mary has curlers in her hair EOS'] ['SOS mary lleva rulos en el pelo EOS']
The translation produced by TransformerFG: EOS mary lleva nada en el pelo EOS EOS EOS [PARTIALLY CORRECT]

10: The input sentence pair: ['SOS tom thought about mary a lot EOS'] ['SOS tom pensó mucho acerca de maría EOS']
The translation produced by TransformerFG: EOS tom pensó mucho acerca de maría EOS EOS EOS [CORRECT]

11: The input sentence pair: ['SOS you are so shallow EOS'] ['SOS eres tan superficial EOS']
The translation produced by TransformerFG: EOS eres tan tan EOS EOS EOS EOS EOS EOS [WRONG]

12: The input sentence pair: ['SOS can you solve this problem EOS'] ['SOS podéis resolver este problema EOS']
The translation produced by TransformerFG: EOS puede resolver este problema EOS EOS EOS EOS EOS [CORRECT]

13: The input sentence pair: ['SOS they were listening to the radio EOS'] ['SOS ellos estaban escuchando la radio EOS']
The translation produced by TransformerFG: EOS ellos estaban escuchando la radio EOS EOS EOS EOS [CORRECT]

14: The input sentence pair: ['SOS come right in EOS'] ['SOS ven adentro EOS']
The translation produced by TransformerFG: EOS entra adentro EOS EOS EOS EOS EOS EOS EOS [CORRECT]

Translations Produced by TransformerFG (contd.)

(..... continued from the previous slide)

15: The input sentence pair: ['SOS when did you learn to swim EOS'] ['SOS cuándo aprendiste a nadar EOS']
The translation produced by TransformerFG: EOS cuándo aprendiste a nadar EOS EOS EOS EOS EOS [CORRECT]

16: The input sentence pair: ['SOS tom has been busy all morning EOS'] ['SOS tom estuvo ocupado toda la mañana EOS']
The translation produced by TransformerFG: EOS tom ha estado ocupado toda la mañana EOS EOS [CORRECT]

17: The input sentence pair: ['SOS i just want to read EOS'] ['SOS solo quiero leer EOS']
The translation produced by TransformerFG: EOS solo quiero leer EOS EOS EOS EOS EOS EOS [CORRECT]

18: The input sentence pair: ['SOS tell us something EOS'] ['SOS díganos algo EOS']
The translation produced by TransformerFG: EOS EOS algo EOS EOS EOS EOS EOS EOS EOS [WRONG]

19: The input sentence pair: ['SOS how often does tom play hockey EOS'] ['SOS con qué frecuencia juega tom al hockey EOS']
The translation produced by TransformerFG: EOS con qué frecuencia juega con al EOS EOS EOS [PARTIALLY CORRECT]

20: The input sentence pair: ['SOS he was reelected mayor EOS'] ['SOS él fue reelegido alcalde EOS']
The translation produced by TransformerFG: EOS él fue a alcalde EOS EOS EOS EOS EOS [PARTIALLY CORRECT]

Translations Produced by TransformerPreLN

- Show below and on the next three slides are the results produced by TransformerPreLN class on the same 20 randomly selected sentences from the English-Spanish dataset that I used for the results obtained with the TransformerFG class. The results that follow were obtained after 60 epochs of training:
- CONCLUSION: Comparing the results shown earlier as obtained with TransformerFG and the results shown below with TransformerPreLN, the former is clearly superior — at least based on this admittedly unscientific comparison.

```
Size of the English vocab in the dataset:  11258
Size of the Spanish vocab in the dataset:  21823

The number of learnable parameters in the Master Encoder: 124583936
The number of layers in the Master Encoder: 128

The number of learnable parameters in the Master Decoder: 149886015
The number of layers in the Master Decoder: 234



1:  The input sentence pair:  ['SOS anybody can read it EOS'] ['SOS cualquiera puede leerlo EOS']

    The translation produced by TransformerPreLN:  SOS cualquiera puede puede EOS EOS EOS EOS EOS EOS                    [WRONG]
```


(Continued on the next slide .....)

Translations Produced by TransformerPreLN (contd.)

(..... continued from the previous slide)

2: The input sentence pair: ['SOS is he your teacher EOS'] ['SOS es tu profesor EOS']
The translation produced by TransformerPreLN: SOS él es vuestro profesor EOS EOS EOS EOS EOS [CORRECT]

3: The input sentence pair: ['SOS i wanted to study french EOS'] ['SOS quería estudiar francés EOS']
The translation produced by TransformerPreLN: SOS quería estudiar francés EOS EOS EOS EOS EOS EOS [CORRECT]

4: The input sentence pair: ['SOS what are you doing next monday EOS'] ['SOS qué vas a hacer el próximo lunes EOS']
The translation produced by TransformerPreLN: SOS qué harás a va el EOS EOS EOS EOS [WRONG]

5: The input sentence pair: ['SOS it was a beautiful wedding EOS'] ['SOS fue un hermoso casamiento EOS']
The translation produced by TransformerPreLN: SOS fue un hermoso agresivo EOS EOS EOS EOS EOS [WRONG]

6: The input sentence pair: ['SOS there were two glasses under the mirror EOS'] ['SOS bajo el espejo había dos vasos EOS']
The translation produced by TransformerPreLN: SOS bajo dos espejo cáncer EOS EOS EOS EOS EOS [WRONG]

7: The input sentence pair: ['SOS he has a very interesting book EOS'] ['SOS él tiene un libro muy divertido EOS']
The translation produced by TransformerPreLN: SOS él tiene un muy muy EOS EOS EOS EOS [WRONG]

8: The input sentence pair: ['SOS i was waiting for tom EOS'] ['SOS estaba esperando a tom EOS']
The translation produced by TransformerPreLN: SOS estaba esperando por tom EOS EOS EOS EOS EOS [PARTIALY CORRECT]

9: The input sentence pair: ['SOS mary has curlers in her hair EOS'] ['SOS mary lleva rulos en el pelo EOS']
The translation produced by TransformerPreLN: SOS mary lleva rulos pestañas en el EOS EOS EOS [WRONG]

(Continued on the next slide .....)

Translations Produced by TransformerPreLN (contd.)

(..... continued from the previous slide)

10: The input sentence pair: ['SOS tom thought about mary a lot EOS'] ['SOS tom pensó mucho acerca de maría EOS']
The translation produced by TransformerPreLN: SOS tom abuela había acerca de EOS EOS EOS EOS [WRONG]

11: The input sentence pair: ['SOS you are so shallow EOS'] ['SOS eres tan superficial EOS']
The translation produced by TransformerPreLN: SOS eres tan EOS EOS EOS EOS EOS EOS EOS [WRONG]

12: The input sentence pair: ['SOS can you solve this problem EOS'] ['SOS podéis resolver este problema EOS']
The translation produced by TransformerPreLN: SOS puede resolver este problema EOS EOS EOS EOS EOS [CORRECT]

13: The input sentence pair: ['SOS they were listening to the radio EOS'] ['SOS ellos estaban escuchando la radio EOS']
The translation produced by TransformerPreLN: SOS ellos escuchando en al por EOS EOS EOS EOS [WRONG]

14: The input sentence pair: ['SOS come right in EOS'] ['SOS ven adentro EOS']
The translation produced by TransformerPreLN: SOS entra adentro EOS EOS EOS EOS EOS EOS EOS [CORRECT]

15: The input sentence pair: ['SOS when did you learn to swim EOS'] ['SOS cuándo aprendiste a nadar EOS']
The translation produced by TransformerPreLN: SOS cuándo aprendiste a EOS EOS EOS EOS EOS EOS [PARTIALLY CORRECT]

16: The input sentence pair: ['SOS tom has been busy all morning EOS'] ['SOS tom estuvo ocupado toda la mañana EOS']
The translation produced by TransformerPreLN: SOS tom estado ocupado toda todo el EOS EOS EOS [PARTIALLY CORRECT]

17: The input sentence pair: ['SOS i just want to read EOS'] ['SOS solo quiero leer EOS']
The translation produced by TransformerPreLN: SOS solo quiero leer EOS EOS EOS EOS EOS EOS [CORRECT]

(Continued on the next slide .....)

Translations Produced by TransformerPreLN (contd.)

(..... continued from the previous slide)

```
18:  The input sentence pair:  ['SOS tell us something EOS'] ['SOS díganos algo EOS']
     The translation produced by TransformerPreLN:  SOS díganos algo EOS EOS EOS EOS EOS EOS EOS                    [CORRECT]

19:  The input sentence pair:  ['SOS how often does tom play hockey EOS'] ['SOS con qué frecuencia juega tom al hockey EOS']
     The translation produced by TransformerPreLN:  SOS con qué mañana cuesta tom al EOS EOS EOS                    [WRONG]

20:  The input sentence pair:  ['SOS he was reelected mayor EOS'] ['SOS él fue reelegido alcalde EOS']
     The translation produced by TransformerPreLN:  SOS él nubes reelegido entre EOS EOS EOS EOS EOS               [WRONG]
```