

Transformers: Learning with Purely Attention Based Networks

Avinash Kak
Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Sunday 15th February, 2026 17:56

©2026 A. C. Kak, Purdue University

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 of the fundamental ideas involved will be in the context of **sequence-to-sequence learning** as required for automatic translation. In particular, I will focus on English-to-Spanish translation as a case study. **Later I'll talk about how to apply these ideas for solving problems in computer vision.**

Preamble (contd.)

For seq2seq learning in general, attention takes two forms: **self-attention** and **cross-attention**. However, for solving recognition problems in vision (or in languages), you may need only **self-attention**.

Self-attention means for a neural network to figure out on its own what parts of a sequence, such as a sentence of words or a sequence of patches in images, together contribute to solving the problem at hand. For example, for language translation, the goal of self-attention would be to figure out which words together in the source language contribute to the production of any single word in the target language. In image recognition, on the other hand, self-attention would help a network figure out which patches together contribute the most for correctly predicting the class label.

To elaborate, in seq2seq learning, 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.

Preamble (contd.)

A neural network with self-attention would be able to accomplish what is mentioned at the bottom of the previous slide. Such a network 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.

For another example, again in seq2seq learning, 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.

Preamble (contd.)

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.

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 with regard to 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 that would not be appropriate here.

Preamble (contd.)

To continue with the example at the bottom of the previous slide, 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”.

In this lecture, I’ll be teaching purely-attention based learning with the following three inner classes in the [Transformers](#) module of DLStudio:

- `TransformerFG`
- `TransformerPreLN`
- `visTransformer`

The first two, meant for seq2seq learning, are only slightly different variants of the same implementation. I have kept them separate for educational reasons. The last one shows how to use the self-attention in Transformers for solving image recognition problems in computer vision.

Preamble (contd.)

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 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>

The class, `visTransformer`, meant for solving image recognition problems, is based on the paper “*An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale*” by Dosovitskiy et al.:

<https://arxiv.org/pdf/2010.11929.pdf>

All three Transformer classes mentioned above are defined in the module file

`Transformers.py` in DLStudio.

Preamble (contd.)

About the dataset I'll be using to demonstrate Transformers for seq2seq learning, 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 seq2seq Transformer code in DLStudio:

```
seq2seq_with_transformerFG.py  
seq2seq_with_transformerPreLN.py
```

For the image recognition class [visTransformer](#), I'll use the CIFAR-10 dataset that you are already very familiar with. The following two scripts in the same [ExamplesTransformers](#) directory as mentioned above are your main entry points for playing with the vision related Transformer code in DLStudio:

```
image_recog_with_visTransformer.py  
test_checkpoint_for_visTransformer.py
```

Preamble – How to Learn from These Slides

At your first reading of these slides, just focus on thoroughly understanding the following three topics:

- What do we mean by attention and the theory and implementation of the QKV attention that the transformers are based on? This is explained on Slides 12 through 20, **for a total of 8 slides**.
- Your next topic should be coming to grips with the notion of multi-headed attention and its DLStudio implementation as explained on Slides 27 through 35, and with the design of the overall architecture for attention based predictions on Slides 37 through 41, **for a total of 12 slides**.
- Now jump to the end and spend some time on the vision transformer on Slides 91 through 101, **for a total of 10 slides**.

That makes for a total of 30 slides for your first reading. Note, however, in this lecture in particular, the rest of the material not included above is just as important. However, after you have understood the core concept of what exactly is meant by Attention, you should be able to breeze through the rest with relative ease.

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

Word Embeddings and the Associated q, k, v Vectors

- Modern attention networks based on transformers were developed originally for solving seq2seq learning problems as required for automatic translation. Although more recently similar networks have also been used for solving problems in other domains — for example, computer vision — seq2seq learning still feels like the most natural “domain” for a first introduction to the concept of transformers.
- For language translation, at this point at least, we will start with *word* as a basic element of a language. [Later you will see that, from the standpoint of deep learning, the most basic element of a language is a “token”]
- We will represent each word by a learnable numeric vector, denoted w , of real numbers that we will refer to as the *embedding* for the word. We will use the symbol M to denote the dimensionality of the embedding vectors. The value to use for M would be a hyperparameter of the system.

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

- Computing the attention means trying to figure out the importance of each word to every other word in the input sequence. To that end, we associate a triple of vectors denoted q , k , and v with each word w . Regarding the notation, the symbol q stands for the Query Vector, the symbol k for the Key Vector; and the symbol v for the Value Vector.

[At this point you are likely to be thinking why not use the dot product of the two embedding vectors w_1 and w_2 directly as a measure of their importance to each other. Since both w_1 and w_2 are going to be learned anyway, we could set up a loss function that would maximize such dot products. The problem with such thinking is that it over-constrains the embedding vectors. The same two words are not likely to be important to each other in all sentences. What you need is a learning mechanism that maps each word embedding vector into one or more other spaces in a context aware fashion. One can then try to maximize the dot products in those spaces. That is exactly what is achieved by the (q, k, v) vectors.]

- It is through these three vectors — the q , k , and v vectors — that each word w expresses itself vis-a-vis all the other words in a given input sequence.
- As you will see, when all of the words in an input sequence are considered together in the form of a tensor, the three vectors defined above become tensors. I'll denote these tensors as Q for Query, K for Key, and V for Value.

Calculating (q, k, v) Vectors with Matrix Multiplications

- Continuing with the thought expressed in the previous slide, 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:

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

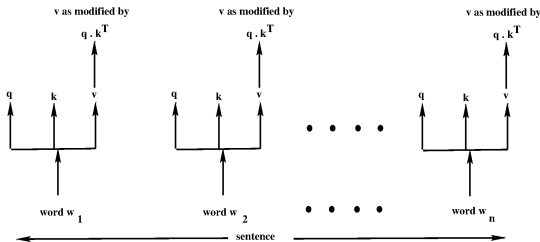
where w is the M -dimensional embedding vector for the word in question. All three matrices W_q , W_k and W_v are of size $M \times M$.

- You can think of the q , k , and v vectors as a word w 's three representatives for assessing and recording the importance of the word in question to every other word in a sentence. You could say that 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.

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

- It might help if you think of the $M \times M$ matrix W_q as a *mapping device* that can map each word's M -dimensional embedding to its M -dimensional query vector. The same would apply to W_k and W_v .
- 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.]



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.)
- We can similarly pack together all the query vectors q for the individual words in the input sentence into a tensor Q , all the key vectors k into a tensor K , and, finally, all the value vectors v into a tensor V . Since the learnable mapping matrices W_q , W_k , and W_v remain the same for all the words, we will simply represent them by W_Q , W_K , and W_V merely to indicate their use with the input sentence tensor.

Calculating the (Q, K, V) Tensors

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

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

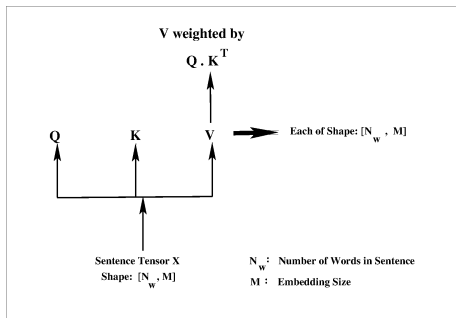
- Using N_w to denote the number of words in a sentence, we have $[N_w, M]$ for the shape of the input tensor X . In line with the earlier discussion, we know that the three matrices W_Q , W_K , and W_V are each of size $M \times M$. As a result, we'll have $[N_w, M]$ for the shapes of the output Q , K , and V tensors. [To be precise, the three tensors are all of shape (B, N_w, M) when you include the batch axis. But I am ignoring the batch axis for now.]

[To reiterate for emphasis what was mentioned on the previous slide, the point here is that the same $M \times M$ matrix W_Q can “inform” each word w what its query vector q should be. That is, the W_Q shown in Eq. (1) on Slide 14 are the same for all the different words in the input sentence. Remember, each word w is represented by its own embedding vector. The product of that embedding with W_Q is what makes the q vector different for each word w . This assumption applies to W_K and W_V also.]

- 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 15 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 M is also 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^T 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, M) 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:

$$Z = \frac{\text{nn.Softmax}(\text{dim} = -1)(Q \cdot K^T)}{\sqrt{M}} \cdot V \quad (3)$$

The `nn.Softmax` is applied along the word axis for the K tensor in the dot-product “matrix” $Q \cdot K^T$ (see the small-font red note below). The shape of Z is also $N_w \times M$, which is the same as the shape of the input sentence. You can think of Z as the attention-enriched form of the input.

[Note that $Q \cdot K^T$ also has 3 axes, like the tensors Q and K , but with different ‘meanings’. While Axis 0 of $Q \cdot K^T$ is the batch axis, as you’d expect, Axis 1 is for indexing along the words in Q tensor, and Axis 2 for indexing along the words in K tensor. What would normally be the embedding axis gets decimated in the dot-product matrix $Q \cdot K^T$]

- The additional normalization by \sqrt{M} in Eq. (3) 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 .]

What's Achieved by `nn.Softmax` Normalization of $Q.K^T$

- On this and the next few slides, my goal is to explain what it is we achieve by `nn.Softmax` normalization in Eq. (3) on the previous slide.
- The main purpose of `nn.Softmax` is to turn an array of numbers into a probability distribution. This is illustrated by the following interactive 1-D example:

```

>>> x = torch.randn(4)
>>> x
      tensor([ 1.2863,  0.8004, -0.4011, -0.4316])          ## (1)
>>>
>>> m = torch.nn.Softmax(dim=0)                            ## (2)
>>>
>>> y = m(x)                                               ## (3)
>>> y
      tensor([0.5052, 0.3108, 0.0934, 0.0906])           ## (4)
>>>
>>> torch.sum(y)
>>>
      tensor(1.)                                          ## (5)

```

- The `nn.Softmax` callable instance constructed in Line 2 above normalizes the 1-D data in Line 1 by applying to it the normalization

$$\text{Softmax}(x_j) = \frac{\exp(x_j)}{\sum_j \exp(x_j)}.$$

nn.Softmax Normalization of $Q \cdot K^T$ (contd.)

- The numbers returned by the `nn.Softmax` operator are shown in Line 4 on the previous slide. These numbers represent a probability distribution because they are all non-negative and they add up to 1, as shown in Line 5.
- Let's now consider a case that looks more like the normalization in Eq. (3) on Slide 20. As you know, if the input sequence consists of N_w words, the $Q \cdot K^T$ tensor will look like a $N_w \times N_w$ square matrix.
- Let's assume that $N_w = 4$ and pretend that the tensor `x` created below represents an example of $Q \cdot K^T$. In what follows, I'll also assume a batch size of 1.

```
>>> x = torch.randn(1,4,4) ## (6)
```

```
>>> x ## (7)
      tensor([[[[-0.1139,  0.2006,  0.3630,  0.3736],
                [ 1.3405,  1.2014, -0.5397,  1.0641],
                [-0.2859,  0.9316, -0.2158,  1.4118],
                [-0.7513, -0.1098, -1.5254,  0.2604]]]])
```

nn.Softmax Normalization of $Q.K^T$ (contd.)

- As for 1D case on Slide 21, I'll now construct an instance of the `nn.Softmax` operator, but this time we set the constructor parameter `dim` to `-1`, which would correspond to Axis 2 of a 3-axis tensor.

```
>>> c = torch.nn.Softmax(dim=-1)                                ## (8)
```

- Applying this operator to the tensor `x`, we get

```
>>> out = c(x)                                                ## (10)
```

```
>>> out
      tensor([[[[0.1783, 0.2442, 0.2872, 0.2903],
                [0.3596, 0.3129, 0.0549, 0.2727],
                [0.0916, 0.3096, 0.0983, 0.5005],
                [0.1636, 0.3108, 0.0755, 0.4501]]]])          ## (11)
```

So applying `nn.Softmax(dim=-1)` to the array in Line 7 of the previous slide **converts each of the rows of that array into a probability distribution.**

nn.Softmax Normalization of $Q \cdot K^T$ (contd.)

- You can verify that each row of the array returned by `nn.Softmax(dim=-1)` as shown in Line 11 is a probability distribution by

```
>>> torch.sum(out, 2)
      tensor([[1.0000, 1.0000, 1.0000, 1.0000]])
```

- On the other hand, if you were to sum the tensor along the columns, you'd get the following, which says that the columns in the array in Line 11 do NOT constitute probabilities.

```
>>> torch.sum(out,1)
      tensor([[0.7931, 1.1775, 0.5158, 1.5136]])
```

- What the above implies is that the `nn.Softmax(dim=-1)` normalized dot-product $Q \cdot K^T$ is still a $N_w \times N_w$ matrix with the only difference that the normalization causes each row of the dot product to become a probability distribution over the words supplied by the K tensor. **That is, for each word in the Query tensor, there corresponds a row in the normalized dot-product tensor and the numbers in that row are a probability distribution over all the words in the Key tensor with regard to their importance to the Query word.**

nn.Softmax Normalization of $Q \cdot K^T$ (contd.)

- To complete our analysis of Eq. (3) on Slide 20, let's now consider the matrix product formed by multiplying the normalized $Q \cdot K^T$ tensor with the Value Tensor V . Again the shape of $Q \cdot K^T$ is (N_w, N_w) and the shape of V is (N_w, M) . To that end, let's assume that $M = 7$ and $N_w = 4$ in the following example. The value of `out` is as in Line 11 on Slide 23.

```
>>> V = torch.randn(1,4,7)
>>>
>>> V
      tensor([[[[-1.7409,  1.4361, -1.6446, -0.4108, -1.1131,  0.6096,  0.5573],
                [-1.5100, -0.9030, -0.0375, -0.8503, -0.3711, -0.7047, -0.4001],
                [-2.2639, -0.7271, -1.4457, -1.2576, -1.2605, -0.2686,  1.4099],
                [ 0.2188, -0.1943, -0.8498, -1.3114, -0.2914,  1.1875,  0.4002]]]])

>>> out @ V
      tensor([[[[-1.2659, -0.2297, -0.9643, -1.0228, -0.7357,  0.2042,  0.5228],
                [-1.1629,  0.1410, -0.9141, -0.8404, -0.6649,  0.3078,  0.2617],
                [-0.7401, -0.3167, -0.7297, -1.0808, -0.4866,  0.4056,  0.2661],
                [-0.8266, -0.1880, -0.7724, -1.0166, -0.5238,  0.3949,  0.2534]]]])
```

- This would be the attention suffused version of the Value Tensor V , obtaining which is the entire purpose of Eq. (3) on Slide 20.** Note that the shape of the final result shown above is still (N_w, M) .

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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 S_{qkv} .

Multi-Headed Attention (contd.)

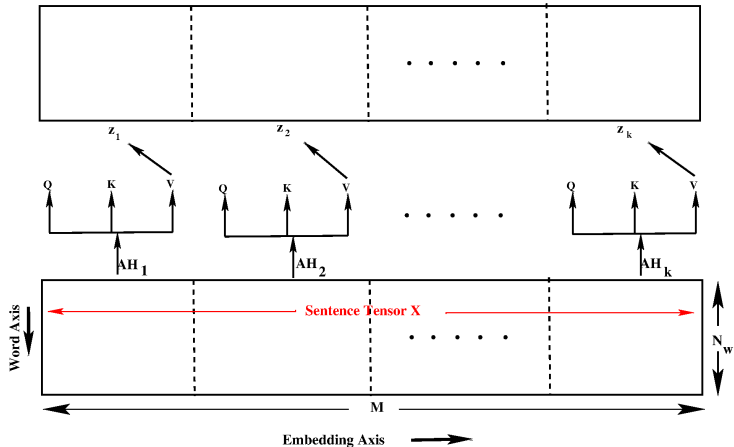


Figure: **Correction:** In the upper part of the figure, read Z_K as Z_{N_H} . And, in the middle of the figure, read AH_k as AH_{N_H} . The symbol N_H stands for the number of Attention Heads used.

Multi-Headed Attention (contd.)

- Continuing with the last bullet on Slide 27, we can write the following for the size of the word embedding slice fed into any single attention head:

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

- Each Attention Head learns its own values for the Q , K , and V tensors with its own mapping matrices for W_Q , W_K , and W_V .
- Since each Attention Head receives only a s_{qkv} -sized slice from the embedding axis of the input sentence, its output tensors Q , K , V will be of shape (N_w, s_{qkv}) for the same reason as described in the previous section.
- What's interesting is that while Q and K are of shape (N_w, s_{qkv}) for an N_w -word sentence for each Attention Head, **the inner-product $Q \cdot K^T$ continues to be of the same shape as in the previous section, that is $N_w \times N_w$.** What that implies is that you'll end up with N_H different assessments of the word interdependencies with N_H attention heads.

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

Attention Head Implementation

- The next slide shows the implementation of the `AttentionHead` in the three transformer classes in DLStudio.
- In the code shown on the next slide, the dot-product mentioned previously is calculated in line (K). Next, as shown in line (L) 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:

$$Z = \frac{\text{nn.Softmax(dim = -1)}(Q \cdot K^T)}{\sqrt{M}} \cdot V$$

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 `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_attn_heads):
        super(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( self.qkv_size, self.qkv_size )      ## (B)
        self.WK = nn.Linear( self.qkv_size, self.qkv_size )      ## (C)
        self.WV = nn.Linear( self.qkv_size, 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 )               ## (G)
        K = self.WK( sent_embed_slice )               ## (H)
        V = self.WV( sent_embed_slice )               ## (I)
        A = K.transpose(2,1)                          ## (J)
        QK_dot_prod = Q @ A                           ## (K)
        rowwise_softmax_normalizations = self.softmax( QK_dot_prod ) ## (L)
        Z = rowwise_softmax_normalizations @ V          ## (M)
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(self.dl_studio.device) ## (N)
        Z = coeff * Z                                  ## (O)
        return Z

```

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 35, for an input sentence consisting of N_w words and the embedding size denoted by M , the sentence tensor at the input to `forward()` of `SelfAttention` in Line (B) on Slide 35 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` in its `forward()` in Line (F) 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) on Slide 35.

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) on the next slide is $[B, N_w, M]$, that will also be the shape of the output produced by layer.
- Since the shape of the tensor at the output of a `SelfAttention` layer is exactly the same as the shape of the tensor at its input, you could say that the main purpose of self-attention is to generate attention-enriched versions of the tensor at its input. Obviously, for the first such layer that is receiving a tensor composed of the word embedding vectors, its output will be the attention-enriched versions of those vectors.
- 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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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.
- For seq2seq learning, 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 inter-word dependencies in the source-language sentence that take into account what's needed for achieving the ground-truth translations 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 41.
- 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`.
- 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`.

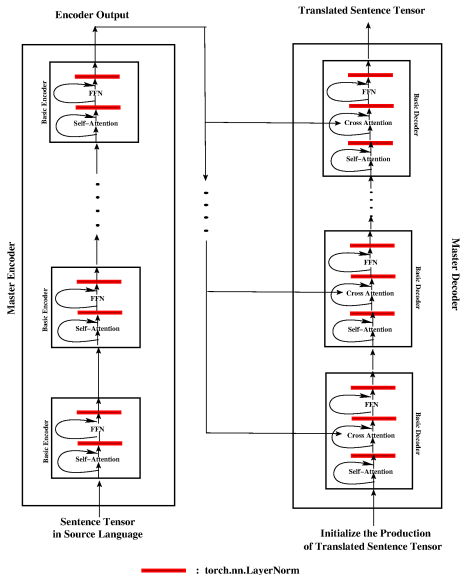
The Transformer Architecture (contd.)

- The implementation classes mentioned on the previous slide are explained in greater detail in the several sections that follow.
- Earlier I said 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 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.

The Transformer Architecture (contd.)

- 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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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_attn_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_attn_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)):
            out_tensor = self.basic_encoder_arr[i](out_tensor)    ## (B)
        return out_tensor
```

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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 41.
- 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 35, 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 words in the 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)
        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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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.
- As you know, self-attention consists of taking dot products of the Query vector for each individual word in a sentence with the Key vectors for all the other words in order to discover the inter-word dependencies in a sentence. **On the other hand, in cross-attention we take the dot products of the Query vector for each individual word in the *target-language* sentence with all 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_atten_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_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:

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

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:

$$Z_{\text{cross}} = \frac{nn.\text{Softmax}(\text{dim} = -1)(Q_{\text{source}} \cdot K_{\text{target}}^T)}{\sqrt{M}} \cdot V_{\text{source}} \quad (6)$$

```
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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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 41.
- 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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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 Slide 59, 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) on Slide 59. The mask is 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)])
        ## (A)
        ## 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)):
                out_tensor = self.basic_decoder_arr[i](out_tensor, final_encoder_out, mask)
                ## (K)
            last_word_tensor = out_tensor[:, mask_index]
            ## (L)
            last_word_onehot = self.out(last_word_tensor.view(sentence_tensor.shape[0], -1))
            ## (M)
            output_word_logprobs = nn.LogSoftmax(dim=-1)(last_word_onehot)
            ## (N)
            _, idx_max = torch.max(output_word_logprobs, 1)
            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

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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 every element of the embedding vectors used for all the words.
- 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 numerators of the args to 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.)

- 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

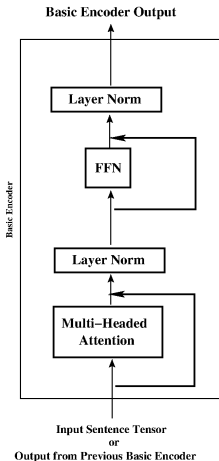
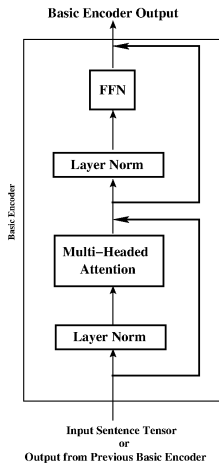
1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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 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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

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. By n -gram here, I mean 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”. 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 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 you 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

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	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

Results on the English-Spanish Dataset

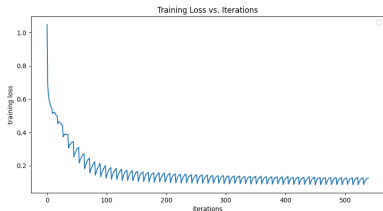


Figure: Training loss vs. iterations for 20 epochs with the TransformerFG class in DLStudio.

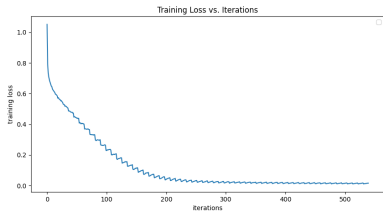


Figure: Training loss vs. iterations for 60 epochs with the TransformerPreLN class in DLStudio.

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 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 [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 [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 [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 [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 [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 [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 [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 79). 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 79, the English vocab used in the dataset consists of 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 preconditioning 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 preconditioning strategies mentioned in the above paper that are known as **Generative Pre-Trained Transformer (GPT)** strategies.

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

Generative Pre-Trained Transformer (GPT)

- 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. [More accurately speaking, the acronym GPT stands for Generative Pre-trained Transformer.]
- 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.:

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. More on tokenization in my Week 15 lecture.]

- 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.
- I'll be discussing these ideas in much greater detail in my next week's lecture on LLM (Large Language Models).

Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

Image Recognition with a Transformer

- As mentioned in the Preamble, it was shown in the paper “*An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale*” by Dosovitskiy et al. that transformers could also be used for solving image recognition problems.
- The authors referred to their contribution as ViT for “Vision Transformer”.
- The main contribution of ViT was to demonstrate that if you chopped up an image into an array of patches, with each patch of size 16×16 , and if you then represented each patch with a learnable embedding, you could literally use the same transformer architecture as in language modeling for solving image recognition problems.
- You could think of the non-overlapping patches extracted from an image — left to right and top to bottom — as constituting a patch sequence, in very much the same way you think of a sentence as consisting of a sequence of words.

Image Recognition with a Transformer (contd.)

- Through embedding vectors, you would then be able to represent an image by exactly the same sort of a tensor as you have seen earlier in this lecture.
- I have illustrated this idea in the figure on the next slide that shows us representing an image with an 5×5 array of nonoverlapping patches. If a patch consists of $p \times p$ pixels, and assuming that we are talking about color images, we will have a total of $p^2 \times 3$ numeric values in a patch.
- For transformer based processing, our goal is to learn to map the $3 \times p^2$ numeric values in a patch to an embedding vector of size M . If P is the total number of patches in an image, this mapping will convert an image into a tensor of shape $[P, M]$. This is exactly the sort of a tensor as for a sentence of words as shown in the lower half of Slide 22.

Image Recognition with a Transformer (contd.)

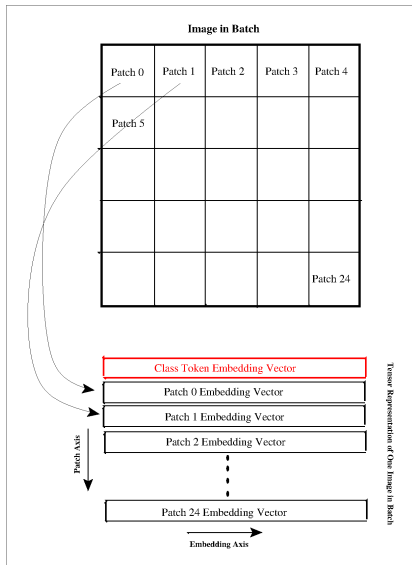


Image Recognition with a Transformer (contd.)

So far the idea of applying transformers to images seems straightforward. But here are two key highly novel ideas in the ViT architecture that would not be so easy to think of even in hindsight:

- For image recognition, you have a class label for every training image and the question is how to incorporate that in a transformer based neural network.

[In a regular convolutional neural network, you push the image through the neural network that makes a prediction for the image class label at the output. You compare the predicted label with the true label and thus estimate the loss that is backproped through the network. Unfortunately, that does not work for transformer based networks simply because — if I could put it that way — such networks are more intensive in the extent of learning they need to carry out.]

The authors of the ViT paper discovered that if they gave the transformer a “cell” in which it could store its understanding of what was unique to all the images for the same class, that helped the neural network make a correct guess for the class label. This “cell” is referred to as the **class token** in the ViT architecture.

- As you know already, transformer based learning for languages required positional encoding for the words that gave the network a sense of the order in which the words existed in a sentence. As you’ll recall, I presented sinusoidal positional encoding for the case of language modeling on Slides 60-64. The question now is: **How does one do that for patch sequences?** Here again, the solution consisted of providing another “cell”, **but this time on a per-patch basis**, where the network can put away its understanding of the order in which the patches relate to one another spatially. **These per-patch cells are referred to as positional-encodings in ViT.**

Coding Issues for Vision Transformers

- I'll now review some of the coding issues you are going to run into if writing your implementation for a vision transformer, or if you are trying to understand the `visTransformer` class in the `Transformers` module of DLStudio.
- But first you have to realize that the overall neural architecture for a vision transformer is much simpler than what it is for language modeling. That is because you do not need the Decoder you saw earlier for the case of languages.
- For example, for a vision transformer meant for image recognition, you feed the output of the Encoder into a couple of Fully Connected (FC) layers. The number of nodes in the final output layer of the FC section equals the number of classes in your dataset.
- Actually, a vision transformer is even simpler than what would be implied by the above claim, as explained on the next slide.

Coding Issues for Vision Transformers (contd.)

- Since the sole job of a vision transformer (meant for image recognition) is to predict the class label of the input image and since the purpose of the **class token** mentioned on the previous slide is to learn what is unique about all the images that belong to the same class, you only need to retain the class token from the output of the transformer. **That is, you would feed the embedding vector for just the class token into the FC section for the prediction of the class label.**
- With that general introduction to the overall architecture of a Vision Transformer, I'll now go into the specifics of what you're going to need in your code.
- Obviously, the very first thing you would need to do in your code would be to extract the patches from the images and, for each image, construct a tensor of shape $P \times M$ for its representation, where P is the number of patches in an image and M the dimensionality of the embedding representation of a patch.

Coding Issues for Vision Transformers (contd.)

- About extracting the patches and mapping them to their embedding vectors, you can use one of the following two ways for that. **Although they look very different, under the hood they are the same.**
 - You invoke a convolutional layer in kernel-size and the stride equal the patch size. Let's say you patch size 16×16 . You will construct an instance of the 2D convo operator as follows:

```
conop = nn.Conv2d( 3, M, P, stride=P )
```

where 3 is for the three color channels of a training image, M the embedding size and P the kernel size. By setting both the kernel and the stride to the same value, you will directly output the embedding vector of size M for each non-overlapping $P \times P$ patch in the image. If your training dataset is CIFAR-10, your input images are of size 32×32 . If you want your patches to be of size 16×16 , you would set $P = M = 16$.

- The second approach is based on separately extracting the patches by calling `torch.tensor.unfold()` and then mapping them with an `nn.Linear` layer to the embedding vectors, as shown below:

```
for i, data in enumerate(self.train_data_loader):
    input_images, labels = data
    ...
    patch_sequences = input_images.unfold(2, self.patch_size[0], self.patch_size[1]).unfold(3, \
                                                self.patch_size[0], self.patch_size[1])
    patch_sequence_embeddings = patch_embedding_generator( patch_sequences )
```

visTransformer in DLStudio

- The `visTransformer` class in the `Transformers` module in DLStudio consists of the following inner classes and methods:

```
class visTransformer(nn.Module)

    class PatchEmbeddingGenerator(nn.Module)
    class MasterEncoder(nn.Module)
    class BasicEncoder(nn.Module)
    class SelfAttention(nn.Module)
    class AttentionHead(nn.Module)

    def run_code_for_training_visTransformer(self, dls, vis_transformer, display_train_loss=False,
                                           checkpoint_dir='checkpoints')
    def run_code_for_evaluating_visTransformer(self, encoder_network, patch_embedding_generator)
    def run_code_for_evaluating_visTransformer(self, encoder_network, patch_embedding_generator)
    def run_code_for_evaluating_checkpoint(self, encoder_network, patch_embedding_generator,
                                          checkpoints_dir)
```

- The names I have used the main vision transformer class `visTransformer` and its five inner classes should make them self-explanatory.
- Of the five inner classes, you have already seen the last four. In what follows, I'll present the definitions for the main class `visTransformer` and its inner class `PatchEmbeddingGenerator`

Definition of the `visTransformer` Class

- Shown on the next slide is the the top-level class for the vision transformer in the `Transformers` module of DLStudio. We instantiate the transformer network in Lines (1), (2) and (3).
- Of the data flow presented in Lines (4) through (9), the most notable fact is the “expansion” of the class token to cover all the patches in a single image and its concatenation with the Axis 1 of the batch. The batch is of shape $[B, P, M]$ where B is the batch size, $P + 1$ the number of patches along with the class token, and M the embedding size. So the concatenation you see in Line (5) is along the patch Axis — that is it is in accordance with the image representation shown in Slide 93. Remember, whatever logic you place in the `forward()` of a class derived from `nn.Module` is automatically applied to every instance in a batch.
- As you see in Line (7), we only retain the first embedding vector, the one that corresponds to the class token, for feeding into the fully-connected section.

Definition of the visTransformer Class

```

class visTransformer(nn.Module):
    def __init__(self, dl_studio, patch_size, embedding_size, num_basic_encoders, num_attn_heads,
                 save_checkpoints=True, checkpoint_freq=10):
        super(visTransformer, self).__init__()
        ...
        self.checkpoint_freq = checkpoint_freq
        self.learning_rate = dl_studio.learning_rate
        self.num_patches_in_image = (dl_studio.image_size[0] // patch_size[0] ) *
                                    (dl_studio.image_size[1] // patch_size[1] )
        self.max_seq_length = self.num_patches_in_image + 1
        self.patch_size = patch_size
        self.patch_dimen = (patch_size[0] * patch_size[1]) * 3
        self.embedding_size = embedding_size
        self.num_basic_encoders = num_basic_encoders
        self.num_attn_heads = num_attn_heads
        self.master_encoder = visTransformer.MasterEncoder(dl_studio, self, num_basic_encoders,
                                                         num_attn_heads) ## (1)

        self.fc = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(embedding_size, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 10),
        ) ## (2)

        self.class_token = nn.Parameter(torch.randn((1, 1, embedding_size))).cuda() ## (3)
    def forward(self, x):
        class_token = self.class_token.expand(x.shape[0], -1, -1) ## (4)
        x = torch.cat((class_token, x), dim=1) ## (5)
        x = self.master_encoder(x) ## (6)
        predicted_class_tokens = x[:,0] ## (7)
        output = self.fc(predicted_class_tokens) ## (8)
        return output ## (9)

```

Definition of the PatchEmbeddingGenerator Class

- Shown below is the class `PatchEmbeddingGenerator`. It is an inner class of the `visTransformer` class shown on the previous slide.
- The most notable part of the code is how we add Positional Encodings to the patches in Line (5). As mentioned earlier, Positional Encoding consists of learning a parameter on a per-patch basis that is unique to that patch in the image. Taking all of the training images into account, what is unique to each patch is its position in the image. The parameter that learns this fact is defined in Line (2).

```

class PatchEmbeddingGenerator(nn.Module):
    def __init__(self, vis_xformer, embedding_size):
        super(visTransformer.PatchEmbeddingGenerator, self).__init__()
        self.num_patches_in_image = vis_xformer.num_patches_in_image
        self.patch_dimen = vis_xformer.patch_dimen          ## (num of pixels in patch) * 3 for color
        self.embedding_size = embedding_size
        self.embed = nn.Linear(self.patch_dimen, embedding_size)          ## (1)
        self.positional_encodings = nn.Parameter(torch.randn((1,
                                                                self.num_patches_in_image, self.embedding_size)))          ## (2)

    def forward(self, x):
        x = x.reshape(x.shape[0], -1, self.patch_dimen).cuda()          ## (3)
        patch_embeddings = self.embed(x)          ## (4)
        position_coded_embeddings = patch_embeddings + self.positional_encodings          ## (5)
        return position_coded_embeddings

```

The Vision Transformer Examples in the ExamplesTransformer Directory

- You will find the following scripts in the ExamplesTransformers of DLStudio:

```
image_recog_with_visTransformer.py
```

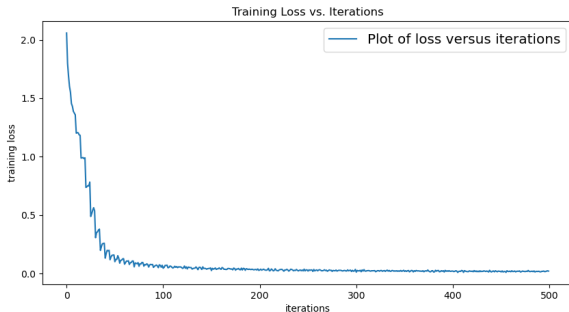
```
test_checkpoint_for_visTransformer.py
```

- When you run the first script, it outputs checkpoints every 10 epochs (by default). As the first is continuing to train further, you can test the quality of the model learned in a checkpoint by executing the second script. [See the doc section of the second script for to specify a particular checkpoint.](#)
- Without any hyperparameter tuning**, shown below are some results on the testing-portion of the CIFAR-10 dataset:

Displaying the confusion matrix:

	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane:	65.63	1.60	3.41	3.71	2.91	1.90	2.40	1.20	13.23	4.01
car:	4.80	60.46	0.30	4.70	1.30	2.20	1.40	0.80	10.31	13.71
bird:	8.81	0.50	38.34	14.21	15.12	7.51	7.91	3.90	2.60	1.10
cat:	2.51	0.90	5.92	51.05	7.22	15.75	10.23	2.81	1.71	1.91
deer:	3.70	0.20	7.80	10.40	53.50	6.20	10.20	3.80	2.70	1.50
dog:	1.81	0.30	6.02	29.29	6.42	44.53	6.42	2.91	1.50	0.80
frog:	1.40	0.70	5.00	12.70	7.90	4.10	65.40	0.70	1.10	1.00
horse:	4.11	1.00	2.71	12.34	9.93	11.74	2.81	52.26	1.00	2.11
ship:	1.40	1.40	1.40	3.71	2.71	1.91	1.00	0.90	72.92	4.21
truck:	5.10	11.50	0.70	7.00	2.00	3.60	2.40	2.30	7.50	57.90

Training Loss on the CIFAR-10 Dataset Over 40 Epochs



Outline

1	The Basic Idea of Dot-Product Attention	11
2	Multi-Headed Attention	26
3	Implementation of Attention in DLStudio's Transformers	30
4	The Encoder-Decoder Architecture of a Transformer	36
5	The Master Encoder Class	42
6	The Basic Encoder Class	44
7	Cross Attention	47
8	The Basic Decoder Class	53
9	The Master Decoder Class	56
10	Positional Encoding for the Words	60
11	TransformerFG and TransformerPreLN Classes in DLStudio	65
12	Regarding the Difficulty of Training a Transformer Network	70
13	Results on the English-Spanish Dataset	76
14	Transformers with Generative Pre-Training (GPT)	87
15	The Vision Transformer Class <code>visTransformer</code>	91
16	Using QKV Modeling for Inter-Pixel Attention	105

QKV for Inter-Pixel Attention

- I am now going to revisit DLStudio's [GenerativeDiffusion](#) module — more specifically, the [AttentionBlock](#) inner class in that module. Although the Attention there is also based on the QKV concept as explained in this lecture, there are significant (and very interesting differences) between the implementation of the concept you have seen so far in this lecture and how the same concept is made to work for the case of diffusion.
- As you have seen in this lecture, the notion of the embedding vector representation of the basic units of the input data plays a fundamental role in the original formulation of Attention. [As you have seen already, Single-Headed Attention consists of learning from the embedding vector for each input unit (such as a word or a patch) a Query vector Q , a Key vector K , and a Value vector V . The QKV vectors for the different input units interact through dot-products for each input unit to figure out how it should attend to the other input units. And that's what's referred to as the Attention mechanism. Multi-headed attention does the same thing but by first segmenting the embedding vectors into P segments where P is the number of Attention Heads. Subsequently, the QKV attention is calculated for each segment in exactly the same manner as for Single-headed attention.]
- The same notion is used in UNetModel in GenerativeDiffusion for **inter-pixel attention** at a couple of different levels in the UNet.

QKV for Inter-Pixel Attention (contd.)

- As you would expect, the data that is input into the UNet is of shape (B, C, H, W) . For calculating the inter-pixel attention, for each **pixel** in the $H \times W$ array, we consider the C floating-point values along the channel axis as the embedding vector representation of that pixel.
- Subsequently, (1) We first flatten the $H \times W$ array of pixels into a 1-dimensional pixel array — just to make it easier to write the dot-product code later. (2) We use a 1-dimensional convolution on the 1-dimensional array of pixels to convert the C channels associated with each pixel into a $3 * C$ channels.
- Since the channel axis is used as the embedding vector at each pixel, increasing the number of channels gives us more latitude in dividing the channel axis into portions reserved for Q , K , and V .
- The next slide mentions a very interesting computationally efficient way of implementing Vaswani attention for the inter-pixel case.

QKV for Inter-Pixel Attention (contd.)

- An interesting difference between the formulation of Attention as in Vaswani et al. and the same mechanism for inter-pixel attention as implemented below is the absence of the matrices that multiply the embedding vectors for the calculation of Q , K , and V .
- In the implementation code you will see in the `GenerativeDiffusion` class, the Q , K , V matrices are incorporated implicitly in the matrix operator used for the 1-dimensional convolution carried out by the `self.qkv` operator that is declared in the constructor of the `AttentionBlock` class there.
- It is the `self.qkv` operator declared there that increases the the number of output channels from C to $3 * C$. Since, under the hood, a convolution in PyTorch is implemented with a matrix-vector product (as explained in my Week 8 slides), we can conceive of the matrix being segmented along its row-axis into three different parts, one that outputs the Q vector, the second that outputs the K vector, and third that output the V vector.