

# Transformers: Learning with Purely Attention Based Networks

Lecture Notes on Deep Learning

**Avi Kak and Charles Bouman**

Purdue University

Friday 5<sup>th</sup> May, 2023 10:48

©2023 A. C. Kak, Purdue University

# 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 al. 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*”. Again, a neural network endowed with self-attention should be able to make that connection.

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 and each phrase 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 DLStudio that 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” in `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 al.:

<https://arxiv.org/pdf/1706.03762.pdf>

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

<https://arxiv.org/pdf/2002.04745.pdf>

## Preamble (contd.)

The two classes `TransformerFG` and `TransformerPreLN` are defined in the module file `Transformers.py` in DLStudio.

About the dataset I'll be using to demonstrate transformers, 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.

The following 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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81



# Outline

1	<b>The Basic Idea of Dot-Product Attention</b>	<b>9</b>
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

# 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 as tensors in each of the three categories, 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 then 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 are created **neither** by convolution **nor** 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:

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

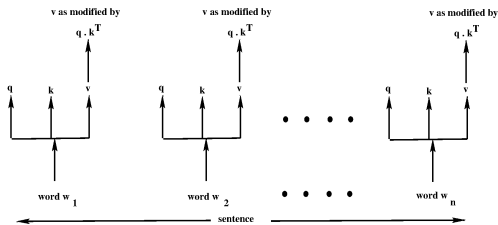
where  $w$  is a vector of numbers that numerically represents a word in the input sentence.

- 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.

[This figure is somewhat misleading because it does NOT show the  $q$  for one word engaged in a dot-product with the  $k$  of another word. This issue disappears in the tensor formulation you will see next.]



## 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

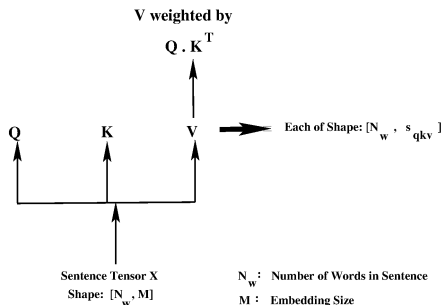
- Calculation of the sentence-level Query, Key, and Value tensors can be expressed more *accurately* and compactly as

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

- 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  $W_Q$ ,  $W_K$ , and  $W_V$ .
- 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  $k_1, k_2, k_3, \dots, k_{N_w}$  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:

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

The `nn.Softmax` is applied along the word axis (Axis 1).

- 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  $N$ .

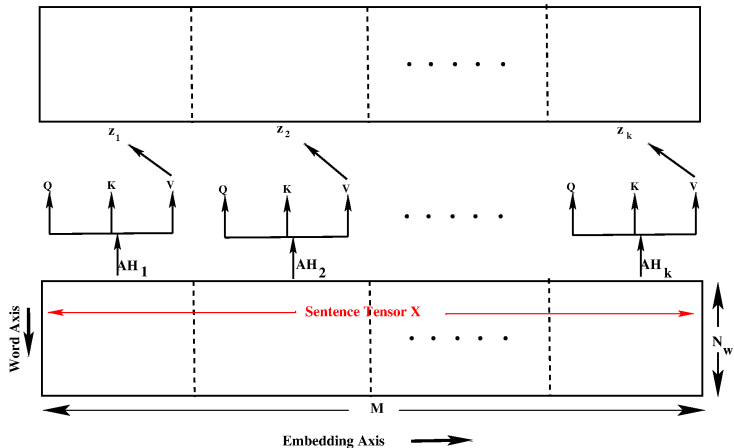
# Outline

1	The Basic Idea of Dot-Product Attention	9
2	<b>Multi-Headed Attention</b>	<b>19</b>
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

# 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  $N_H$  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.)



**Figure:** **Correction:** In the upper part of the figure, read  $Z_K$  as  $Z_{N_H}$ . And, in the middle of the figure, read  $AH_k$  as  $AH_{N_H}$ . The symbol  $N_H$  stands for the number of Attention Heads used.

## 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

$$s_{qkv} = \frac{M}{N_H} \quad (4)$$

- 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 of 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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	<b>Implementation of Attention in DLStudio's Transformers</b>	<b>23</b>
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

## Attention Head Implementation

- The next slide shows the implementation of the `AttentionHead` in the two transformer classes, `TransformerFG` and `TransformerPreLN`, in DLStudio.
- 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:

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

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



# Attention Head Class in DLStudio's Transformers Class

```

class AttentionHead(nn.Module):

    def __init__(self, dl_studio, max_seq_length, qkv_size, num_attn_heads):
        super(TransformerFG.AttentionHead, 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 )
        self.WK = nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )
        self.WV = nn.Linear( max_seq_length * self.qkv_size, max_seq_length * self.qkv_size )
        self.softmax = nn.Softmax(dim=1)

    def forward(self, sent_embed_slice):
        Q = self.WQ( sent_embed_slice.reshape(sent_embed_slice.shape[0],-1).float() )
        K = self.WK( sent_embed_slice.reshape(sent_embed_slice.shape[0],-1).float() )
        V = self.WV( sent_embed_slice.reshape(sent_embed_slice.shape[0],-1).float() )
        Q = Q.view(sent_embed_slice.shape[0], self.max_seq_length, self.qkv_size)
        K = K.view(sent_embed_slice.shape[0], self.max_seq_length, self.qkv_size)
        V = V.view(sent_embed_slice.shape[0], self.max_seq_length, self.qkv_size)
        A = K.transpose(2,1)
        QK_dot_prod = Q @ A
        rowwise_softmax_normalizations = self.softmax( QK_dot_prod )
        Z = rowwise_softmax_normalizations @ V
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(self.dl_studio.device)
        Z = coeff * Z
        return Z

```

## (A)  
## (B)  
## (C)  
## (D)  
## (E)  
## (F)  
## (G)  
## (H)  
## (I)  
## (J)  
## (K)  
## (L)  
## (M)  
## (N)  
## (O)  
## (P)  
## (Q)  
## (R)

## Self-Attention in DLStudio's Transformers Co-Class

- The `AttentionHead` class on the previous slide is the building block in a `SelfAttention` layer that concatenates the outputs from all the `AttentionHead` instances and presents the result as its own output.
- In the code shown for `SelfAttention` in Slide 28, 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) will be of shape  $(B, N_w, M)$  where  $B$  is the batch size.
- As explained earlier, this tensor is sliced off into `num_attn_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  $M/\text{num\_attn\_heads}$ . 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 applies to all of the components of a transformer.

## Self Attention (contd.)

```

class SelfAttention(nn.Module):

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

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

        return concat_out_from_attn_heads

```

# Outline

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	<b>The Encoder-Decoder Architecture of a Transformer</b>	<b>29</b>
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

# 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` module. I refer to each individual encoder insider the Master Encoder as a Basic Encoder that is an instances of the class `BasicEncoder`.

## The Transformer Architecture (contd.)

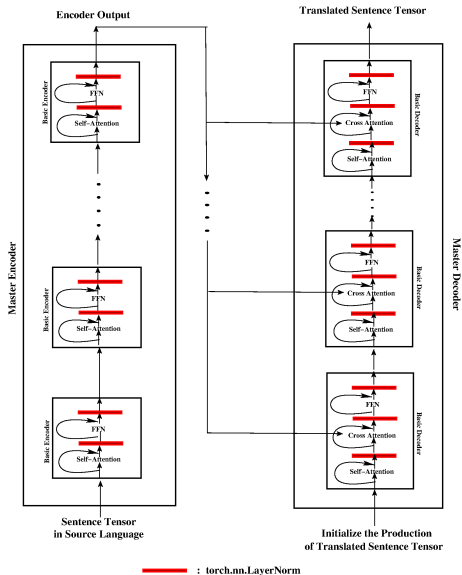
- Similarly, on the decoder side, I refer to the overall decoder as the Master Decoder that is implemented in the class `MasterDecoder`. I refer to each decoder in the Master Decoder as a Basic Decoder that I have implemented with the class `BasicDecoder`.
- The implementation classes mentioned above are 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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
<b>5</b>	<b>The Master Encoder Class</b>	<b>35</b>
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

# 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(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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
<b>6</b>	<b>The Basic Encoder Class</b>	<b>37</b>
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

## Basic Encoder

- The `BasicEncoder` consists of a layer of self-attention (SA) followed by a purely feed-forward layer (FFN). You already know 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_attn_heads):
        super(TransformerFG.BasicEncoder, self).__init__()
        self.dls = dls
        self.embedding_size = xformer.embedding_size
        self.max_seq_length = xformer.max_seq_length
        self.num_attn_heads = num_attn_heads
        self.self_attention_layer = xformer.SelfAttention(dls, xformer, num_attn_heads)
        self.norm1 = nn.LayerNorm(self.embedding_size)
        ## 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)

    def forward(self, sentence_tensor):
        sentence_tensor = sentence_tensor.float()
        self_attn_out = self.self_attention_layer(sentence_tensor).to(self.dls.device)
        normed_attn_out = self.norm1(self_attn_out + sentence_tensor)
        basic_encoder_out = nn.ReLU()(self.W1(normed_attn_out.view(sentence_tensor.shape[0], -1)))
        basic_encoder_out = self.W2(basic_encoder_out)
        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_attn_out)
        return basic_encoder_out
```

# Outline

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
<b>7</b>	<b>Cross Attention</b>	<b>40</b>
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81



# 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 then 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 the `CrossAttentionHead` instances for the calculation of the dot products and, subsequently, for the output of the dot products 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_attn_heads):
        super(TransformerFG.CrossAttention, self).__init__()
        self.dl_studio = dls
        self.max_seq_length = xformer.max_seq_length
        self.embedding_size = xformer.embedding_size
        self.num_attn_heads = num_attn_heads
        self.qkv_size = self.embedding_size // num_attn_heads
        self.attention_heads_arr = nn.ModuleList( [xformer.CrossAttentionHead(dls,
            self.max_seq_length, self.qkv_size, num_attn_heads) for _ in range(num_attn_heads)] )

    def forward(self, basic_decoder_out, final_encoder_out):
        concat_out_from_attn_heads = torch.zeros( basic_decoder_out.shape[0], self.max_seq_length,
            self.num_attn_heads * self.qkv_size).float()

        for i in range(self.num_attn_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_attn_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_attn_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:

$$Q = X_{target} \cdot W_Q \quad K = X_{source} \cdot W_K \quad V = X_{source} \cdot W_V \quad (5)$$

## 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:

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

```
class CrossAttentionHead(nn.Module):

    def __init__(self, dl_studio, max_seq_length, qkv_size, num_attn_heads):
        super(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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
<b>8</b>	<b>The Basic Decoder Class</b>	<b>46</b>
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

## 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 transformer 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_attn_heads):
        super(TransformerFG.BasicDecoderWithMasking, self).__init__()
        self.dls = dls
        self.embedding_size = xformer.embedding_size
        self.max_seq_length = xformer.max_seq_length
        self.num_attn_heads = num_attn_heads
        self.qkv_size = self.embedding_size // num_attn_heads
        self.self_attention_layer = xformer.SelfAttention(dls, xformer, num_attn_heads)
        self.norm1 = nn.LayerNorm(self.embedding_size)
        self.cross_attn_layer = xformer.CrossAttention(dls, xformer, num_attn_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_tensor = 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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
<b>9</b>	<b>The Master Decoder Class</b>	<b>49</b>
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

# 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_attn_heads):
        super(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_size
        self.basic_decoder_arr = nn.ModuleList([xformer.BasicDecoderWithMasking( dls, xformer,
                                                                                   num_attn_heads) for _ in range(how_many_basic_decoders)])

        ## 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)

    def forward(self, sentence_tensor, final_encoder_out):

        ## This part is for training:
        mask = torch.ones(1, dtype=int)

        ## 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)

        ## 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)

        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)

            ## out_tensor will start as just the first word, then two first words, etc.
            out_tensor = masked_target_sentence

            for i in range(len(self.basic_decoder_arr)):
                out_tensor = self.basic_decoder_arr[i](out_tensor, final_encoder_out, mask)

            last_word_tensor = out_tensor[:, mask_index]
            last_word_onehot = self.out(last_word_tensor.view(sentence_tensor.shape[0], -1))
            output_word_logprobs = nn.LogSoftmax(dim=-1)(last_word_onehot)

            _, idx_max = torch.max(output_word_logprobs, 1)
            predicted_word_index_values[:, mask_index] = idx_max
            predicted_word_logprobs[:, mask_index] = output_word_logprobs

            mask = torch.cat( ( mask, torch.ones(1, dtype=int) ) )

        return predicted_word_logprobs, predicted_word_index_values

    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

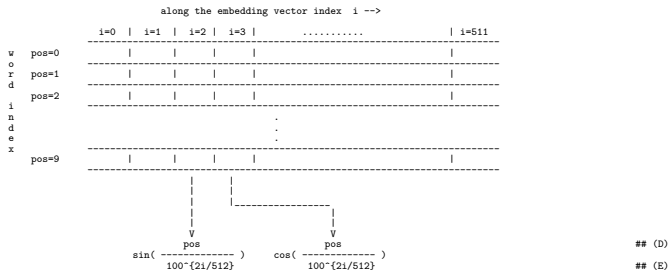
1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
<b>10</b>	<b>Positional Encoding for the Words</b>	<b>53</b>
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

## 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 transformer code 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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	<b>TransformerFG and TransformerPreLN Classes in DLStudio</b>	<b>58</b>
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

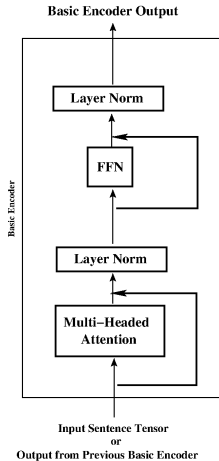
# The Two Transformer Classes in DLStudio

- Everything I have said so far in this lecture is for the transformers as originally envisioned in the much celebrated Vaswani et al. 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 al. 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 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.

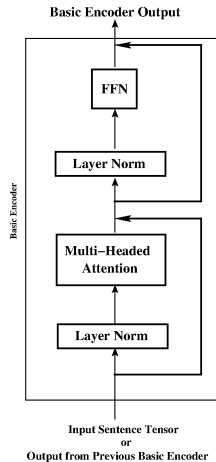
## TransformerFG **VS.** TransformerPreLN

- 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 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

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	<b>Regarding the Difficulty of Training a Transformer Network</b>	<b>63</b>
13	Results on the English-Spanish Dataset	70
14	Transformers with Generative Pre-Training (GPT)	81

# Training Transformer Networks and the Sudden Model Divergence

- Transformers, in general, are difficult to train and that's especially the case with TransformerFG. Using the same learning rate throughout the training process either results in excessively slow learning if the learning-rate is too small, **or unstable learning if the learning-rate is not small enough.**
- When transformer learning becomes unstable, you get what's known as **sudden model divergence**, which means roughly the same thing as mode collapse for the case of training a GAN.
- As you are training a transformer model, you would want to use some metric to measure the performance of the current state of the model so that you can be sure that the model is still learning and that it has not suddenly regressed into a divergence. Obviously, for such a check on the model, you would use an assortment of sentence pairs drawn from the corpus.



## BLEU for Measuring Checkpoint Performance

- DLStudio makes it easier to carry out such checks through the checkpoints it writes out to the disk memory every 5 epochs. You can then apply the very popular BLEU metric to the checkpoints. You have model divergence when the value returned by this metric stays at 0. BLEU stands for “BiLingual Evaluation Understudy”.
- BLEU score measures the performance of a language translation framework by measuring the frequencies of the  $n$ -grams in the predicted sentences in the target language for the  $n$ -grams that exist in the ground-truth sentences. As you know, an  $n$ -gram is a sequence of consecutively occurring words — the qualifier  $n$  refers to the length of the sequence.
- Given a sentence pair, one predicted and the other the target, for a given value of  $n$ , BLEU counts the number of  $n$ -grams in the predicted sentence for each  $n$ -gram that exists in the target sentence for a set of  $n$  values.

## The BLEU Metric for Checkpoint Performance (contd.)

- When comparing the  $n$ -grams between the predicted and the target sentences, you do NOT seek a position based matching of the  $n$ -grams. For a given value of  $n$ , what BLEU calculates is the occurrence count for an  $n$ -gram in the predicted sentence that has a matching  $n$ -gram anywhere in the target sentence. The ratio of this number to the total number of such  $n$ -grams in the predicted sentence is the translation precision as measured for that  $n$ . Typically, one constructs a weighted average of these ratios for  $n \in \{1, 2, 3, 4\}$ .
- The above formula requires a critical modification in order to be effective: You do not want the occurrence based count for an  $n$ -gram in a predicted sentence to exceed the count for the same  $n$ -gram in the target sentence. [To cite an example provided by the original authors of BLEU, consider the case when the predicted sentence is a gibberish repetition of a commonly occurring word like “the” as in the predicted sentence “the the the the the the the”. Assume that the target sentence is “the cat is on the mat”. A unigram based precision in this case would return a value of  $\frac{7}{7} = 1$  since the unigram “the” occurs 7 times in the predicted sentence and it does occur at least once in the target sentence. To remedy this shortcoming, we require that the count returned for any  $n$ -gram not exceed the count for same  $n$ -gram in the target sentence. With that modification, the value returned for the example would be  $\frac{2}{7}$ . You would impose this constraint for all  $n$  in the  $n$ -grams used.]

## The BLEU Metric for Checkpoint Performance (contd.)

- Since the n-gram based counts are based solely on the predicted sentences (albeit on the basis that the same n-grams exist in the target sentences), predicted sentences much shorter than the target sentences will in general score higher. [Consider the case when the predicted sentence is “the cat is” for the target sentence “the cat is on the mat”. In this case, all of the unigram, digram, trigram based scores for the quality of the translation will be perfect.] To guard against, the BLEU metric multiplies the n-gram based scores with the factor  $e^{(1-\frac{r}{c})}$  when  $c < r$  where  $c$  is the length of the predicted sentence and  $r$  the length of the target sentence.
- You use the BLEU metric in your code by calling on its implementation provided by the Natural Language Toolkit (NLTK) library. If you wish, you can download the source code for the BLEU metric from:

[https://www.nltk.org/\\_modules/nltk/translate/bleu\\_score.html](https://www.nltk.org/_modules/nltk/translate/bleu_score.html)

## Examining the Checkpoints in DLStudio

- Assuming you are using TransformerFG, your checkpoints will be deposited every five epochs either in the directory `checkpoints_with_masking` or the directory `checkpoints_no_masking` depending on the choice made for the variable `masking` in a script like `seq2seq_with_TransformerFG`.
- Each checkpoint consists of two models, `encoder_N` and `decoder_N`, where  $N$  is the checkpoint index. Since the checkpoints are produced every 5 epochs, you'll have the following values for  $N$ :  $\{4, 9, 14, 19, 24, \dots\}$ .
- In order to evaluate a checkpoint, you would need to invoke one of the scripts, `test_checkpointFG.py` or `test_checkpointPreLN.py`, with the syntax:

```
python3 test_checkpointFG.py checkpoints_no_masking 24
```

where `test_checkpointFG.py` is a script that comes with the distro, the arg `checkpoints_no_masking` the name of the checkpoints directory, and the arg 24 means that you want to evaluate the checkpoint indexed 24 (meaning the checkpoint for the 25<sup>th</sup> epoch).

## Stabilizing the Learning for TransformerFG

- For the case of TransformerFG, the original authors of the paper on which TransformerFG is based showed that they could prevent model divergence by starting with a very small learning rates, say  $1e-9$ , and then ramping up linearly with each iteration of training.
- This is known as the **learning-rate warm-up** and it requires that you specify the number of training iterations for the warm-up phase. Typically, during this phase, you increment the learning rate linearly with the iteration index.
- Note that the more stable TransformerPreLN does NOT require a learning-rate warm-up — **because that transformer is inherently more stable. The price you pay for that stability is the much slower convergence of the model.**
- In my own rather informal and unscientific comparisons, the performance I get with about 40 epochs of TransformerFG takes more than 100 epochs with TransformerPreLN.

# Outline

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
<b>13</b>	<b>Results on the English-Spanish Dataset</b>	<b>70</b>
14	Transformers with Generative Pre-Training (GPT)	81

# Results on the English-Spanish Dataset



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

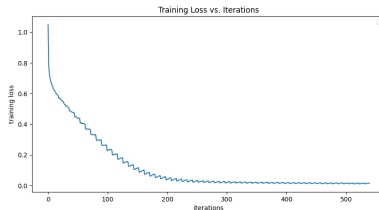


Figure: Training loss vs. iterations for 60 epochs with the TransformerPreLN class in DLStudio.

## Translations Produced by `TransformerFG`

- After 40 epochs of training with `TransformerFG` and with 90,000 pairs of English-Spanish sentences, what follows are the results produced on 20 randomly selected sentences from the dataset.
- The training was carried out on RVL Cloud using a single GPU (NVIDIA GeForce RTX 2080) and by executing the following command in the `ExamplesTransformers` directory of DLStudio:

```
python3 seq2seq_with_transformerFG.py
```

- Here are the parameters used for training the transformer network:

Batch size:	50
Embedding_size:	256
Number Basic Encoders:	4
Number Basic Decoders:	4
Number Attention Heads:	4
Number of Warmup Steps:	4000
Masking:	False



## Translations Produced by TransformerFG (contd.)

- And here is the timing performance:

Training time per 200 iterations:	167 seconds	
Training time per epoch:	9 * 167 seconds	= 25.05 minutes
Total training time for 40 epochs:	16 hours	

- The results shown starting with the next slide were produced by executing the command:

```
python3 test_checkpointFG.py checkpoints_no_masking_FG 39
```

where `test_checkpointFG.py` is a DLStudio supplied script in the `ExamplesTransformers` directory. The integer 39 means that you want to use the checkpoint indexed 39 that is stored in the checkpoints directory `checkpoints_no_masking_FG` of the same directory. This would be the checkpoint produced at the end of the 40<sup>th</sup> epoch of training.

- The results are shown starting with the next slide.

# Translations Produced by TransformerFG (contd.)

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

Number of sentence pairs in the dataset: 90000

No sentence is longer than 10 words (including the SOS and EOS tokens)

## TRANSLATIONS PRODUCED:

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

The translation produced by TransformerFG: EOS cualquiera puede leerlo EOS EOS EOS EOS EOS EOS [CORRECT]
- The input sentence pair: ['SOS is he your teacher EOS'] ['SOS es tu profesor EOS']

The translation produced by TransformerFG: EOS es tu profesor EOS EOS EOS EOS EOS EOS [CORRECT]
- 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]
- 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]
- 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 hermoso 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 tengo en el pelo EOS EOS EOS [WRONG]
10. The input sentence pair: ['SOS tom thought about mary a lot EOS'] ['SOS tom pensó mucho acerca de maria EOS']  
The translation produced by TransformerFG: EOS tom pensó mucho acerca de maria 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 superficial EOS EOS EOS EOS EOS EOS [CORRECT]
12. The input sentence pair: ['SOS can you solve this problem EOS'] ['SOS podéis resolver este problema EOS']  
The translation produced by TransformerFG: EOS puedes 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]

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

# Translations Produced by TransformerFG (contd.)

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

14. The input sentence pair: ['SOS come right in EOS'] ['SOS ven adentro EOS']  
The translation produced by TransformerFG: EOS entra aquí EOS EOS EOS EOS EOS EOS EOS EOS [Semantically CORRECT]
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 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 toda toda mañana EOS EOS EOS EOS [WRONG]
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 EOS [CORRECT]
18. The input sentence pair: ['SOS tell us something EOS'] ['SOS díganos algo EOS']  
The translation produced by TransformerFG: EOS dinos algo EOS EOS EOS EOS EOS EOS EOS EOS [Semantically 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 TransformerFG: EOS con qué frecuencia juega tom al hockey EOS EOS EOS [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 EOS EOS [WRONG]

## The Results Look Great — But What Does That Mean?

- On the basis of the quality of the translations shown on the previous three slides for a random collection of sentences, the results produced by the TransformerFG-based network look very impressive. **Does that mean that I have presented a viable solution for automatic English-to-Spanish translation?**
- The answer to the above question is: **Not by a long shot!**
- The most likely reason for the excellent results: **Overfitting of the model to the training data.**
- A dataset of just 90,000 sentence pairs is much too small to create a generalizable model given the overall complexity of the transformer network. [Despite the fact that my transformer network is small compared to the networks used in corporate labs, it still has around 300 million learnable parameters (see Slide 74). That's still too large a model for the available dataset. ]
- I could have gotten more “juice” out of my small dataset if I had also incorporated in the learning framework the commonly used step of tokenization as a front-end and trained the model with the tokens.

## The Results Look Great, But ... (contd.)

- The smallness of the dataset mentioned on the previous slide can also be measured by the size of the vocabulary. As shown on Slide 74, the English vocab has just 11,258 words. At the least you are going to need a vocabulary that's five times the size I have at my disposal if you want to train a model with any power of generalization. And I'm only talking about just ordinary conversational sentences.
- And that brings me to a fundamental challenge associated with developing deep-learning based solutions for novel problems, especially if the problems require complex models like those based on transformers: **The high cost of creating labeled datasets.**
- A possible solution to this challenge: Non-supervised pre-conditioning of the network with unlabeled data (that's always available in abundance), followed by using the available labeled data in discriminative learning for fine-tuning the learnable parameters for the task at hand.

## The Results Look Great, But ... (contd.)

- To follow up on the last bullet on the previous slide, here is an influential 2010 paper “*Why Does Unsupervised Pre-training Help Deep Learning?*” by Erhan et al. with this message:

<https://www.jmlr.org/papers/volume11/erhan10a/erhan10a.pdf>

- Here is a very insightful quote from this paper:

“In virtually all instances of deep learning, the objective function is a highly non-convex function of the parameters, with the potential for many distinct local minima in the model parameter space. The principal difficulty is that not all of these minima provide equivalent generalization errors and, we suggest, that for deep architectures, the standard training schemes (based on random initialization) tend to place the parameters in regions of the parameters space that generalize poorly.”

What it says is that the standard practice of initializing the learnable parameters with a uniform random distributions may not lead to a model that generalizes well. **We can only expect this problem to become worse when there's a dearth of labeled training data.**

## The Results Look Great, But ... (contd.)

- About the potential of unsupervised pre-training to remediate this problem, the authors Erhan et al. go on to say:

“... unsupervised pre-training as an unusual form of regularization: minimizing variance and introducing bias towards configurations of the parameter space that are useful for unsupervised learning. ”

That is, we can think of pre-training from unlabeled data as a form of “initialization with regularization” for the learnable parameters.

- A rather simple way to carry out such pre-training would be to change the output of your network by possibly extending it with a fully-connected layer so that the entire network acts like an autoencoder. Now you can sensitize the learning weights in the transformer model by requiring that the inputs match the outputs while you feed unlabeled data into the input.
- In the next section, I'll briefly talk about a particular class of data pre-conditioning strategies mentioned in the above paper that are known as **Generative Pre-Training (GPT)** strategies.



# Outline

1	The Basic Idea of Dot-Product Attention	9
2	Multi-Headed Attention	19
3	Implementation of Attention in DLStudio's Transformers	23
4	The Encoder-Decoder Architecture of a Transformer	29
5	The Master Encoder Class	35
6	The Basic Encoder Class	37
7	Cross Attention	40
8	The Basic Decoder Class	46
9	The Master Decoder Class	49
10	Positional Encoding for the Words	53
11	TransformerFG and TransformerPreLN Classes in DLStudio	58
12	Regarding the Difficulty of Training a Transformer Network	63
13	Results on the English-Spanish Dataset	70
14	<b>Transformers with Generative Pre-Training (GPT)</b>	<b>81</b>

# Generative Pre-Training (GPT) for Transformers

- The last couple of slides talked about the general case of unsupervised pre-training of the model using unlabeled datasets for performance boost especially when the labeled datasets are small. Generative pretraining (GPT) is a special case of that.
- As to why “generative”, as was observed by Erhan et al., suppose  $X$  represents the input to a network and  $Y$  its output. A purely discriminative network is only concerned about the conditional  $P(Y|X)$ . On the other hand, a generative network is concerned about the joint  $P(X, Y)$ .

[That is, while a discriminative network focuses on just getting  $Y$  right for whatever  $X$  it is presented with. On the other hand, a generative network places both the input  $X$  and the output  $Y$  on an equal footing. For these reasons, generative approaches are less prone to overfitting than purely discriminative approaches. Becoming aware of  $P(X)$  would be akin to applying PCA to the unlabeled data in traditional machine learning. The same things happens in the deep-learning context when the input data is first mapped to embeddings with the expectation that similar elements at the input would result in embedding vectors that are closer together in value.]

- I'll now present some insights gleaned from the paper “*Improving Language Understanding by Generative Pre-Training*” by Radford et al.:

<https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf>

## GPT for Transformers (contd.)

- In the context of creating language models with transformers, the focus of the paper by Radford et al. is exclusively on generative approaches to make such a model aware of  $P(X)$  with unsupervised training using unlabeled datasets.
- The generative pretraining as presented as proposed by Radford et al. consists of maximizing the likelihood  $L$  given by

$$L(\mathcal{U}) = \sum_i P(u_i | u_{i-k}, \dots, u_{i-1}; \Theta)$$

where  $\mathcal{U}$  represents the “tokens” in the corpus and  $k$  the size of the context window. [Using the words directly in creating a language model can result in too large a vocabulary — you’ll need a separate representation for every possible inflection of each noun and every possible conjugation of each verb. Besides, you will also run into problems with “synthesized” words like “overparameterized”. Language modeling becomes more efficient if the words are first decomposed into tokens through a step called tokenization. As you would expect, tokenization is highly language specific.]

- For the purpose of pretraining, the idea would be to possibly extend transformer model you want to train so that you can measure the conditional probability shown above and use its maximization as the learning objective during pretraining.

## GPT for Transformers (contd.)

- As you would expect, the maximization of the pretraining objective shown on the previous slide will make the network smarter about the context for the tokens, for the words, for the sentences, etc.
- The weights that are learned during pretraining would give the network a good sense of what tokens, what words, what sentences, and, perhaps, even what paragraphs constitute good sequences with regard to how one word follows another, how one sentence follows another, or, even, how one para follows another.