Visual Transformer

October 2, 2022

Useful papers - [VSP+17], [DBK+20], [CLJ19]

Contents

1 Introduction

Transformer is a powerful neural network architecture that covers all areas of the deep learning (audio, images, speech, video) and make SOTA in them. So, it is necessary to build a strong understanding of transformers. We will discuss transformer fundamentals based on sequences (e.g. text) and then generalize them for images. I think the most important part of the latter is the attention mechanism, thus lets start with it.

2 Sequences

2.1 Attention

I would say that attention is a mechanism that we use to focus on some part of the objects (image, text, audio and everything else) to make a decision. Definitely, we obtain this mechanism throughout our life. For example, if we need to choose dog or cat is in the image, we have not to explore the whole image, just enough to focus on animal's muzzle to make a correct decision.

So, our aim in the deep learning is to teach a neural network the attention mechanism because it is very important part of the human mind. Also, this mechanism can probably decrease computational resources needed to train neural network, so it makes us available to build more deep architectures and as a consequence obtain better results. Lets show it on the example of the convolutional layer. As you can remember in the convolutional layer we have a filter which runs through the entire image. But do we need to run the whole image? I think to make a correct decision we only need to choose specific region to convolve filter with this region. In other words, we can create attention mechanism for filter that gives us the specific region for that filter for further convolution. In such a way we decrease computational resources, specially when we have a large number of filters, because we use convolutional operation less times than in standard case.

Finally, lets consider how attention works for sequences. Also, we will process these sequences with recurrent nets. Let

$$\begin{gathered}\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_N\} - \text{some sequence} \\ \mathbf{x}_i \in \mathbb{R}^d, N - \text{sequence length}\end{gathered} \tag{1}$$

Our aim is to translate this sequence to another one $\mathbf{Y} = \{\mathbf{y}_1, ..., \mathbf{y}_N\}$, for example we are doing text translation between languages. We pass all elements of $\mathbf{X}$ through the RNN blocks, that is

$$\begin{gathered}\mathbf{H} = \{\mathbf{h}_1, ..., \mathbf{h}_N\}, \mathbf{h}_i \in \mathbb{R}^m \\ \mathbf{h}_i = \text{RNN}(\mathbf{x}_i, \mathbf{h}_{i-1}) - \text{hidden state}\end{gathered} \tag{2}$$

To predict $\mathbf{y}_i$ we will use attention. It is shown in Figure 1. Lets discuss how it works. As we said before attention gives us the opportunity to focus on some part of the object to make a decision. Lets keep it in mind. So, here as object we have hidden states, $\mathbf{H}$. Thus, to make a prediction, i.e. to predict $\mathbf{y}_i$, we need to calculate importance of each $\mathbf{h}_i$ for the $\mathbf{y}_i$. This importance tells us to which $\mathbf{h}_i$ (part of the object) we need to focus more. So, in such a way we realize the attention. Lets consider the formulas.

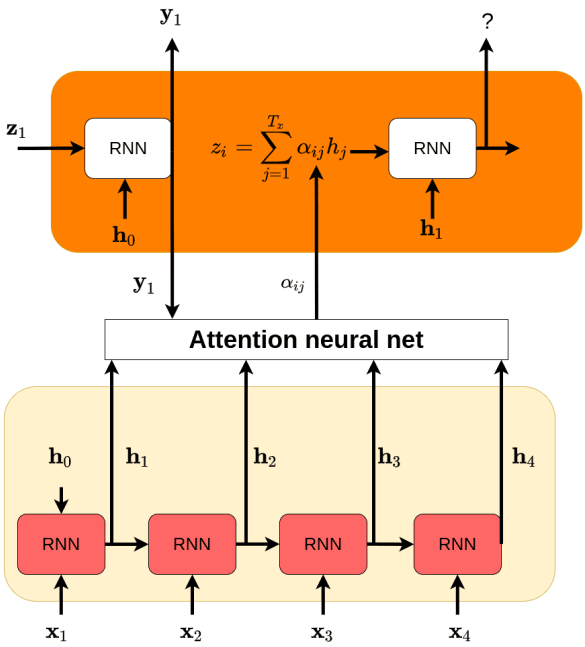

Figure 1: Schematic form of the attention for sequences

Attention layer gives the importance for some $\mathbf{y}_i$:

$$\begin{gathered}\mathbf{Attn}(\mathbf{y}_i, \mathbf{H}) = \sum_{j=1}^{N} \alpha_{ij} \cdot \mathbf{h}_j, \alpha_i \in \mathbb{R}^N \\ 0 \leq \alpha_{ij} \leq 1, \forall\ j = 1, ..., N \\ \sum_{j=1}^{N} \alpha_{ij} = 1\end{gathered} \tag{3}$$

α_i is a vector of weights for the y_i. The attention layer can be presented as the following

$$\begin{gathered}\mathbf{Attn}(\mathbf{y}_i, \mathbf{H}) = \text{softmax}\left(\text{score}(\mathbf{y}_i, \mathbf{H})\right)\mathbf{H} \\ \text{score}(\mathbf{y}_i, \mathbf{H}) \in \mathbb{R}^N, \mathbf{H} \in \mathbb{R}^{N \times m}, \mathbf{y}_i \in \mathbb{R}^m\end{gathered} \tag{4}$$

In theory, attention is defined as the weighted average of values. But this time, the weighting is a learned function. The score is unnormalized weights (the importance), and there are different choices for this function. The most famous one has the following form:

$$\text{score}(\mathbf{y}_i, \mathbf{H}) = \frac{\mathbf{y}_i \mathbf{H}^\top}{\sqrt{m}} \tag{5}$$

Here $\mathbf{y}_i \mathbf{H}^\top$ playing role of the similarity between y_i and $\mathbf{H}^\top$. I think this form of the score wins other ones because the dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

2.2 Transformer

Lets consider architecture of the transformer from the paper "Attention is all you need". We will try to understand self-attention, multi-head attention, positional encoding. The architecture is presented in Figure 2. It has encoder-decoder structure, encoder is shown in the left part, decoder - in the right. We will discuss the architecture step by step.

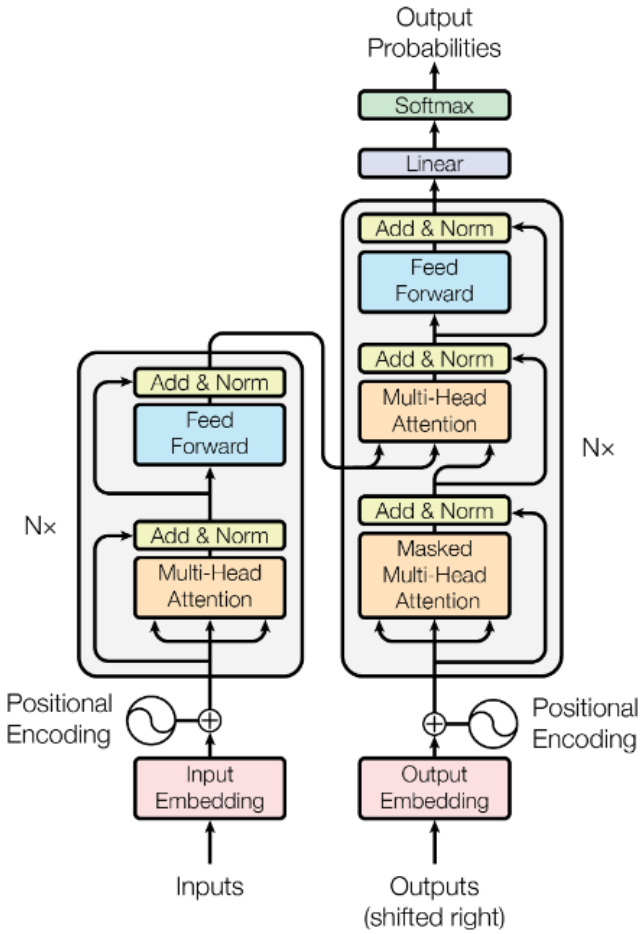

Figure 2: Architecture of the transformer

First of all we have a input sequence, we break this sequence into tokens and pass them through the embedding layer. After this we would have something like this:

$$\mathbf{H} = \{h_1, ..., h_N\}, \mathbf{H} \in \mathbb{R}^{b_{size} \times N \times d_{model}} \tag{6}$$

Here $\mathbf{H}$ - input embedded sequence, b_{size} - batch size, N - sequence length, d_{model} - embedding size. However embedding layer does not take into account positions of each word in sentence. That is, position of h_i is ignored. But this information is quite important, because meaning of some words can changes with respect to position in the sentence. In recurrent nets position of words was taken into account because it process sequence sequentially. But in Transformer we process the entire sequence at once. To fix this issue authors proposed positional encoding.

2.2.1 Positional encoding

From positional encoding we want the following

- It should output a unique encoding for each time-step (word's position in a sentence)
- Distance between any two time-steps should be consistent across sentences with different lengths.
- Our model should generalize to longer sentences without any efforts. Its values should be bounded.
- Positional encoding should provide small values to embedding vector because we do not want to destroy main signal.

Authors proposed to use sine/cosine positional encoding:

$$\begin{gathered}
\mathrm{PE}(\mathbf{H}) = \{h_1 + PE_1, ..., h_N + PE_N\}, PE_k \in \mathbb{R}^{d_{model}} \\
PE_k^i = \begin{cases} sin\left(\frac{k}{1000^{2i/d_{model}}}\right), \text{if } i \text{ is even} \\ cos\left(\frac{k}{1000^{2i/d_{model}}}\right), \text{if } i \text{ is odd} \end{cases} \\
PE_k = \left[sin\left(\frac{k}{1000^{2/d_{model}}}\right), cos\left(\frac{k}{1000^{2/d_{model}}}\right), ..., sin\left(\frac{k}{1000^{2N/d_{model}}}\right), cos\left(\frac{k}{1000^{2N/d_{model}}}\right)\right]
\end{gathered} \tag{7}$$

It is not clear why this encoding was used. But we can list some useful properties: it has small values (we know that sine/cosine bounded from -1 to 1); we can encode large sequences (with large N), because this encoding will give small values even for large positions.

2.2.2 Attention layer

After embedding and encoding layer we have obtained $\mathrm{PE}(\mathbf{H}) = \mathbf{H}^{'} \in \mathbb{R}^{b_{size} \times N \times d_{model}}$. We have already discussed general basics of the attention. However, in this paper authors used more advanced variants of the attention: multi-head self-attention (in the encoder and decoder); and multi-head attention (in the decoder only). Lets discuss the first one.

First, what does it mean self-attention and why? This type of attention can be written as follows:

$$\begin{gathered}
\mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'}) = \mathrm{softmax}\left(\mathrm{score}(\mathbf{Q}, \mathbf{K})\right)\mathbf{V} \in \mathbb{R}^{b_{size} \times N \times d_{model}} \\
\mathrm{score}(\mathbf{Q}, \mathbf{K}) = \frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_k}}, \mathrm{score}(\mathbf{Q}, \mathbf{K}) \in \mathbb{R}^{b_{size} \times N \times N} \\
\mathbf{Q} = \mathbf{H}^{'} \cdot \mathbf{W}_Q, \mathbf{W}_Q \in \mathbb{R}^{d_{model} \times d_k}; \\
\mathbf{K} = \mathbf{H}^{'} \cdot \mathbf{W}_K, \mathbf{W}_K \in \mathbb{R}^{d_{model} \times d_k}; \\
\mathbf{V} = \mathbf{H}^{'} \cdot \mathbf{W}_V, \mathbf{W}_V \in \mathbb{R}^{d_{model} \times d_v}
\end{gathered} \tag{8}$$

What just happened: as you can see we calculate attention between the same sequences opposite to standard attention where we had different inputs. It is called self attention.

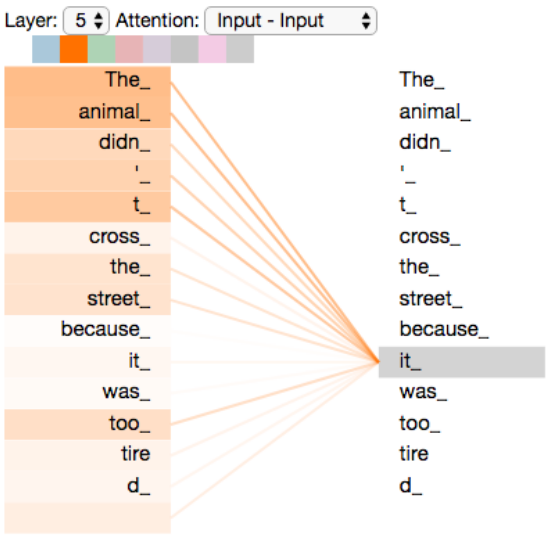

Figure 3: Example of the self attention.

We can illustrate it as in Figure 3. This mechanism works extremely good in DL. Self-attention enables us to find correlations between different words of the input indicating the syntactic and contextual structure of the sentence. Authors indicate three main reasons of the attention successful:

- Good computation complexity (number of operations)
- Low number of the sequential operations (operations that cannot be parallelized)
- Low maximum path length (maximum path between output neuron and input neuron, for example in feed forward network all output neurons connected with input neuron, then this path is zero).

Authors have shown the comparison with convolutional and recurrent layers in Figure 4. As it can be seen self attention layer dominates on recurrent and convolutional layers

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 4: Some comparison

in terms of computational complexity (if $n << d$). Self-attention can be completely parallelized compared to recurrent nets. Also it wins over convolutional in terms of maximum path length, because in conv layers output neurons do not connect with all input neurons.

Now lets discuss multi-head attention. In this case we use different weights $\mathbf{W}_Q, \mathbf{W}_V, \mathbf{W}_K$, apply attention layer with different weights and then concatenates them. Motivation consists in the idea that multi-head attention allows the model to jointly attend to information from different representation subspaces at different position. It can be written as follows:

$$\begin{gathered}\mathbf{MHSA}(\mathbf{H}^{'}, \mathbf{H}^{'}) = Concat\left(\mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'})_1, ..., \mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'})_h\right)\mathbf{W}^O \\ \mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'})_i = \text{softmax}\left(\frac{\mathbf{H}^{'} \cdot \mathbf{W}_Q^i \left(\mathbf{H}^{'} \cdot \mathbf{W}_K^i\right)^\top}{\sqrt{d_k}}\right)\mathbf{H}^{'} \cdot \mathbf{W}_V^i \\ \mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'})_i \in \mathbb{R}^{b_{size} \times N \times d_v}, \mathbf{W}^O \in \mathbb{R}^{hd_v \times d_{model}}\end{gathered} \tag{9}$$

Authors used $h = 8, d_k = d_v = d_{model} = 512$. This operation can be shown as in Figure 5

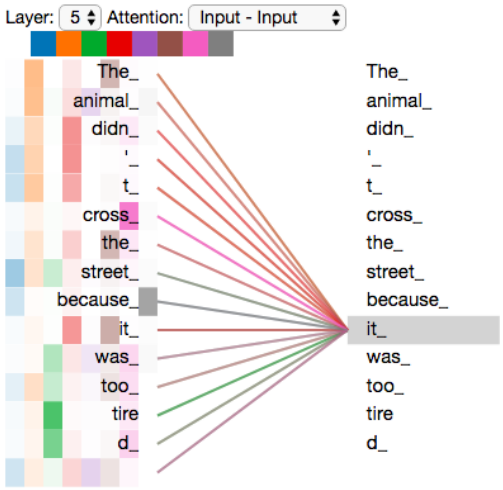

Figure 5: MHSA

In the decoder multi-head self-attention slighty changes. More precisely, in the decoder we have multi-head attention and masked multi-head self-attention. The first one can be written almost the same as previous:

$$\mathbf{MHA}(\mathbf{H}_{de}, \mathbf{H}_{en}) = Concat\left(\mathbf{Attn}(\mathbf{H}_{de}, \mathbf{H}_{en})_1, ..., \mathbf{Attn}(\mathbf{H}_{de}, \mathbf{H}_{en})_h\right)\mathbf{W}^O \tag{10}$$

As you can see here we have not attention between same sequences. We calculate attention between output sequence of the encoder, $\mathbf{H}_{en}$ and currently generated output sequence $\mathbf{H}_{de}$.

However masked multi-head self-attention is a tricky one. It comes from from the fact that in the decoding stage, we predict one word (token) after another. The difference here is that we don't know the whole sentence because it hasn't been produced yet. It can be written as

$$\begin{gathered}\mathbf{MaskAttn}(\mathbf{H}^{'}, \mathbf{H}^{'})_i = \text{softmax}\left(\frac{\mathbf{H}^{'} \cdot \mathbf{W}_Q^i \left(\mathbf{H}^{'} \cdot \mathbf{W}_K^i\right)^\top + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{H}^{'} \cdot \mathbf{W}_V^i \\ \mathbf{M} \in \mathbb{R}^{b_{size} \times N \times N}, \mathbf{M}_{kj} = \begin{cases} -\infty, \text{if } j > k \\ 0, \text{otherwise} \end{cases}\end{gathered} \tag{11}$$

To understand masked multi-head attention we have to remember that we want to train our model in parallel not in sequential mode. Due to the mask **we can realize parrallel training** without any loops. For example, we want to translate sequence that has T length. Then, to make translation parallel we can assign to worker 1 a prediction of first word; to worker 2 a prediction of the second word knowing the real first one and masked from second to Tth words; to worker 3 a prediction of the third word knowing the first and second one and masked from the fourth to to Tth words; and so on. It can be done via the data paralell. However **the inference stage cannot be parallel**. We need to generate words sequentially.

Lets highlight one more important thing. The question may arise: in training we know the length of the output sequence (and we apply mask to it), but **in inference we do not know the output length, what shall we do?** We should clarify the following fact: length of the sequences that transformer can generated is limited!!! So, if we do not know the output length we just take the maximum one and generate until we meet key word. In deep learning this key word is [PAD]. Or we change this length when predicted new word started with length = 1?

2.2.3 Normalization, skip, feed forward

After positional encoding, multi-head self-attention we are applying normalization and skip connection. It can be written as follows:

$$\mathbf{H}^{'} = \mathbf{H} + \mathbf{LNorm}(\mathbf{MHSA}(\mathbf{H}, \mathbf{H})) \tag{12}$$

In this case we are using layer normalization, this type of normalization is shown in Figure 6. The reason of using layer normalization is that batch normalization works

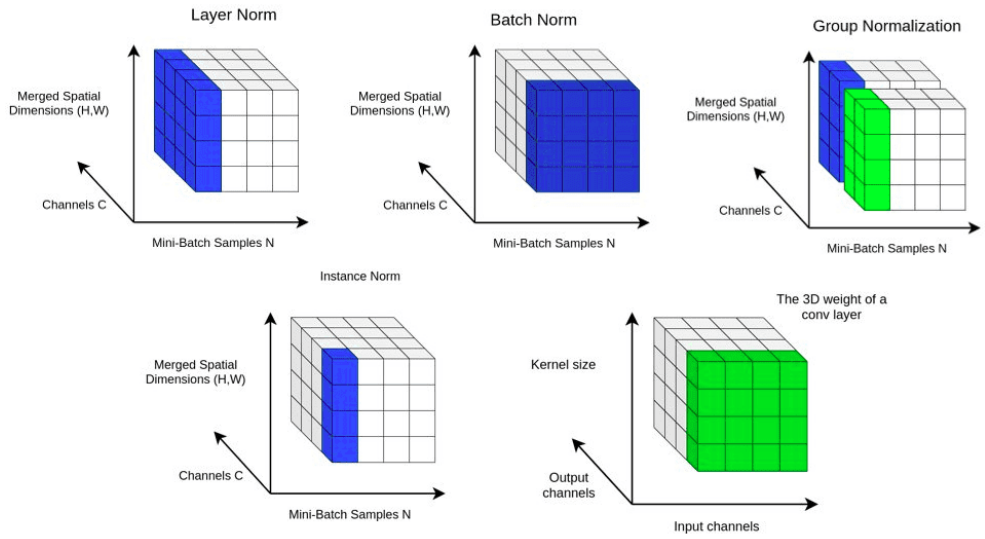

Figure 6: Different normalizations

bad for small batches, but layer normalization works fine. After that we use usual feed forward layer with skip connection and layer normalization.

2.2.4 Output layer

As the output of the decoder we want to have probability of next word (token) that should be generated. If we work sequentially, then our output would have the shape $\mathbf{O} \in \mathbb{R}^{b_{size} \times n_{tokens}}$. That is, for each example in batch we generate probability of each token and then take the word with the highest probability.

However, we want to work without any loops (i.e. in parallel), thus this type of output is not our choice. As output we take $\mathbf{O} \in \mathbb{R}^{b_{size} \times N \times n_{tokens}}$, i.e. we add length of the sequence N. We apply feed forward and softmax to obtain the output. The meaning of this N depends on the stage: training or inference.

- **Training**
  In training stage we are working in parallel mode. Thus, in this case N is the length of the real output sequence. We apply masking so that the neural network simply does not copy the correct answers.

- **Inference**
  In inference stage N changes with every step because of the sequential style of working. In the first step we put into decoder sequence with $N = 1$ (special word [START]) and generate following (the second) word. In the second step we have $N = 2$ (special word and generated one from the previous step). We pass it through the decoder and as a prediction we take the last item in the generated sequence. That is, if output of the decoder is $\mathbf{O} \in \mathbb{R}^{b_{size} \times N \times n_{tokens}}$, the the next word is $max(\mathbf{O}, dim = 2)[:, -1]$. We continue this procedure until we meet special word [END] that means the stopping critetion.

3 Visual Transformer

3.1 Introduction

So, now lets move on from sequences to images. We know that convolutions have some inductive bias to images. I mean the following:

1. **Weight sharing**. We use same filter for whole image. This leads to local dependencies between pixels. For some computer vision tasks this property is useful. For example, for classification we have to look only on small piece of image to make a decision. Moreover, weight sharing allows to build very deep networks and extract useful features.

2. **Translation equivariant** and learnable **Translation invariant**, this makes possible to be generalized. This makes convnets **inductive biased** to computer vision problem.

3. **Data structure accounting**, in convnets we process image as image, while in fully connected we build a vector and lose info about structure of data.

So, transformers lack the inductive biases like Convolutional Neural Networks. Because of this, vision transformers shows worse results that convolution nets for small datasets. However, for large datasets the situation is changes. From my point of view, if dataset

is large then transformer networks are able to learn features which convolution has. To successfully apply vision transformer for you task it is necessary to take pretrained transformer on a huge dataset and then fune-tune it.

The main difference between CNN and self-attention layers is that the new value of a pixel depends on every other pixel of the image. As opposed to convolution layers whose receptive field is the $k \times k$ neighborhood grid, the self-attention's receptive field is always the full image. In other words, self-attention mostly focused on long-range dependencies, but convolution on local dependencies. I think that good idea is to combine these two approaches: one catch global dependencies, another - local. Usually, researchers build architectures which contain self-attention blocks in the middle of the network [ZGMO19] (i still have no complete understanding of this phenomena)

3.2 Architecture of the ViT

We know that transformers work with sequences. So, we need to understand how to present images as sequences. This is done as follows: sequence creates from spatial coordinates and each element of the sequence is a vector which dimension is equal to the number of channel in image. It is shown in Figure 7. Here we have shown

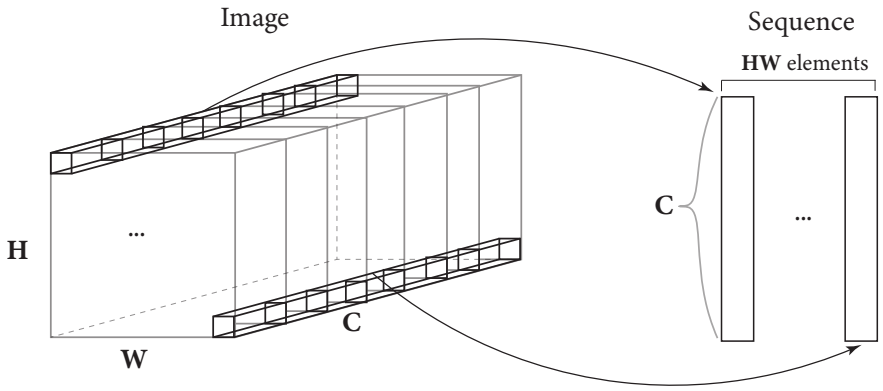

Figure 7: From image to sequence

a translation in which only one element from every channel is taken. However it is possible to take several elements and it is called patch. Image patches are treated the same way as tokens (words) in an NLP application. This technique was used in the ViT paper. The main motivation of using patches instead of pixels is computational complexity. As we know self attention has quadratic complexity with respect to length of the sequence. So, with quadratic cost in the number of pixels, this does not scale to realistic input sizes. But with patching length of sequence become smaller, because we use group of pixels.

The architecture of the model is presented in Figure 8. Firstly, as you can see we pack image into patches:

$$\mathbf{x} \in \mathbb{R}^{b_{size} \times C \times H \times W} \underbrace{\longrightarrow}_{\text{Patching}} \mathbf{x}_p \in \mathbb{R}^{b_{size} \times N \times \left(\frac{H \cdot W}{N} \cdot C\right)} \tag{13}$$

Here N is a number of patches, $\left(\sqrt{\frac{H \cdot W}{N}}, \sqrt{\frac{H \cdot W}{N}}\right)$ is the resolution of each image patch. After patching we make a linear projection (it is common linear layer) and add positional

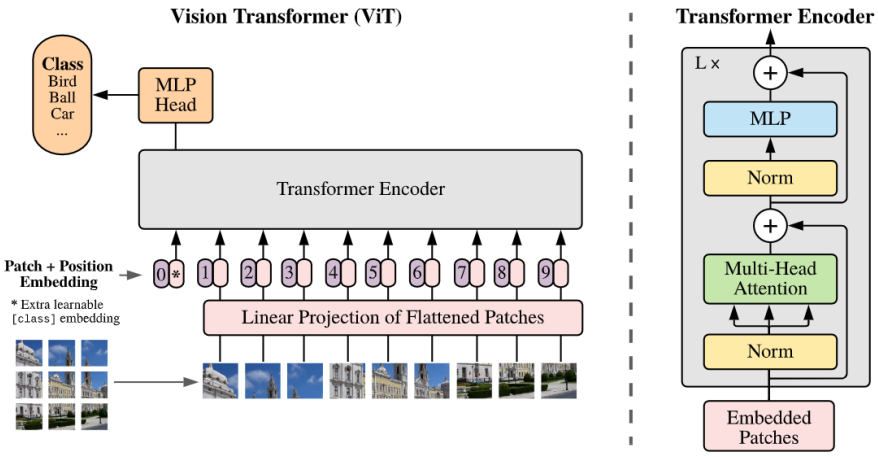

Figure 8: The architecture of the ViT.

embedding. Here we need to note one important thing: besides the image patches we also consider the extra learnable [class] embedding. In other words, we take embedding layer, put [class] token into this embedding and add the result as the zero*th* element in the sequence. This element is responsible for further classification vit the classification head. Both during pre-training and fine-tuning, a classification head is attached to this embedding. The classification head is implemented by a MLP with one hidden layer at pre-training time and by a single linear layer at fine-tuning time. The whole Visual Transformer can be described as follows:

- **Embedding layer**

$$\mathbf{z}_0 = \underbrace{\left[\mathbf{x}_{class}, \mathbf{x}_p^1, ..., \mathbf{x}_p^N\right]}_{\mathbb{R}^{b_{size} \times (N+1) \times \left(\frac{H \cdot W}{N} \cdot C\right)}} \mathbf{E} + \mathbf{E}_{pos}, \mathbf{E} \in \mathbb{R}^{\left(\frac{H \cdot W}{N} \cdot C\right) \times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \tag{14}$$

- **Transformer Encoder** (apply L times)

$$\begin{aligned} \mathbf{z}'_l &= \text{MHSA}\left(\text{LN}(\mathbf{z}_{l-1}), \text{LN}(\mathbf{z}_{l-1})\right) + \mathbf{z}_{l-1}, \;\; l = 1, ..., L \\ \mathbf{z}_l &= \text{MLP}\left(\text{LN}(\mathbf{z}'_l)\right) + \mathbf{z}'_l, \;\; l = 1, ..., L \end{aligned} \tag{15}$$

- **MLP head**

$$\mathbf{y} = \text{Softmax}\left(\text{MLP}(\text{LN}(\mathbf{z}_L^0))\right) \tag{16}$$

Authors note: We note that Vision Transformer has much less image-specific inductive bias than CNNs. In CNNs, locality, two-dimensional neighborhood structure, and translation equivariance are baked into each layer throughout the whole model. In ViT, only MLP layers are local and translationally equivariant, while the self-attention layers are global. The two-dimensional neighborhood structure is used very sparingly: in the beginning of the model by cutting the image into patches and at fine-tuning time for adjusting the position embeddings for images of different resolution (as described below). Other than that, the position embeddings at initialization time carry no information about the 2D positions of the patches and all spatial relations between the patches have to be learned from scratch.

3.2.1 Self attention for images

We flattened images to vector and then pass it through the multi-head self-attention layer. In this layer we use matrix multiplication. However we can make it without destroying the structure of the image and use convolution operations. Lets remember the attention layer:

$$\mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'})_i = \text{softmax}\left(\frac{\mathbf{H}^{'} \cdot \mathbf{W}_Q^i \left(\mathbf{H}^{'} \cdot \mathbf{W}_K^i\right)^\top}{\sqrt{d_k}}\right) \mathbf{H}^{'} \cdot \mathbf{W}_V^i \tag{17}$$

where $\cdot$ is a matrix multiplication operation. If H' is an image, i.e. $H' \in \mathbb{R}^{b_{size} \times C \times H \times W}$, we flatten it into vector - $H' \in \mathbb{R}^{b_{size} \times (H \cdot W) \times C}$ and then apply fully connected layer - $\mathbf{H}^{'} \cdot \mathbf{W}_Q^i, \mathbf{W}_Q^i \in \mathbb{R}^{C \times C'}$. But we can apply convolution operation and achieve the same result:

$$\mathbf{Attn}(\mathbf{H}^{'}, \mathbf{H}^{'})_i = \text{softmax}\left(\frac{\mathbf{H}^{'} * \mathbf{W}_Q^i \left(\mathbf{H}^{'} * \mathbf{W}_K^i\right)^\top}{\sqrt{d_k}}\right) \mathbf{H}^{'} * \mathbf{W}_V^i \tag{18}$$

Here $\mathbf{W}_Q^i \in \mathbb{R}^{C \times C' \times 1 \times 1}, \mathbf{H}^{'} * \mathbf{W}_Q^i \in \mathbb{R}^{b_{size} \times C' \times H \times W}$. As you can see it is convolutional layer with 1×1 kernel. To realize matrix product between $\mathbf{H}^{'} * \mathbf{W}_Q^i$ and $\left(\mathbf{H}^{'} * \mathbf{W}_K^i\right)^\top$ we can use einsum. Unfortunately, to apply softmax function over dimension of sequence length we have to flatten and as a consequence destroy image structure. It is happens because dimension of sequence is $H \times W$ and we cant take softmax over several dimension. Of course, we can realize softmax over multiple dimension by ourselves, however it would be more time-consuming than built-in implementation. Implementation is shown in Figure 9.

```
batchsize, C, H, W = x.size()
N = H * W                                       # Number of features
f_x = self.key(x).view(batchsize,   -1, N)      # Keys                  [B, C_bar, N]
g_x = self.query(x).view(batchsize, -1, N)      # Queries               [B, C_bar, N]
h_x = self.value(x).view(batchsize, -1, N)      # Values                [B, C_bar, N]

s =  torch.bmm(f_x.permute(0,2,1), g_x)         # Scores                [B, N, N]
beta = self.softmax(s)                          # Attention Map         [B, N, N]

v = torch.bmm(h_x, beta)                        # Value x Softmax       [B, C_bar, N]
v = v.view(batchsize, -1, H, W)                 # Recover input shape   [B, C_bar, H, W]
o = self.self_att(v)                            # Self-Attention output [B, C, H, W]

y = self.gamma * o + x                          # Learnable gamma + residual
return y, o
```

Figure 9: Self attention for images

Equivalence of the 1×1 convolutional layer and linear layer is shown in Figure 10

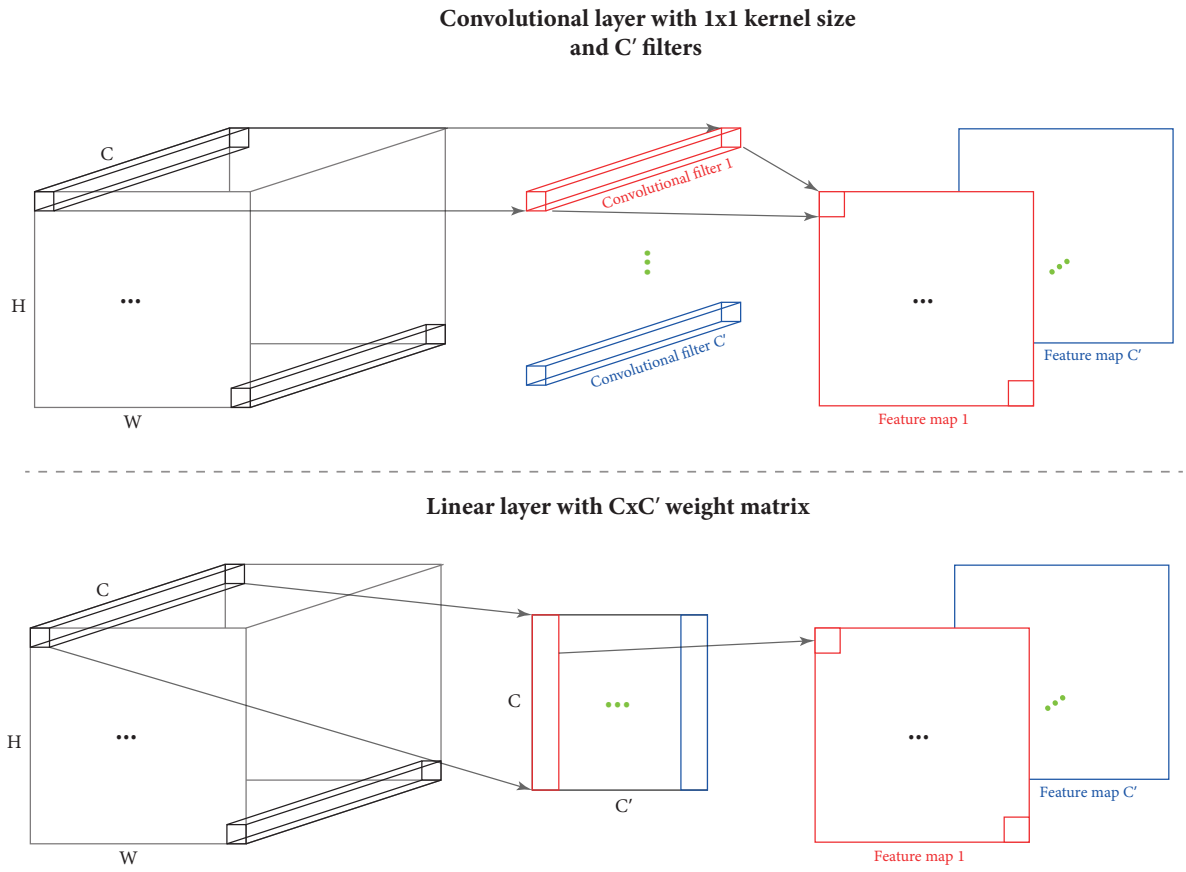

Figure 10: Equivalence of conv layer and linear layer

References

[CLJ19] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

[DBK+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[ZGMO19] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. In *International conference on machine learning*, pages 7354–7363. PMLR, 2019.