ViT (Vision Transformer)

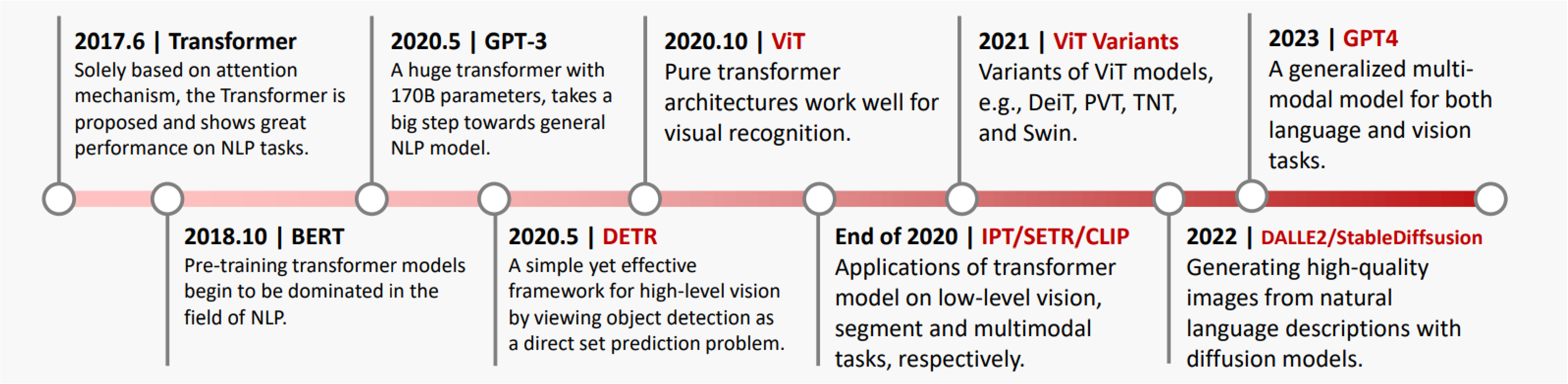


Fig. 1: Key milestones in the development of transformer. The vision transformer models are marked in red.

https://arxiv.org/pdf/2012.12556

ViT (Vision Transformer)

Published as a conference paper at ICLR 2021

AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

Alexey Dosovitskiy[*,†], **Lucas Beyer**[*], **Alexander Kolesnikov**[*], **Dirk Weissenborn**[*], **Xiaohua Zhai**[*], **Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby**[*,†]

[*]equal technical contribution, [†]equal advising

Google Research, Brain Team

{adosovitskiy, neilhoulsby}@google.com

ABSTRACT

While the Transformer architecture has become the de-facto standard for natural language processing tasks, its applications to computer vision remain limited. In vision, attention is either applied in conjunction with convolutional networks, or used to replace certain components of convolutional networks while keeping their overall structure in place. We show that this reliance on CNNs is not necessary and a pure transformer applied directly to sequences of image patches can perform very well on image classification tasks. When pre-trained on large amounts of data and transferred to multiple mid-sized or small image recognition benchmarks (ImageNet, CIFAR-100, VTAB, etc.), Vision Transformer (ViT) attains excellent results compared to state-of-the-art convolutional networks while requiring substantially fewer computational resources to train.[1]

https://arxiv.org/abs/2010.11929

Part 1. Hight Level Overview

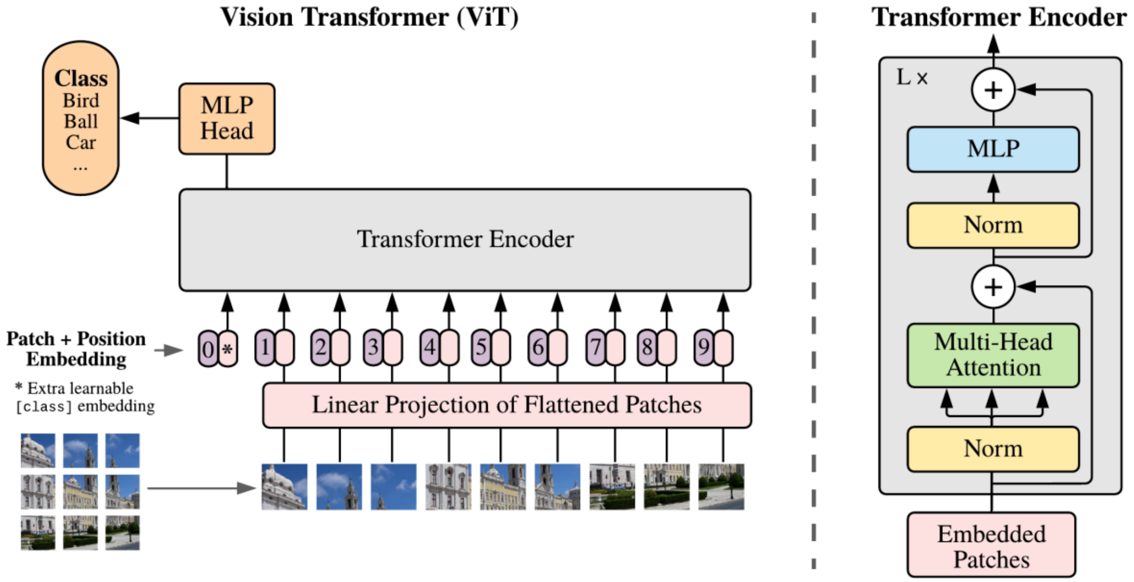


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

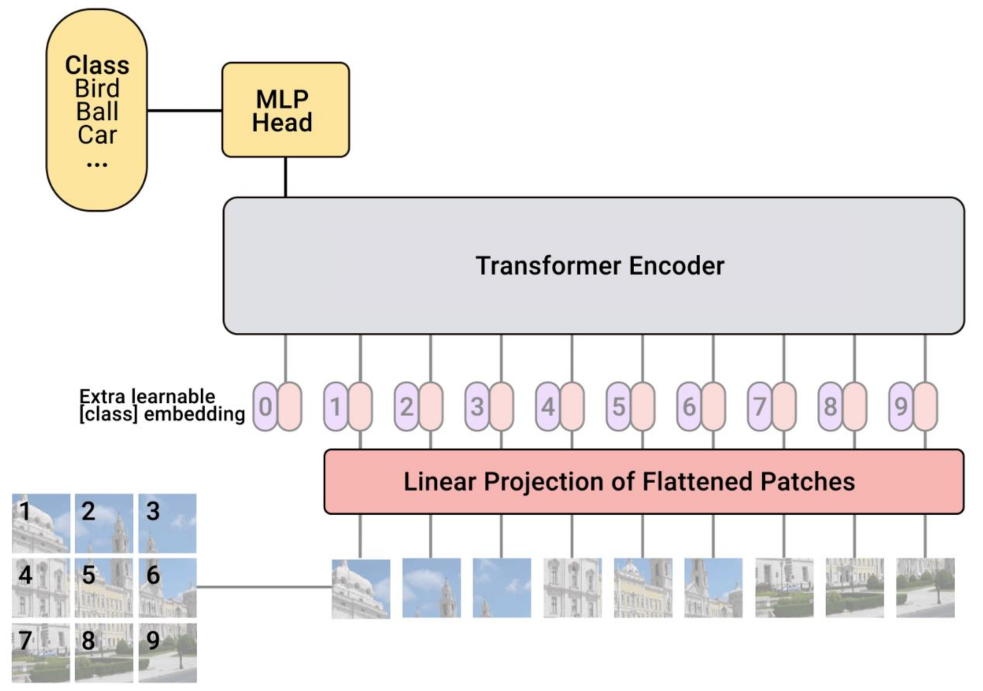
ViT (Vision Transformer)

http://research.google/blog/transformers-for-image-recognition-at-scale/

https://1.bp.blogspot.com/-_mnVfmzvJWc/X8gMzhZ7SkI/AAAAAAAAG24/8gW2AHEoqUQrBwOqjhYB37A7OOjNyKuNgCLcBGAsYHQ/s1600/image1.gif

Google Research   Who we are   Research areas   Our work   Programs & events   Careers

ViT divides an image into a grid of square patches. Each patch is flattened into a single vector by concatenating the channels of all pixels in a patch and then linearly projecting it to the desired input dimension. Because Transformers are agnostic to the structure of the input elements we add learnable position embeddings to each patch, which allow the model to learn about the structure of the images. *A priori*, ViT does not know about the relative location of patches in the image, or even that the image has a 2D structure — it must learn such relevant information from the training data and encode structural information in the position embeddings.



https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

- in general (not only for ViT) we create a network from blocks that consists of several layers

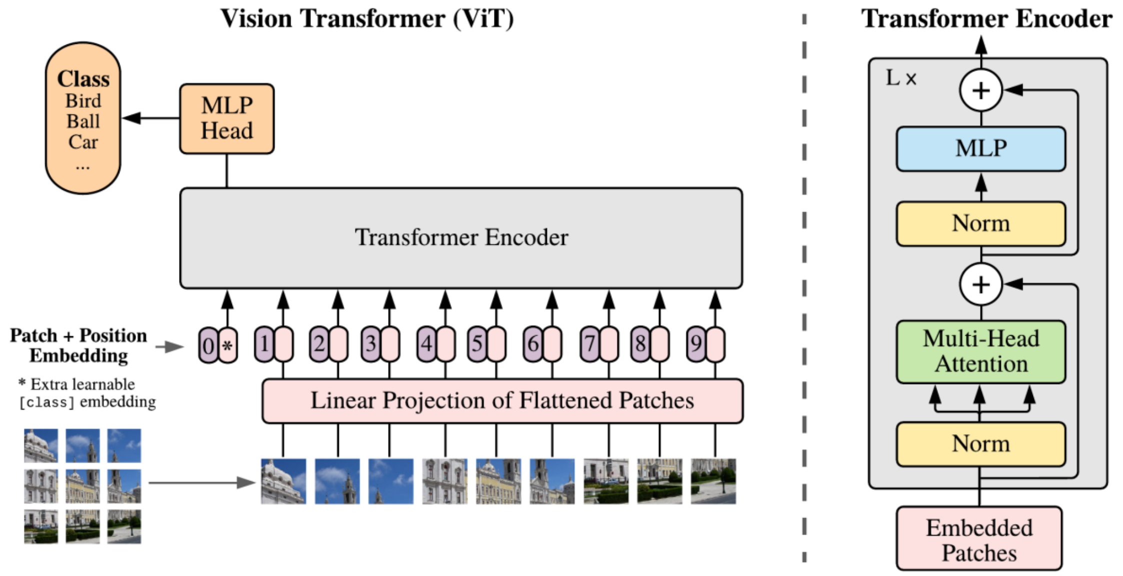


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

- in general (not only for ViT) we create a network from blocks that consists of several layers
- e.g. inception blocks, residual blocks, **transformer encoder blocks**
- what is input for these models (blocks)?

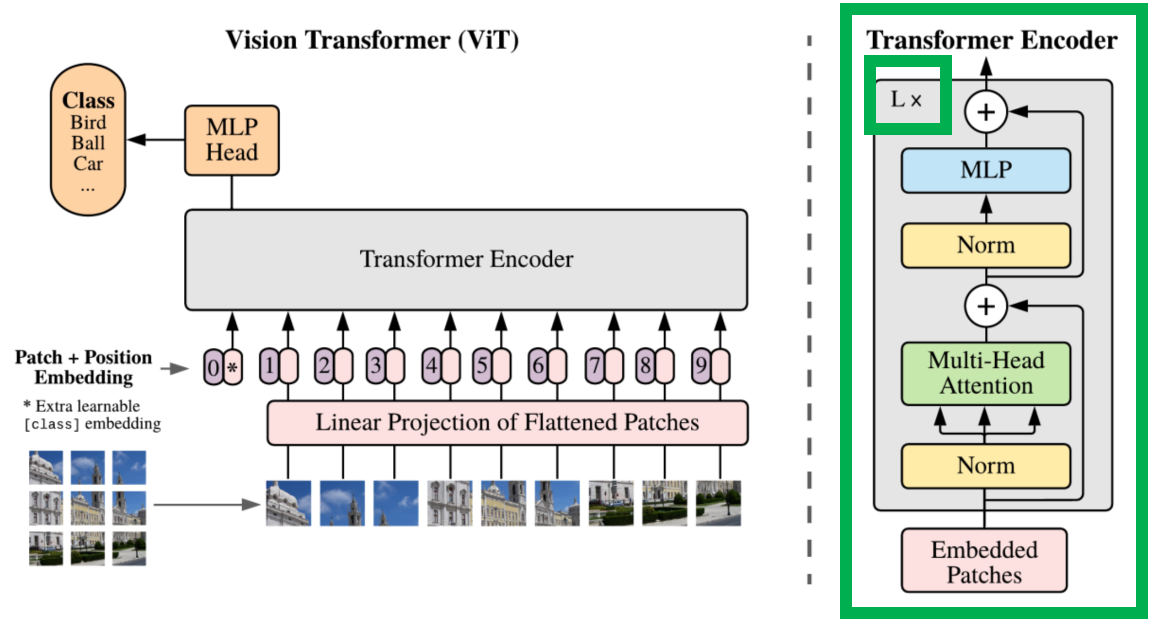


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

- in general (not only for ViT) we create a network from blocks that consists of several layers
- e.g. inception blocks, residual blocks, **transformer encoder blocks**
- what is input for these models (blocks)?
- In the case of CNN, we use raw images
- In the case of **ViT,** we use **fixed size patches**

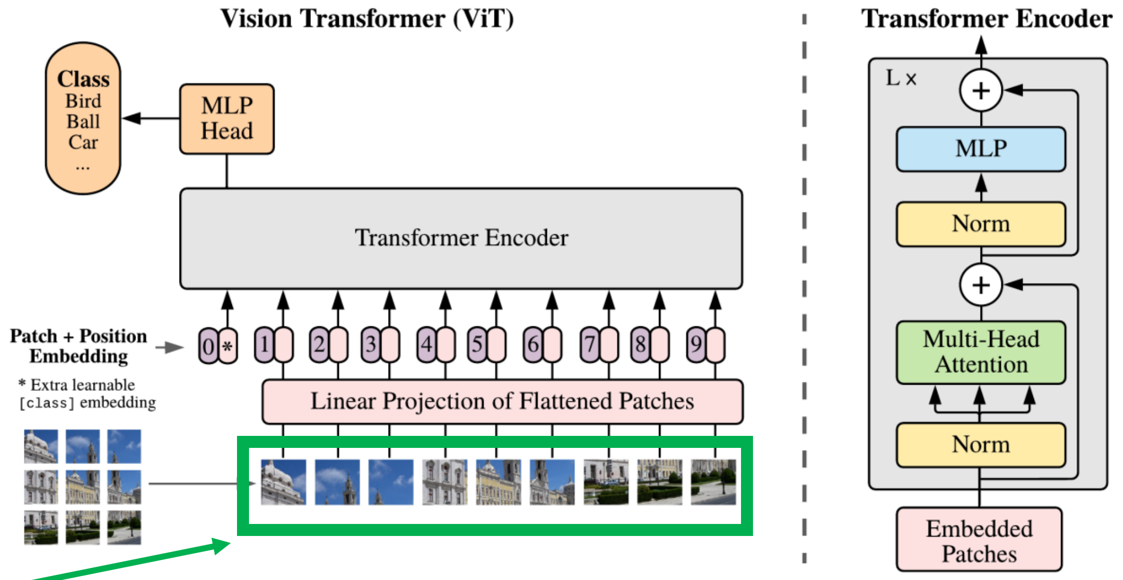


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

1. Divide the input image into smaller **fixed-size patches** (e.g. 16×16 pixels). Each patch will act as a "token" for the transformer.

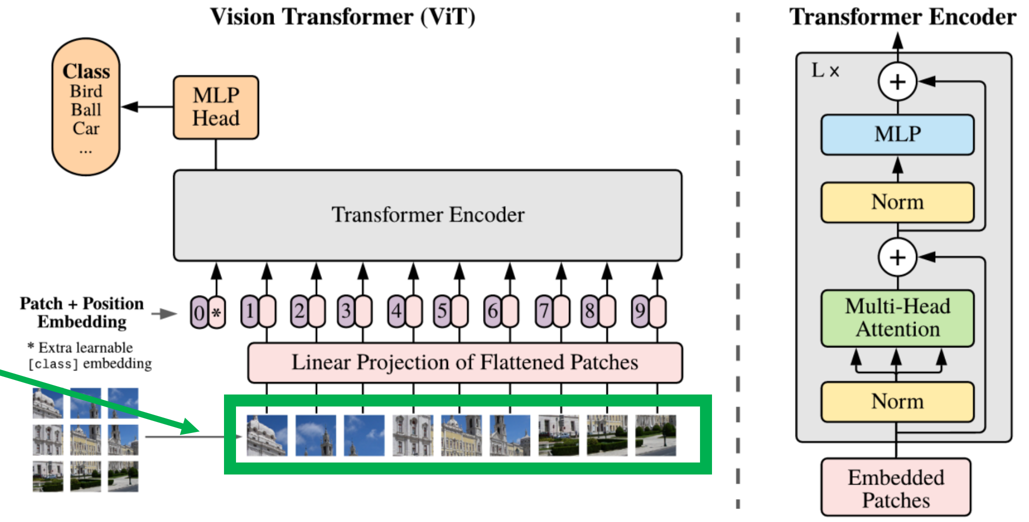


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

1. Divide the input image into smaller **fixed-size patches** (e.g. 16×16 pixels). Each patch will act as a "token" for the transformer.

2. Add positional encoding to encode the patch's position in the original image. These encodings are typically sine/cosine functions or learnable parameters.

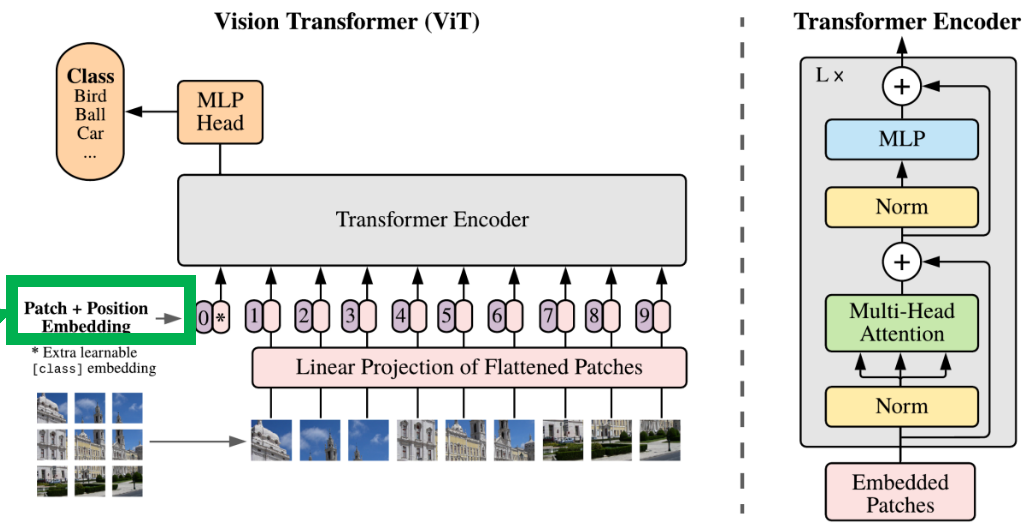


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

1. Divide the input image into smaller **fixed-size patches** (e.g. 16×16 pixels). Each patch will act as a "token" for the transformer.

2. Add positional encoding to encode the patch's position in the original image. These encodings are typically sine/cosine functions or learnable parameters.

3. Add classification token - this token is used for classification tasks.

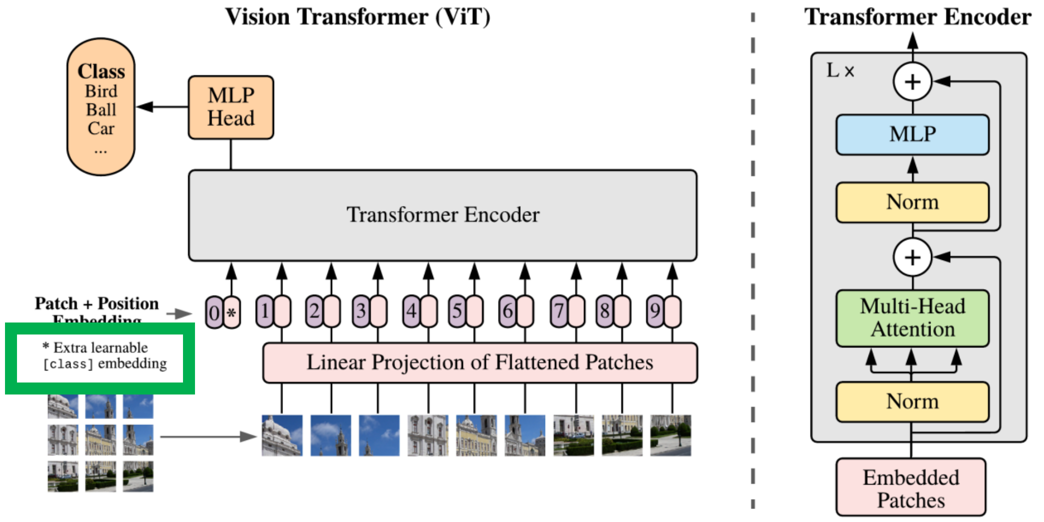


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

1. Divide the input image into smaller **fixed-size patches** (e.g. 16×16 pixels). Each patch will act as a "token" for the transformer.

2. Add positional encoding to encode the patch's position in the original image. These encodings are typically sine/cosine functions or learnable parameters.

3. Add classification token - this token is used for classification tasks.

4. The Transformer encoder processes - enables each token to attend to every other token

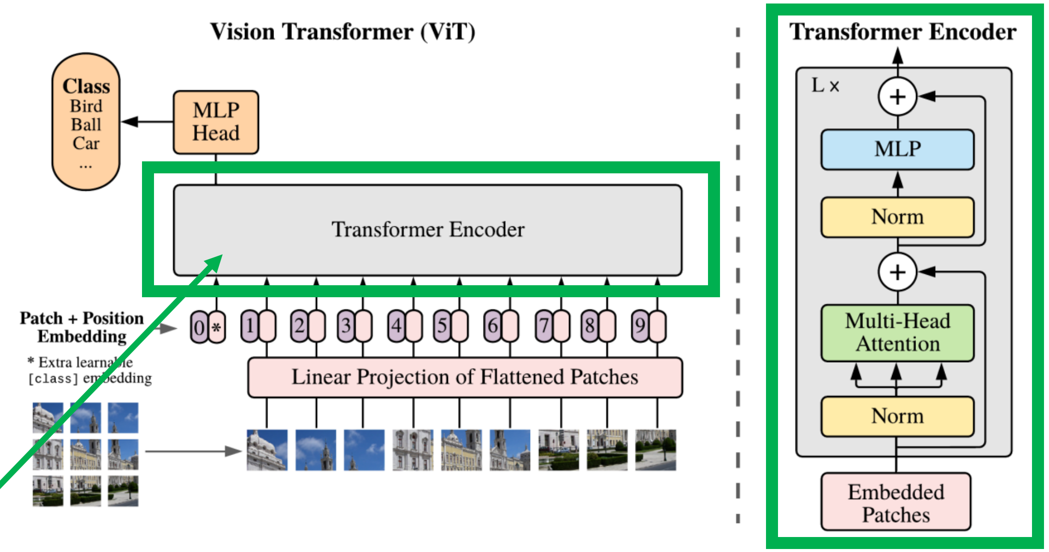


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

High Level Overview

1. Divide the input image into smaller **fixed-size patches** (e.g. 16×16 pixels). Each patch will act as a "token" for the transformer.
2. Add positional encoding to encode the patch's position in the original image. These encodings are typically sine/cosine functions or learnable parameters.
3. Add classification token - this token is used for classification tasks.
4. The Transformer encoder processes - enables each token to attend to every other token
5. Classification Head - output predictions based on the classification token.

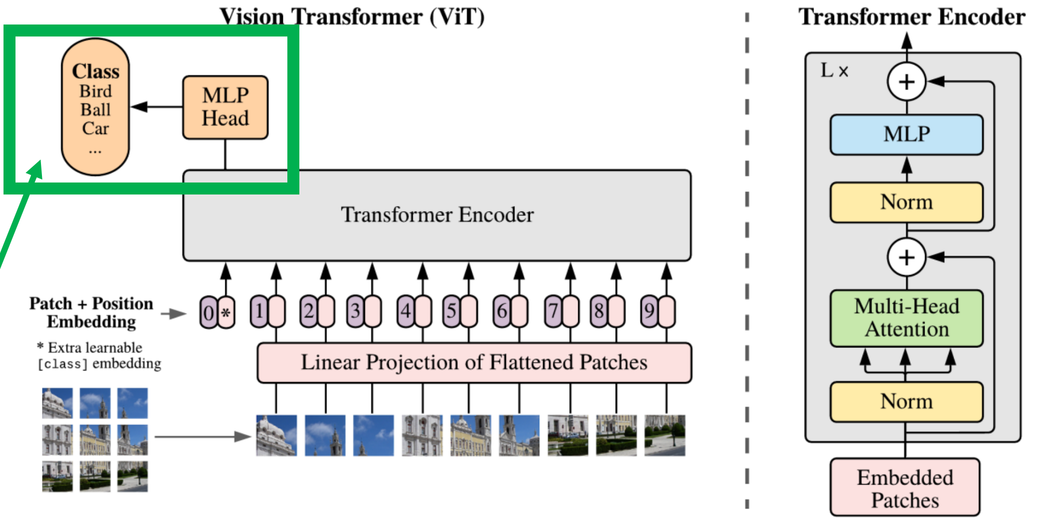


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

Attention Is All You Need

https://arxiv.org/pdf/1706.03762

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Part 2. A Bit More Detail

LayerNorm

CLASS torch.nn.LayerNorm(*normalized_shape, eps=1e-05, elementwise_affine=True, bias=True, device=None, dtype=None*) [SOURCE]

Applies Layer Normalization over a mini-batch of inputs.

This layer implements the operation as described in the paper Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated over the last D dimensions, where D is the dimension of `normalized_shape`. For example, if `normalized_shape` is `(3, 5)` (a 2-dimensional shape), the mean and standard-deviation are computed over the last 2 dimensions of the input (i.e. `input.mean((-2, -1))`). γ and β are learnable affine transform parameters of `normalized_shape` if `elementwise_affine` is `True`. The standard-deviation is calculated via the biased estimator, equivalent to *torch.var(input, unbiased=False)*.

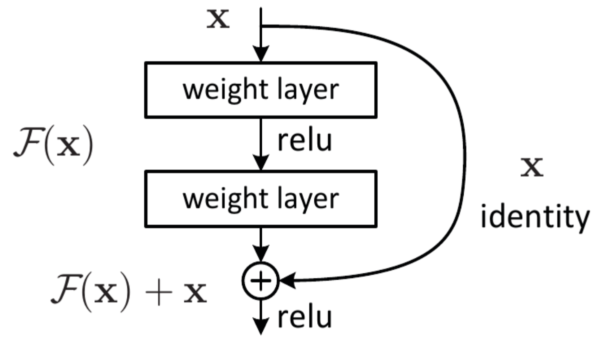
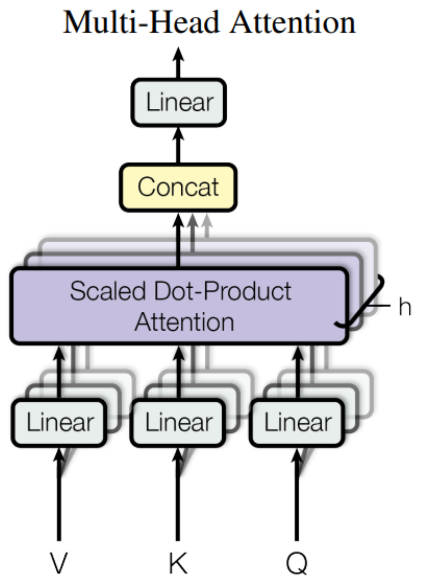


Figure 2. Residual learning: a building block.



https://arxiv.org/abs/2010.11929
https://arxiv.org/abs/1512.03385

ViT (Vision Transformer)

Step 1 - Input

- transform image into 16 x 16 size (patches)
- embed each patch into 768 dimension
- i.e. one patch can be described with 1 x 768 values
- In the case that we have 196 patches with size of 16 x 16, we obtain [14, 14, 768] tensor
- with the use of flatten, we obtain [196, 768] matrix

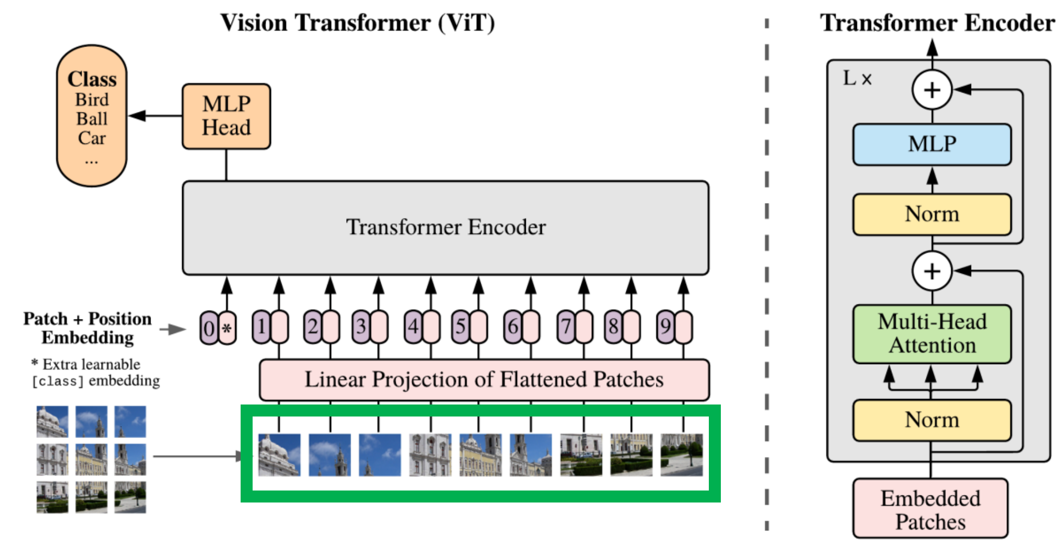


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

An overview of the model is depicted in Figure 1. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers, so we flatten the patches and map to D dimensions with a trainable linear projection (Eq. 1). We refer to the output of this projection as the patch embeddings.

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Step 1 - Input

- transform image into 16 x 16 size (patches)
- embed each patch into 768 dimensions
- i.e. one patch can be described by 1 x 768 values
- In the case that we have 196 patches with size of 16 x 16, we obtain [14, 14, 768] tensor
- with the use of flatten, we obtain [196, 768] matrix

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

An overview of the model is depicted in Figure 1. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers, so we flatten the patches and map to D dimensions with a trainable linear projection (Eq. 1). We refer to the output of this projection as the patch embeddings.

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Step 1 - Input

- transform image into 16 x 16 size (patches)
- embed each patch into 768 dimensions
- i.e. one patch can be described by 1 x 768 values
- In the case that we have 196 patches with size of 16 x 16, we obtain [14, 14, 768] tensor
- with the use of flatten, we obtain [196, 768] matrix

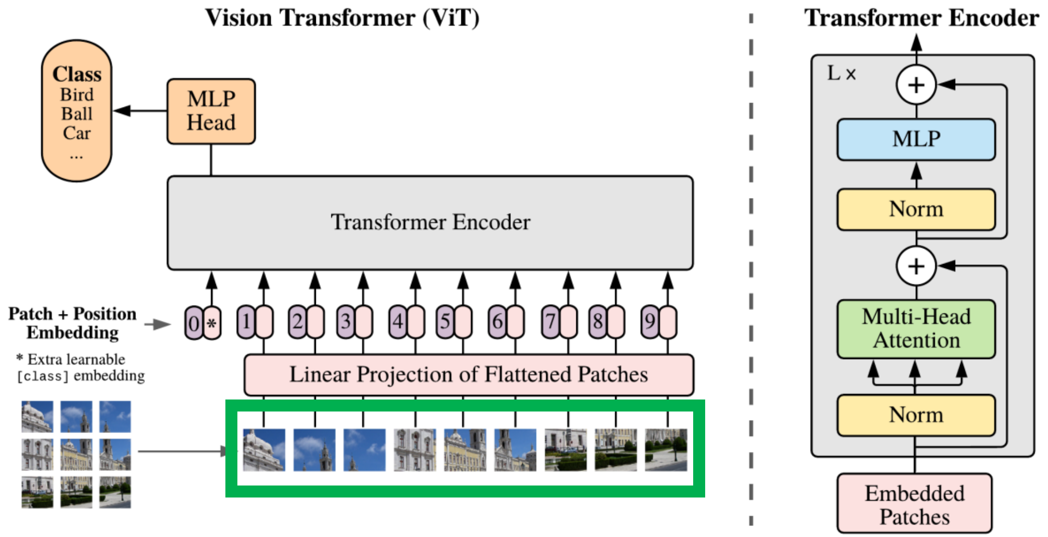


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

An overview of the model is depicted in Figure 1. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers, so we flatten the patches and map to D dimensions with a trainable linear projection (Eq. 1). We refer to the output of this projection as the patch embeddings.

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Step 1 - Input

- transform image into 16 x 16 size (patches)
- embed each patch into 768 dimensions
- i.e. one patch can be described by 1 x 768 values
- In the case that we have 196 patches with size of 16 x 16, we obtain [14, 14, 768] tensor
- with the use of flatten, we obtain [196, 768] matrix

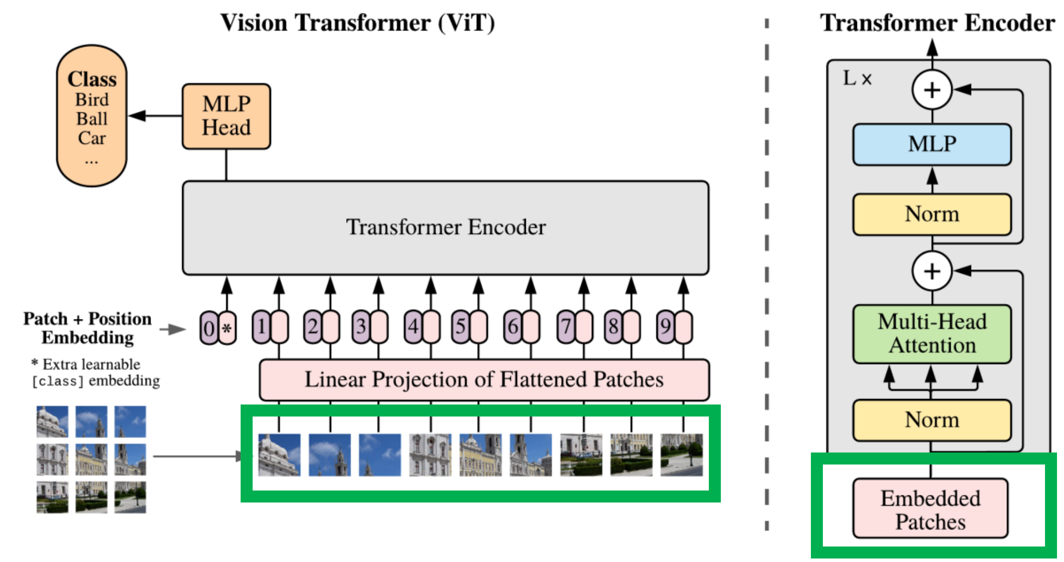


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

An overview of the model is depicted in Figure 1. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers, so we flatten the patches and map to D dimensions with a trainable linear projection (Eq. 1). We refer to the output of this projection as the patch embeddings.

https://arxiv.org/abs/2010.11929

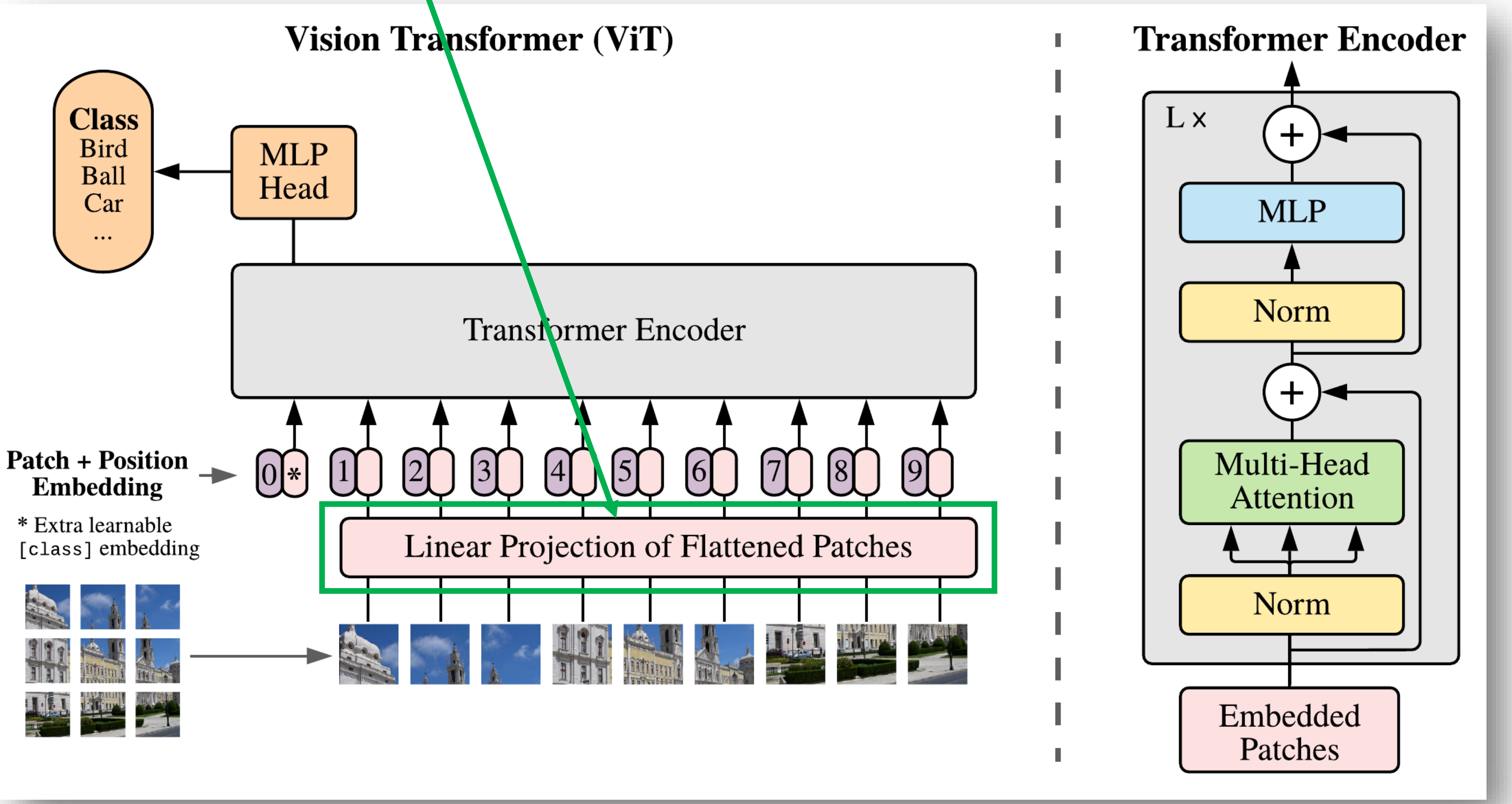
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots ; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \qquad (1)$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \qquad (2)$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \qquad (3)$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \qquad (4)$$



https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

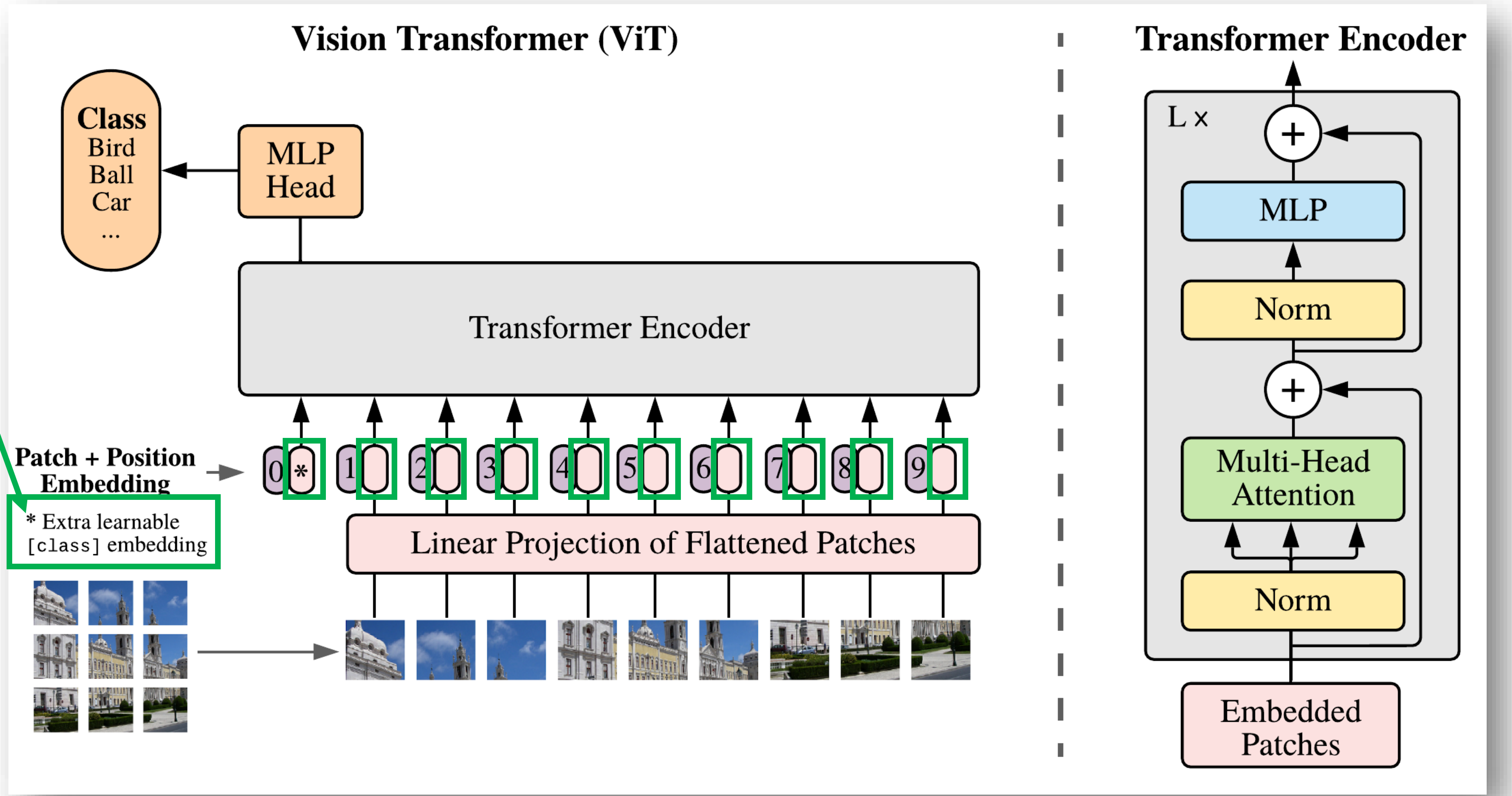
$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C)\times D}, \ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \quad (2)$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \quad (3)$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \quad (4)$$

For classification purposes, the same as the class token used in BERT.



https://arxiv.org/abs/2010.11929

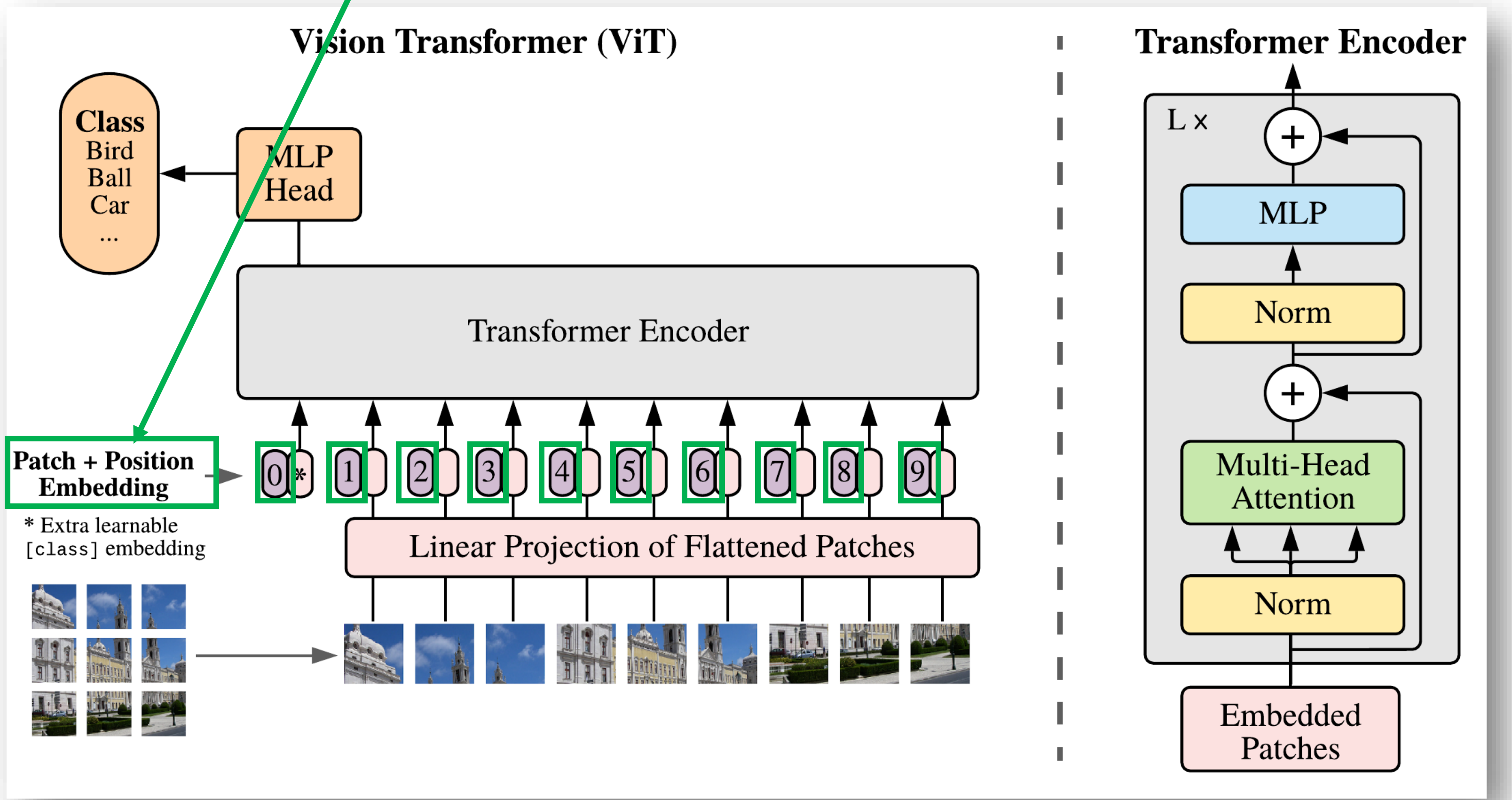
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_\text{class};\, \mathbf{x}^1_p\mathbf{E};\, \mathbf{x}^2_p\mathbf{E}; \cdots;\, \mathbf{x}^N_p\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \tag{1}$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}^0_L) \tag{4}$$



https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

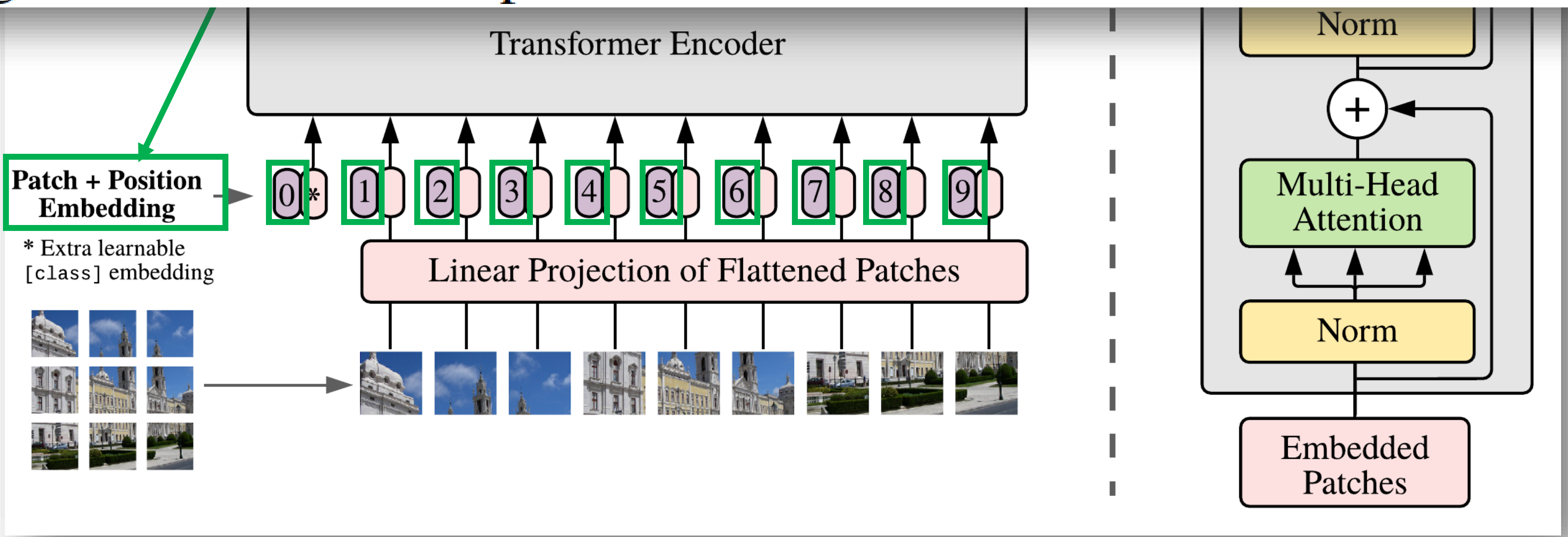
$$\mathbf{z}_0 = [\mathbf{x}_\text{class}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots ; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C)\times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

$$\mathbf{z'}_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \quad (2)$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \quad (3)$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0) \quad (4)$$

Position embeddings are added to the patch embeddings to retain positional information. We use standard learnable 1D position embeddings, since we have not observed significant performance gains from using more advanced 2D-aware position embeddings (Appendix D.4). The resulting sequence of embedding vectors serves as input to the encoder.



https://arxiv.org/abs/2010.11929

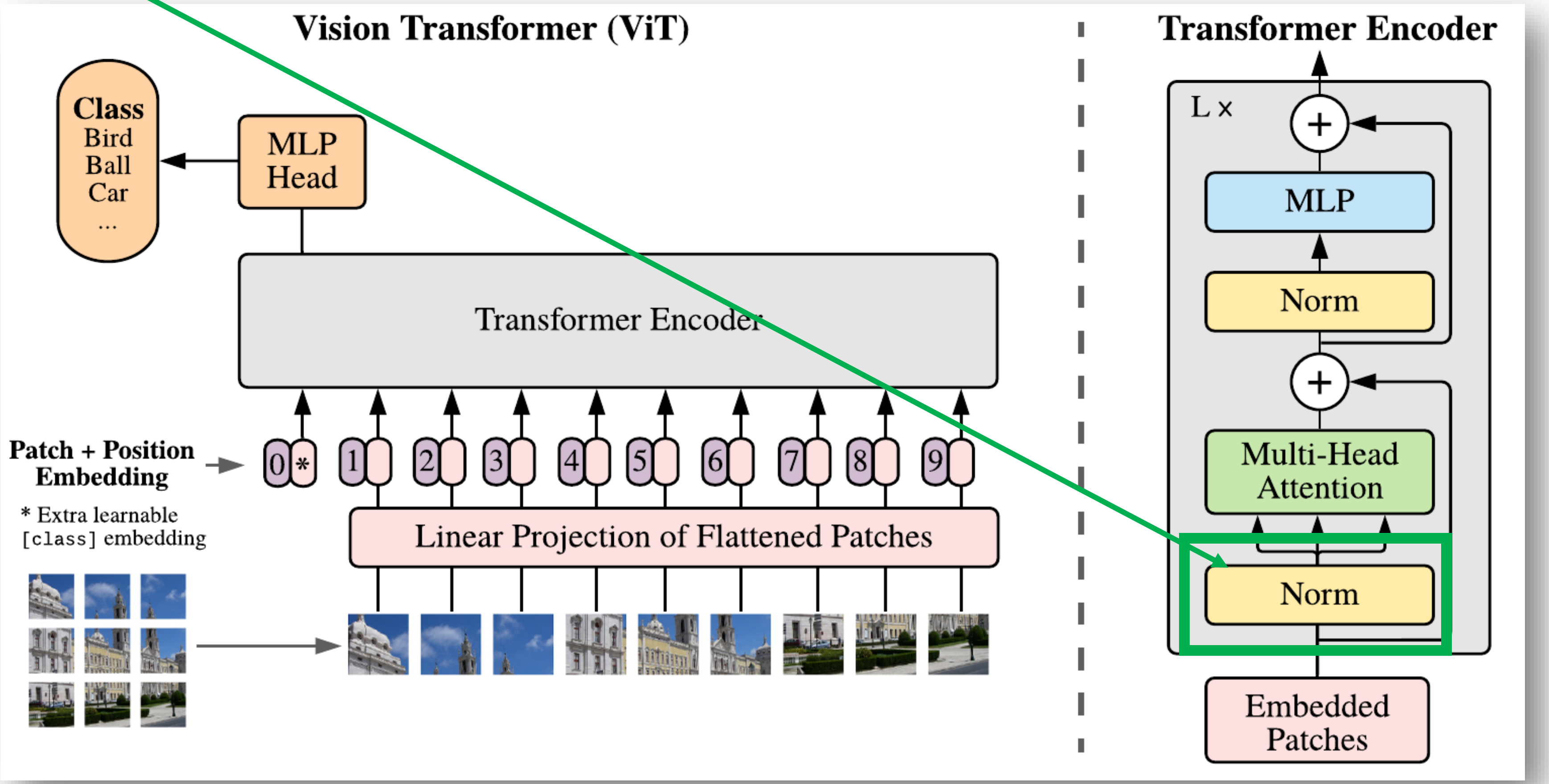
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E}; \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \quad (1)$$

$$\mathbf{z'}_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \quad (2)$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \quad (3)$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0) \quad (4)$$



https://arxiv.org/abs/2010.11929

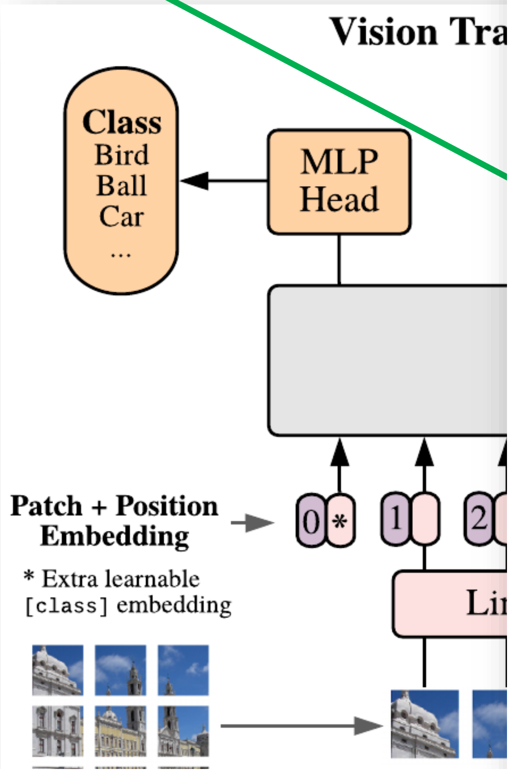
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] \quad (1)$$

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \quad (2)$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \quad (3)$$

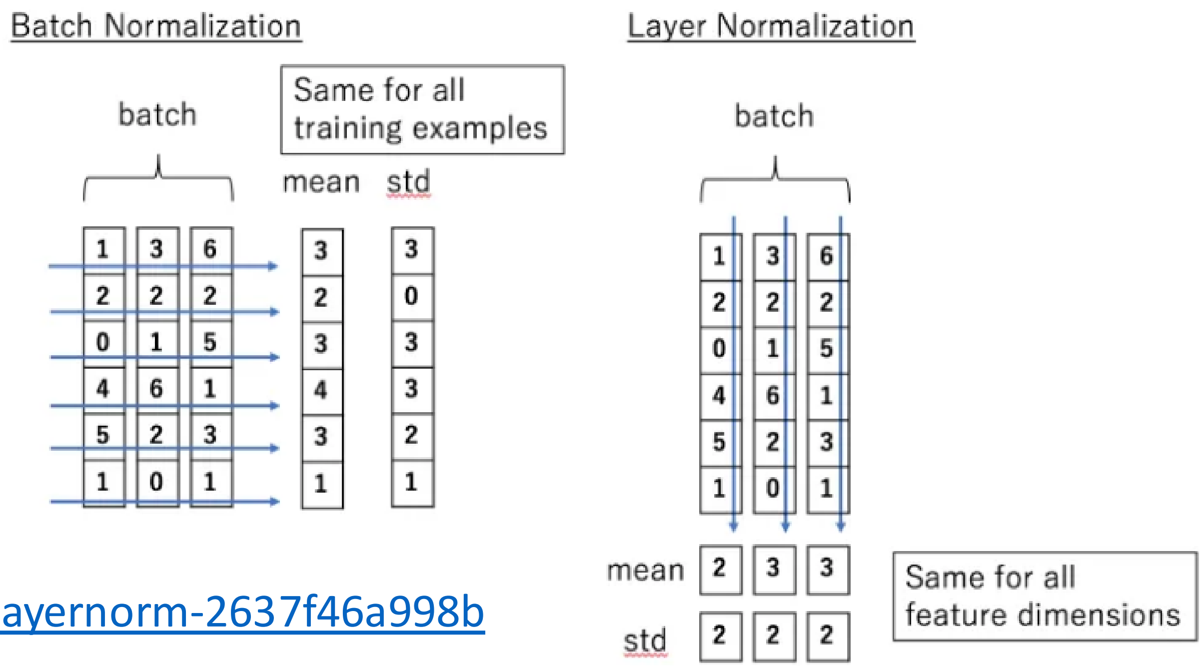
$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0) \quad (4)$$



Difference between Batch Normalization and Layer Normalization

BatchNorm normalizes each feature within a batch of samples, while LayerNorm normalizes all features within each sample.

Let's assume we have a two-dimensional input matrix, where the rows represent the batch and the columns represent the sample features. The target of Batch Normalization is a batch of samples, and the target of Layer Normalization is a single sample, Figure 1 illustrates this concept:



https://medium.com/@florian_algo/batchnorm-and-layernorm-2637f46a998b

https://arxiv.org/abs/2010.11929

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \ldots$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell\ldots}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell \ldots$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0)$$

LayerNorm

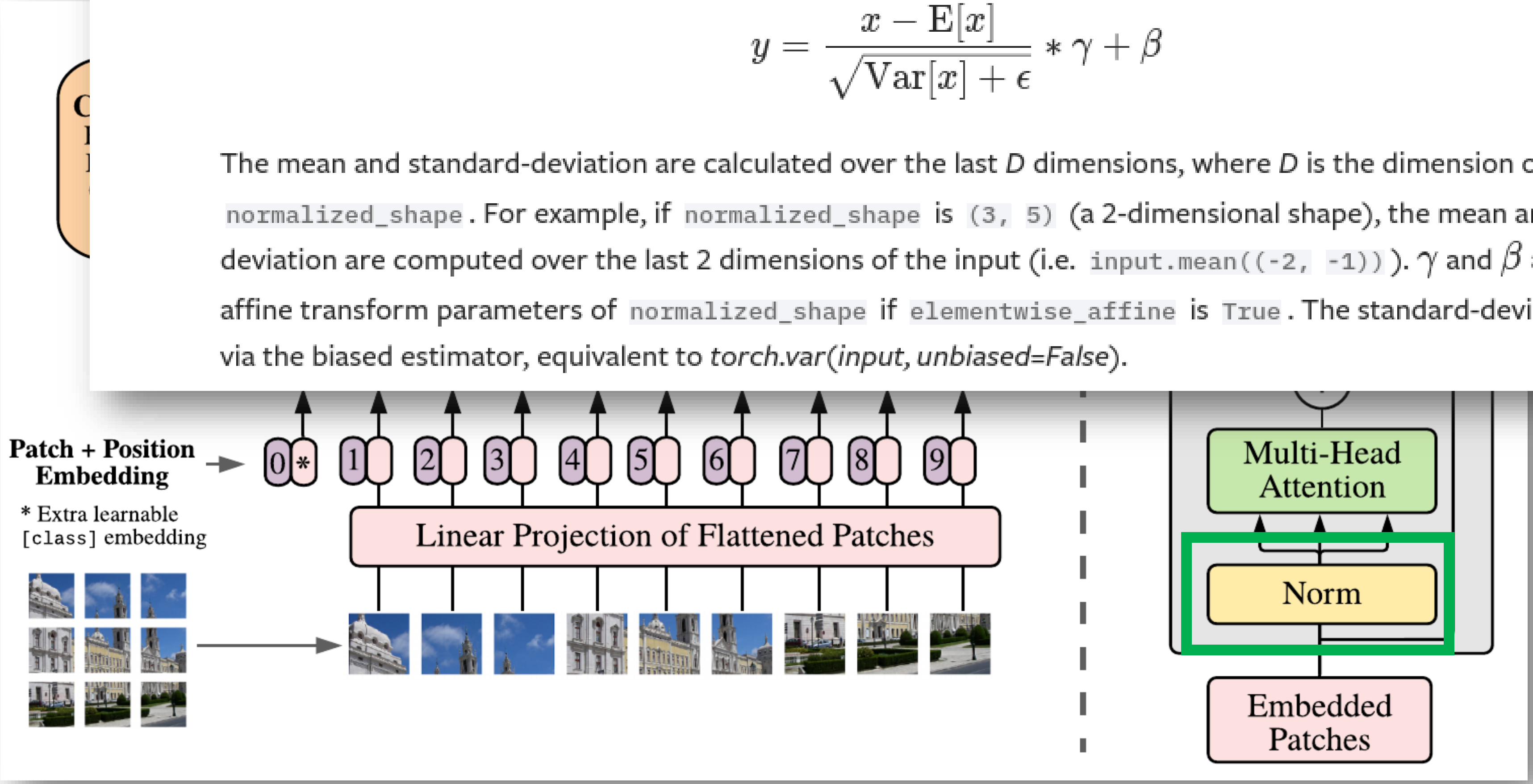
CLASS torch.nn.LayerNorm(*normalized_shape, eps=1e-05, elementwise_affine=True, bias=True, device=None, dtype=None*) [SOURCE]

Applies Layer Normalization over a mini-batch of inputs.

This layer implements the operation as described in the paper Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated over the last D dimensions, where D is the dimension of `normalized_shape`. For example, if `normalized_shape` is `(3, 5)` (a 2-dimensional shape), the mean and standard-deviation are computed over the last 2 dimensions of the input (i.e. `input.mean((-2, -1))`). γ and β are learnable affine transform parameters of `normalized_shape` if `elementwise_affine` is `True`. The standard-deviation is calculated via the biased estimator, equivalent to *torch.var(input, unbiased=False)*.



https://arxiv.org/abs/2010.11929
https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html

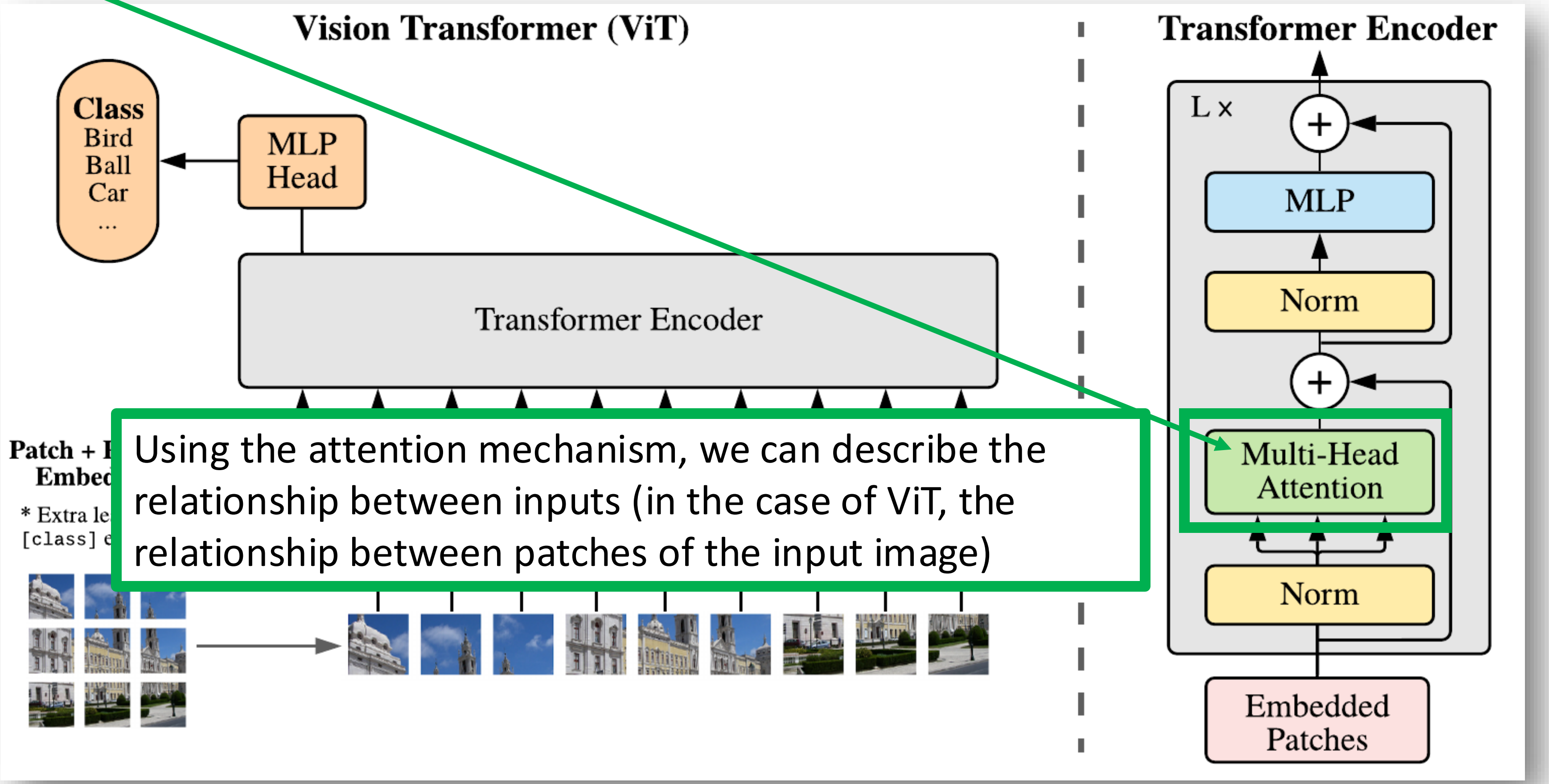
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C)\times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \tag{1}$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1 \ldots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \tag{4}$$



Using the attention mechanism, we can describe the relationship between inputs (in the case of ViT, the relationship between patches of the input image)

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Self attention was proposed in https://arxiv.org/pdf/1706.03762

Following text was taken from https://arxiv.org/pdf/2012.12556

$$\ldots \in \mathbb{R}^{(P^2\cdot C)\times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

$$\ldots = 1 \ldots L \quad (2)$$

2.1 Self-Attention

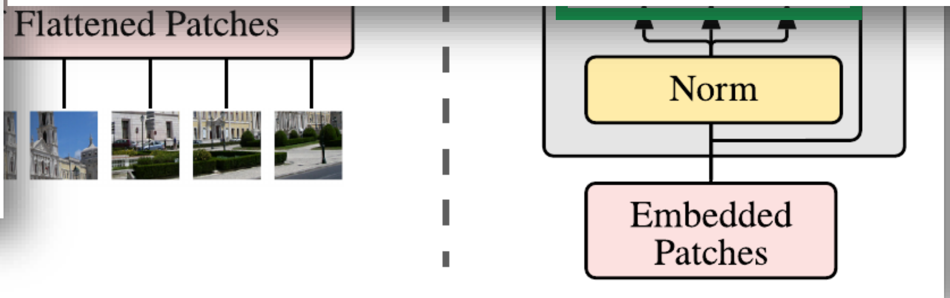
In the self-attention layer, the input vector is first transformed into three different vectors: the query vector $\mathbf{q}$, the key vector $\mathbf{k}$ and the value vector $\mathbf{v}$ with dimension $d_q = d_k = d_v = d_{model} = 512$. Vectors derived from different inputs are then packed together into three different matrices, namely, $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$. Subsequently, the attention function between different input vectors is calculated as follows (and shown in Figure 3 left):

- **Step 1:** Compute scores between different input vectors with $\mathbf{S} = \mathbf{Q} \cdot \mathbf{K}^\top$;
- **Step 2:** Normalize the scores for the stability of gradient with $\mathbf{S}_n = \mathbf{S}/\sqrt{d_k}$;
- **Step 3:** Translate the scores into probabilities with softmax function $\mathbf{P} = \mathrm{softmax}(\mathbf{S}_n)$;
- **Step 4:** Obtain the weighted value matrix with $\mathbf{Z} = \mathbf{V} \cdot \mathbf{P}$.

The process can be unified into a single function:

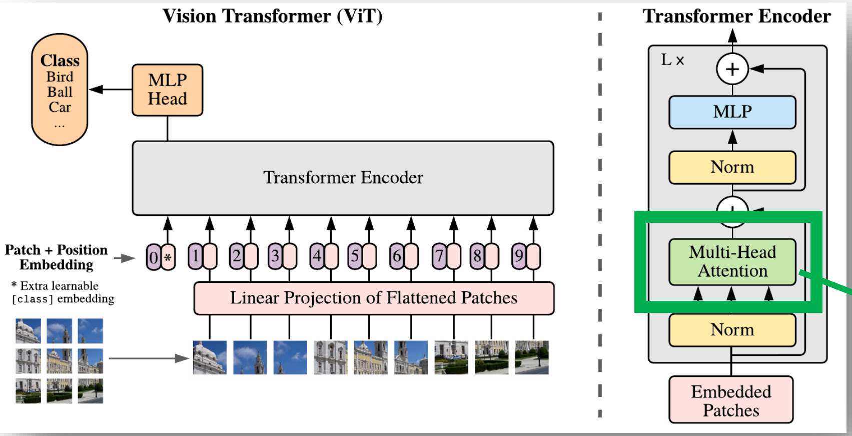
$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\cdot\mathbf{K}^\top}{\sqrt{d_k}}\right)\cdot \mathbf{V}. \quad (1)$$

The logic behind Eq. 1 is simple. Step 1 computes scores between each pair of different vectors, and these scores determine the degree of attention that we give other words when encoding the word at the current position. Step 2 normalizes the scores to enhance gradient stability for improved training, and step 3 translates the scores into probabilities. Finally, each value vector is multiplied by the sum of the probabilities. Vectors with larger probabilities receive additional focus from the following layers.



https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)



$$SA(Q,K,V) = SoftMax(\frac{QK^T}{\sqrt{d_k}}) * V \quad (1)$$

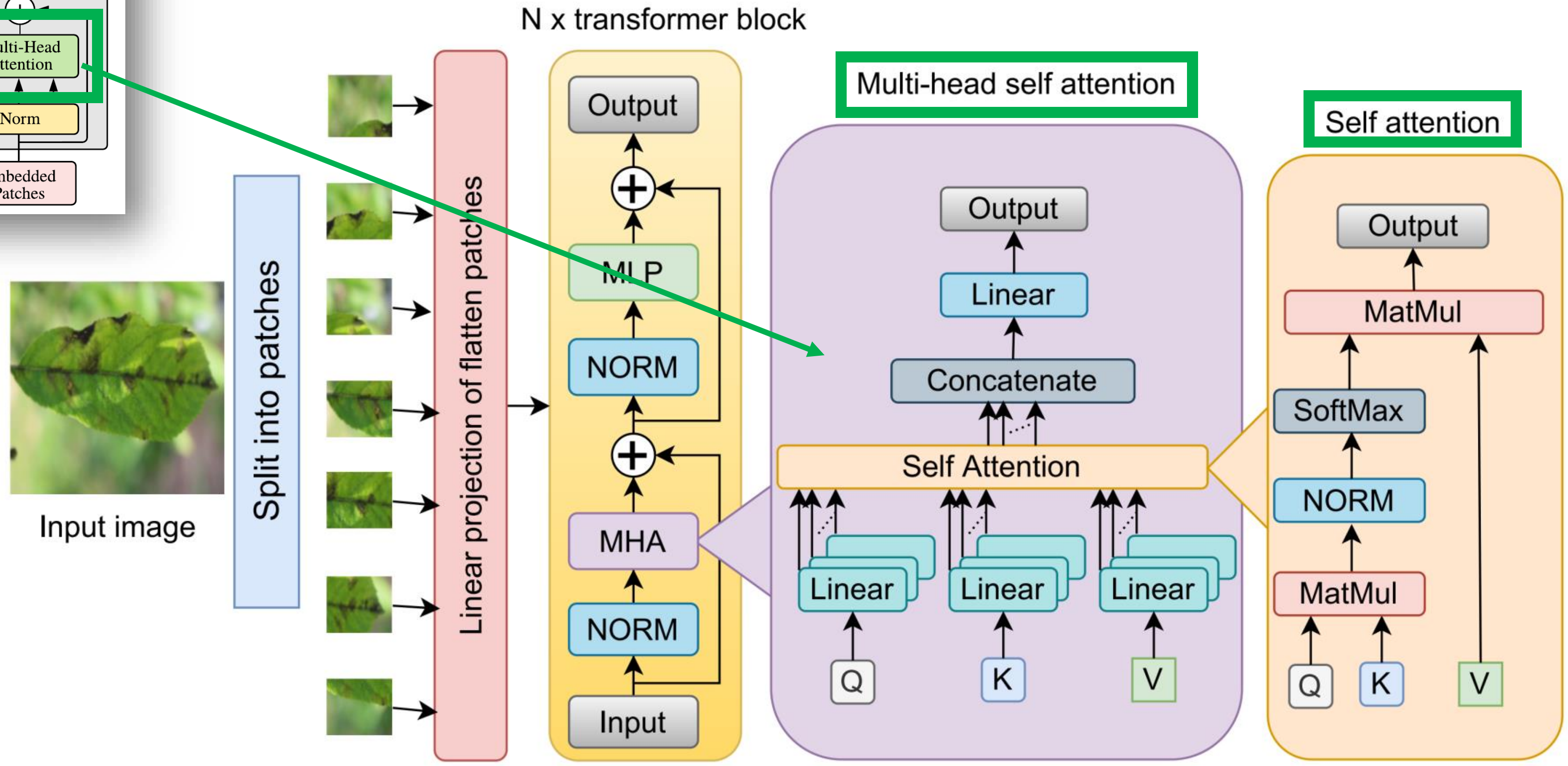


Figure 1: ViT block with multi-head self-attention block and self-attention

https://arxiv.org/abs/2010.11929

https://pytorch.org/vision/main/models/vision_transformer.html

Thakur, Poornima & Khanna, Pritee & Sheorey, Tanuja & Ojha, Aparajita. (2022). Explainable vision transformer enabled convolutional neural network for plant disease identification: PlantXViT. 10.48550/arXiv.2207.07919.

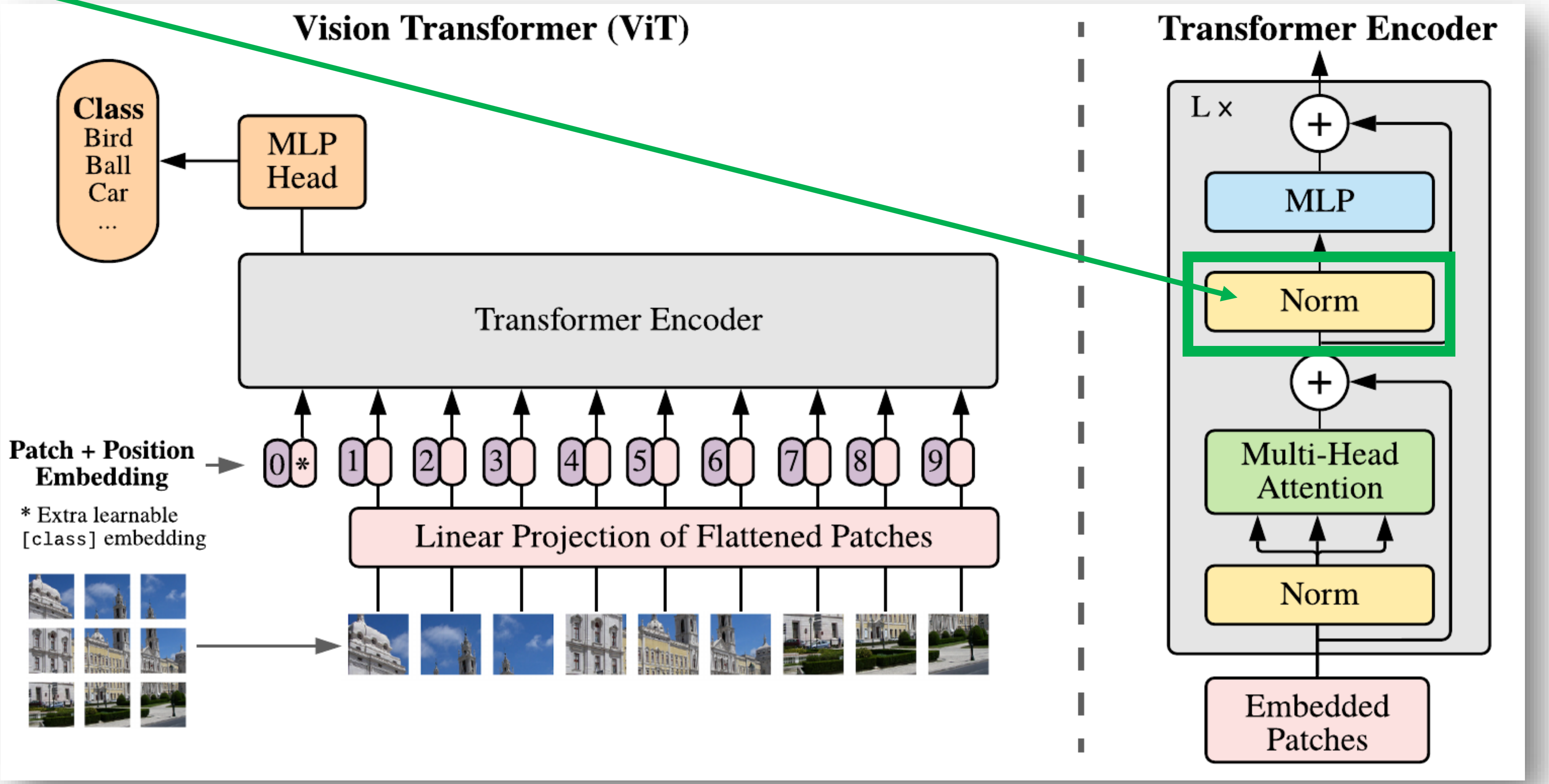
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \tag{1}$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1\ldots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1\ldots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \tag{4}$$



https://arxiv.org/abs/2010.11929

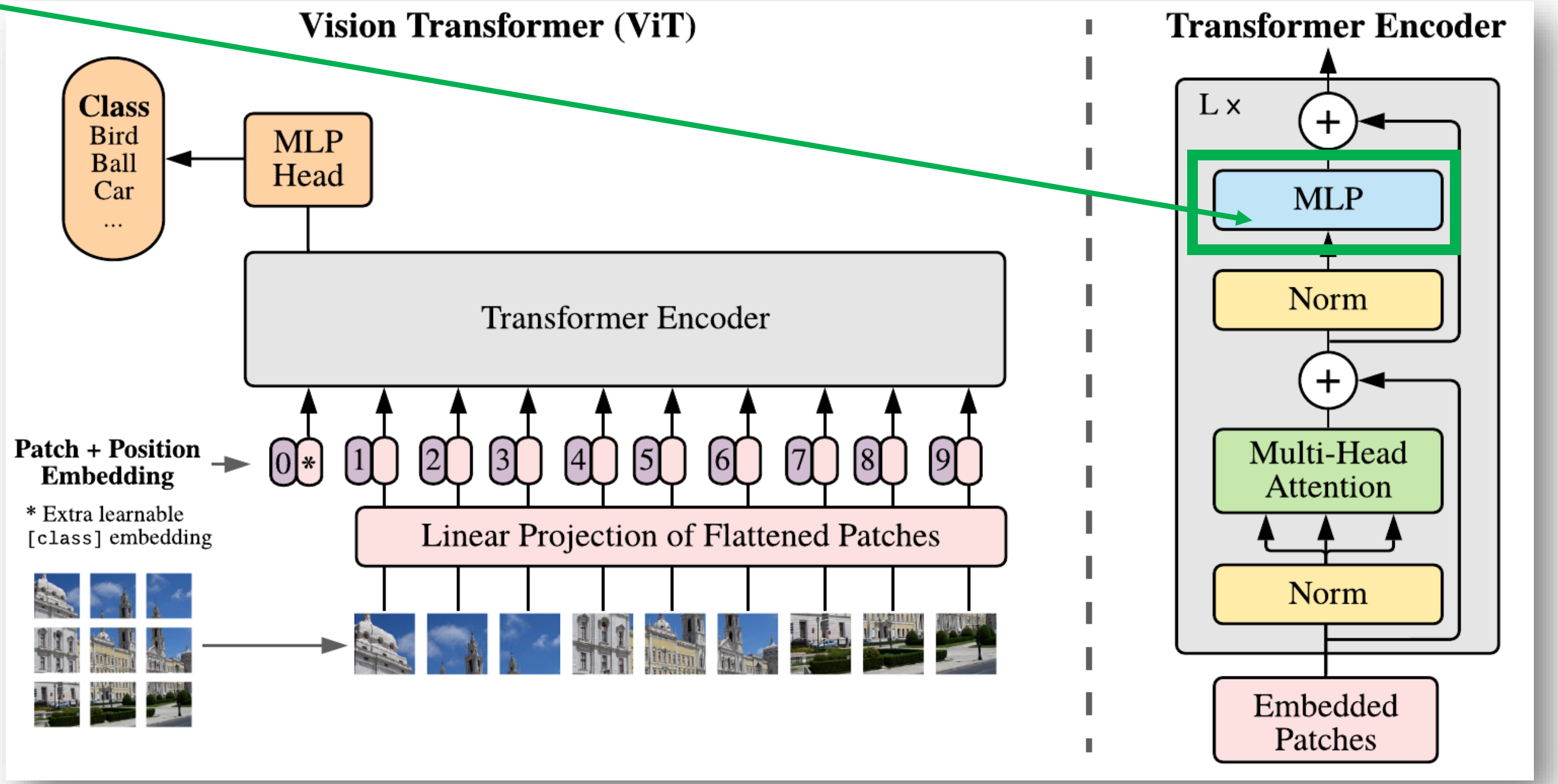
ViT (Vision Transformer)

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \quad (2)$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \quad (3)$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \quad (4)$$



https://arxiv.org/abs/2010.11929

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \ldots$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell} \ldots$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell \ldots$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0)$$

MLP

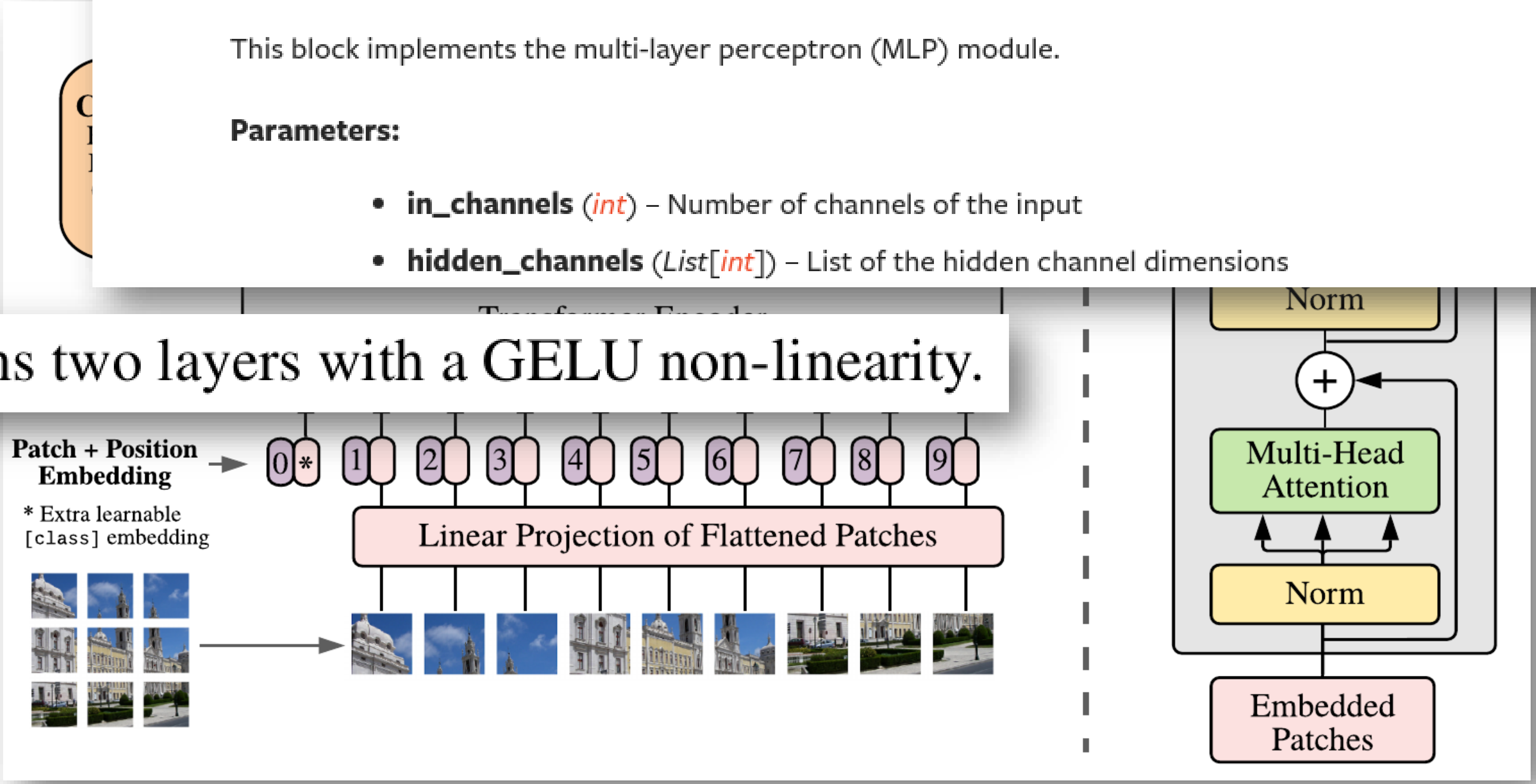
CLASS torchvision.ops.MLP(*in_channels: int, hidden_channels: ~typing.List[int], norm_layer: ~typing.Optional[~typing.Callable[[...], ~torch.nn.modules.module.Module]] = None, activation_layer: ~typing.Optional[~typing.Callable[[...], ~torch.nn.modules.module.Module]] = <class 'torch.nn.modules.activation.ReLU'>, inplace: ~typing.Optional[bool] = None, bias: bool = True, dropout: float = 0.0*) [SOURCE]

This block implements the multi-layer perceptron (MLP) module.

Parameters:

- **in_channels** (*int*) – Number of channels of the input
- **hidden_channels** (*List*[*int*]) – List of the hidden channel dimensions

The MLP contains two layers with a GELU non-linearity.



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/generated/torchvision.ops.MLP.html

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \ldots$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell} \ldots$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell \ldots$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0)$$

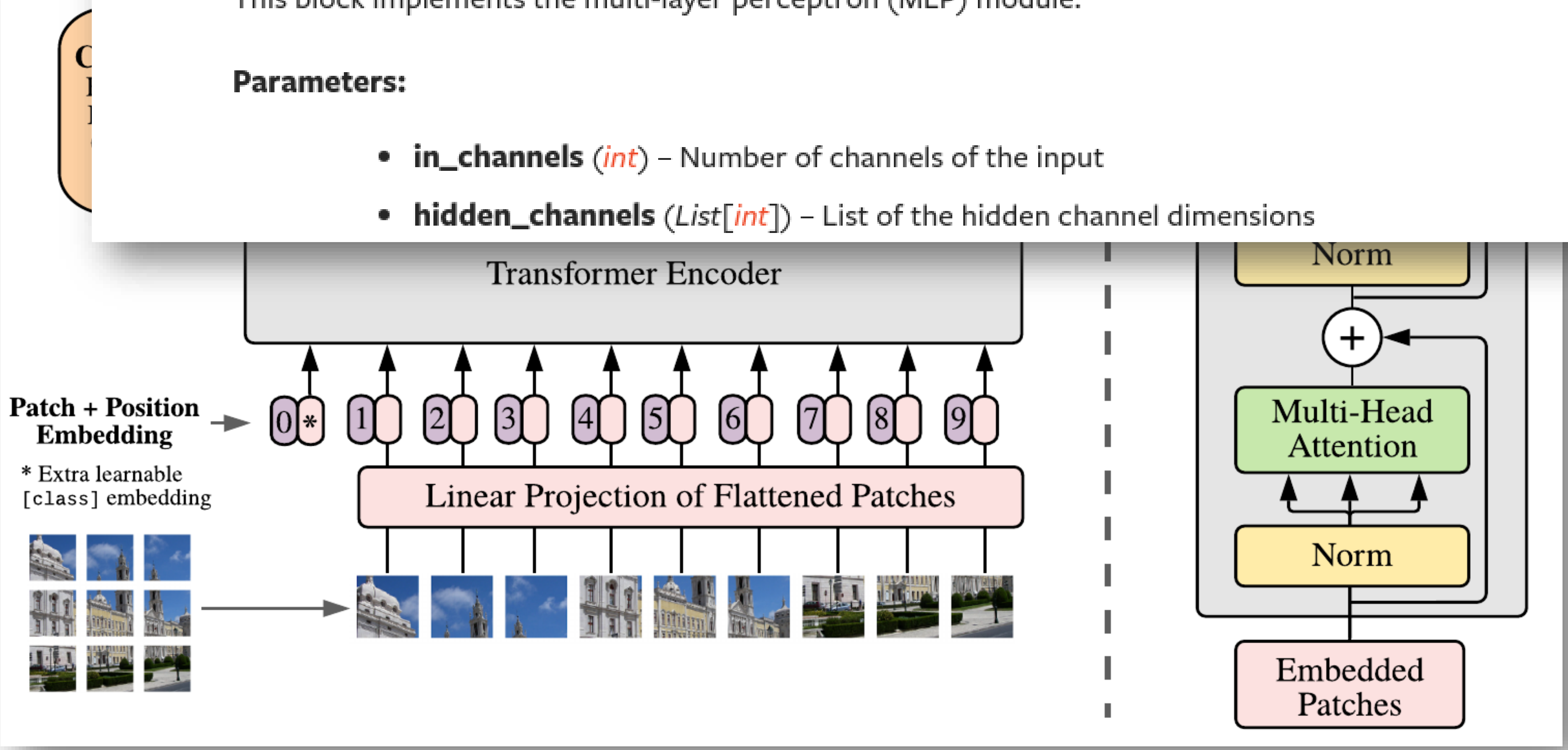
MLP

CLASS torchvision.ops.MLP(*in_channels: int, hidden_channels: ~typing.List[int], norm_layer: ~typing.Optional[~typing.Callable[[...], ~torch.nn.modules.module.Module]] = None, activation_layer: ~typing.Optional[~typing.Callable[[...], ~torch.nn.modules.module.Module]] = <class 'torch.nn.modules.activation.ReLU'>, inplace: ~typing.Optional[bool] = None, bias: bool = True, dropout: float = 0.0*) [SOURCE]

This block implements the multi-layer perceptron (MLP) module.

Parameters:

- **in_channels** (*int*) – Number of channels of the input
- **hidden_channels** (*List[int]*) – List of the hidden channel dimensions



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/generated/torchvision.ops.MLP.html

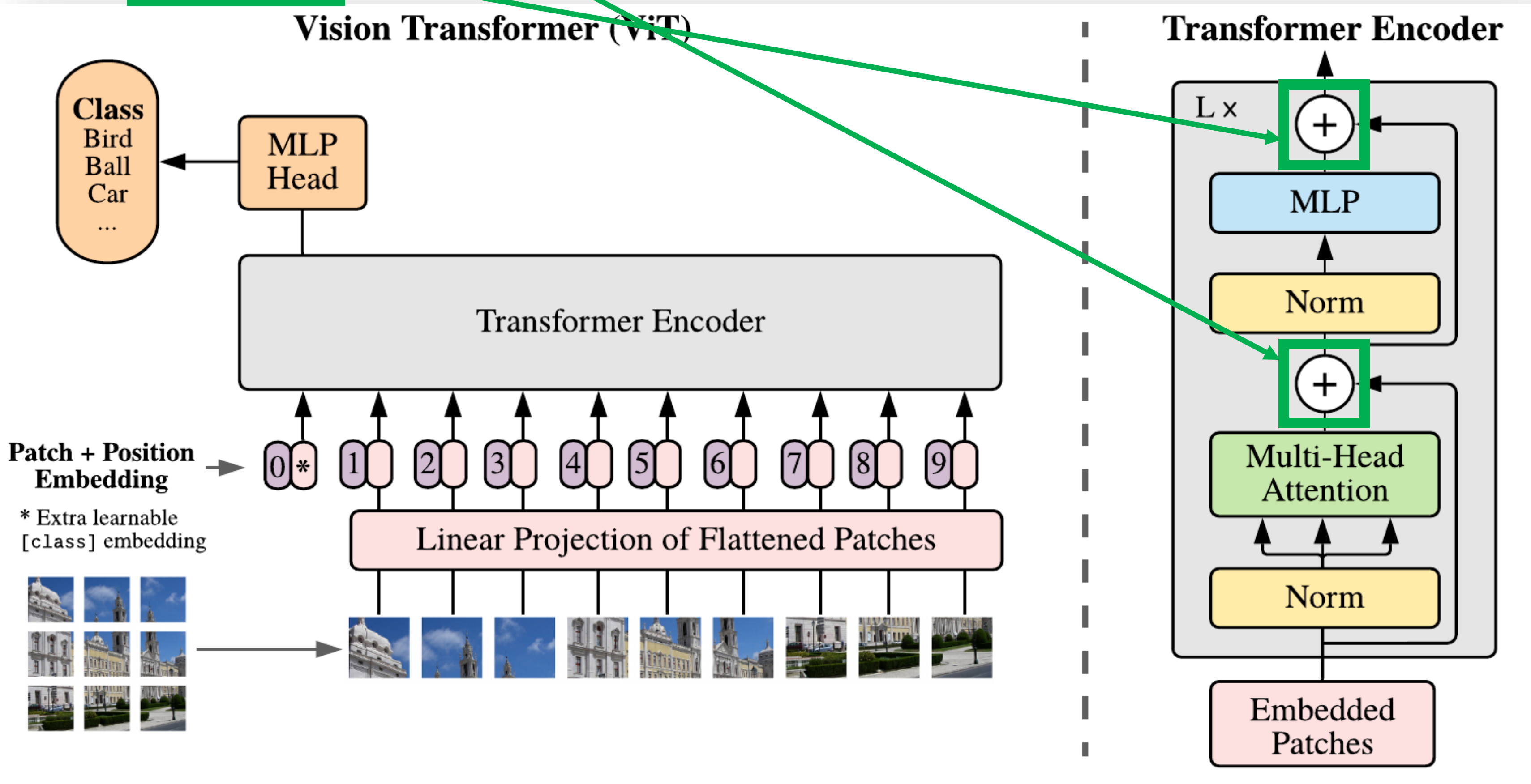
The Transformer encoder (Vaswani et al., 2017) consists of alternating layers of multiheaded self-attention (MSA, see Appendix A) and MLP blocks (Eq. 2, 3). Layernorm (LN) is applied before every block, and residual connections after every block (Wang et al., 2019; Baevski & Auli, 2019).

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E}\in\mathbb{R}^{(P^2\cdot C)\times D},\ \mathbf{E}_{pos}\in\mathbb{R}^{(N+1)\times D} \tag{1}$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1\ldots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1\ldots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \tag{4}$$



https://arxiv.org/abs/2010.11929

The Transformer encoder (Vaswani et al., 2017) consists of alternating layers of multiheaded self-attention (MSA, see Appendix A) and MLP blocks (Eq. 2, 3). Layernorm (LN) is applied before every block, and residual connections after every block (Wang et al., 2019; Baevski & Auli, 2019).

$$, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

(2)

(3)

(4)

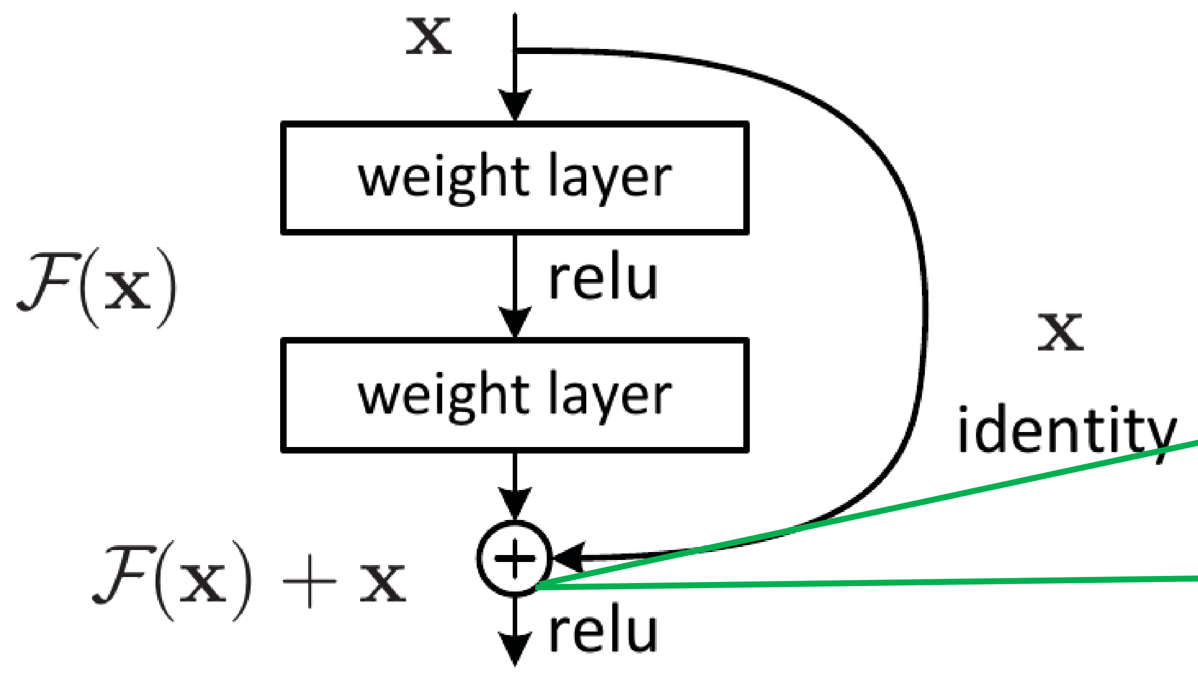
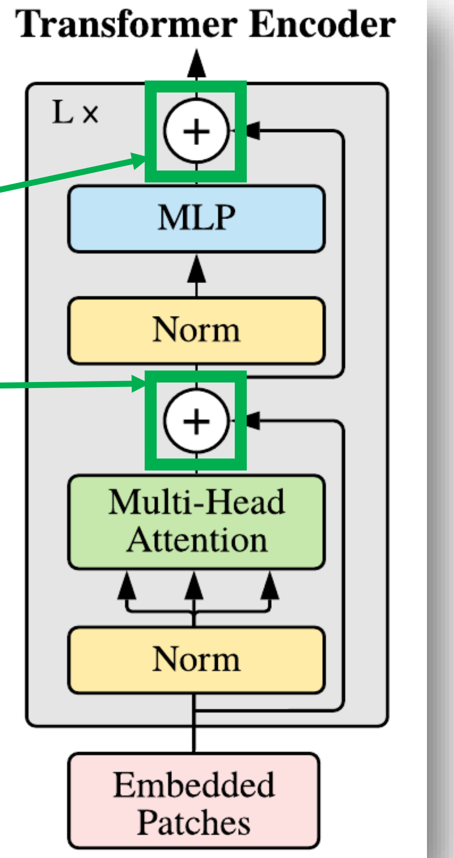


Figure 2. Residual learning: a building block.



Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

https://arxiv.org/abs/2010.11929
https://arxiv.org/abs/1512.03385

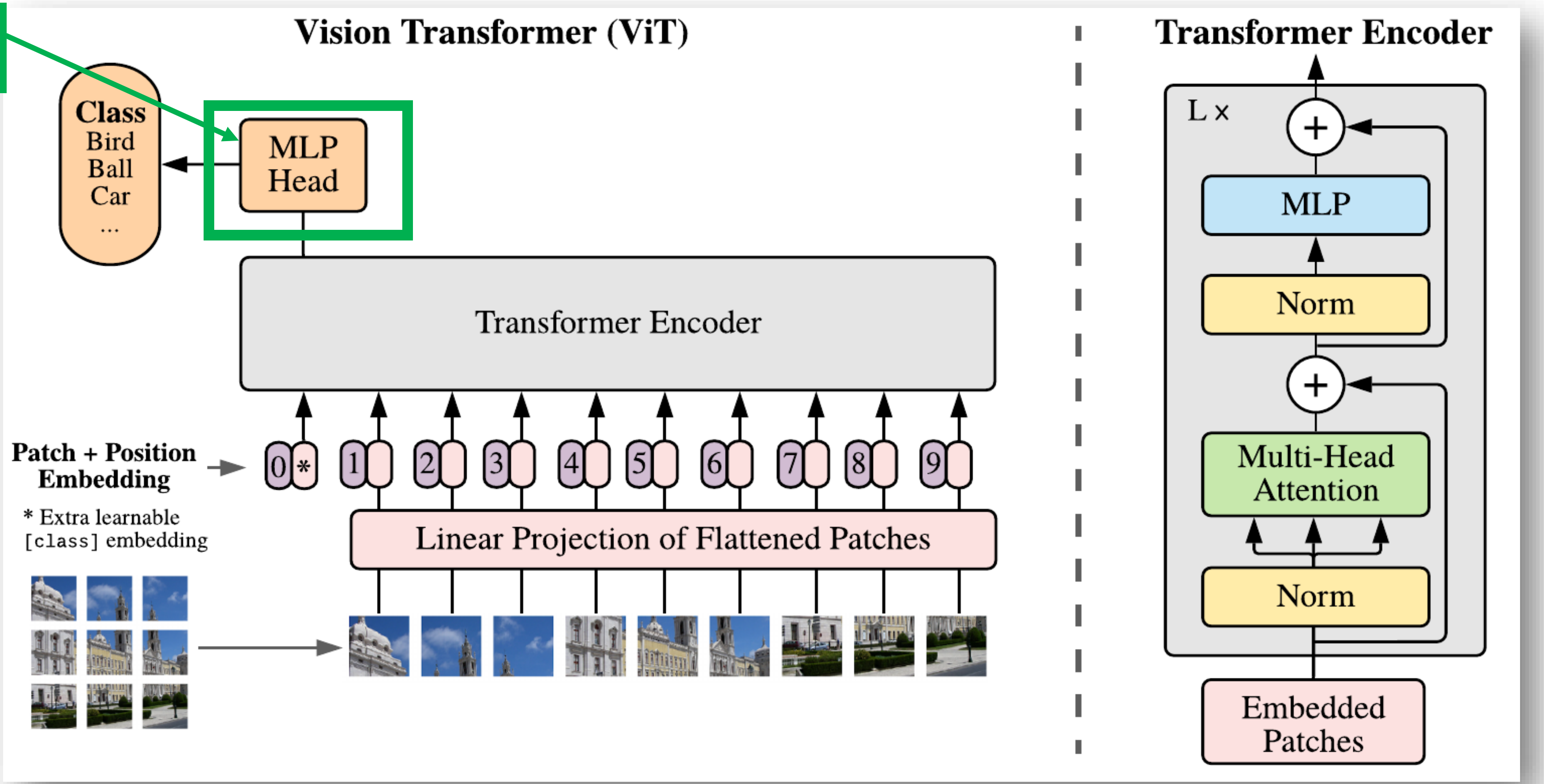
The Transformer encoder (Vaswani et al., 2017) consists of alternating layers of multiheaded self-attention (MSA, see Appendix A) and MLP blocks (Eq. 2, 3). Layernorm (LN) is applied before every block, and residual connections after every block (Wang et al., 2019; Baevski & Auli, 2019).

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \tag{1}$$

$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1\ldots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1\ldots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \tag{4}$$



https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

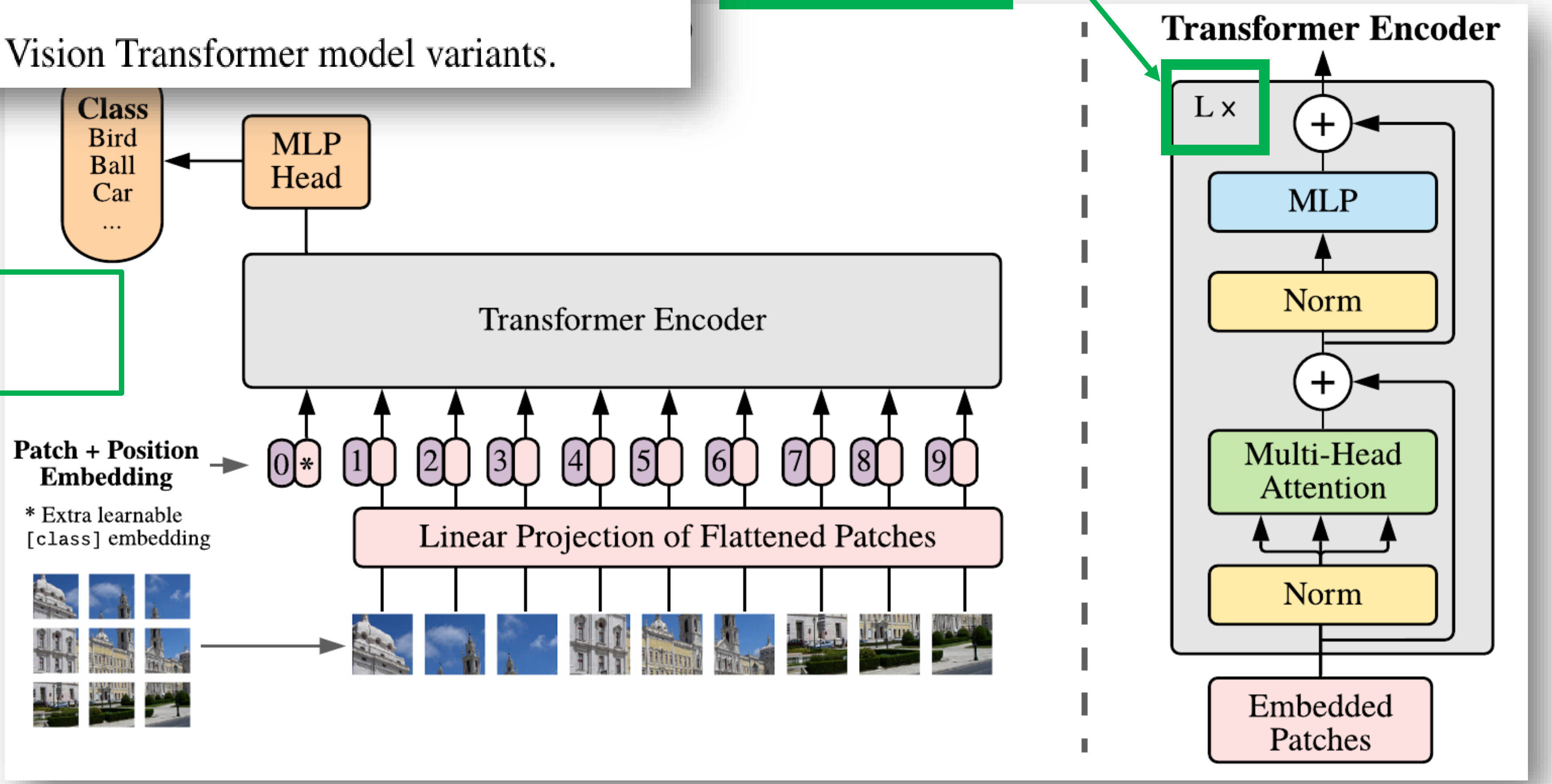
Table 1: Details of Vision Transformer model variants.

$$\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \quad (1)$$

$$\ell = 1 \ldots L \quad (2)$$

$$\ell = 1 \ldots L \quad (3)$$

(4)



number of Transformer Encoder blocks

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

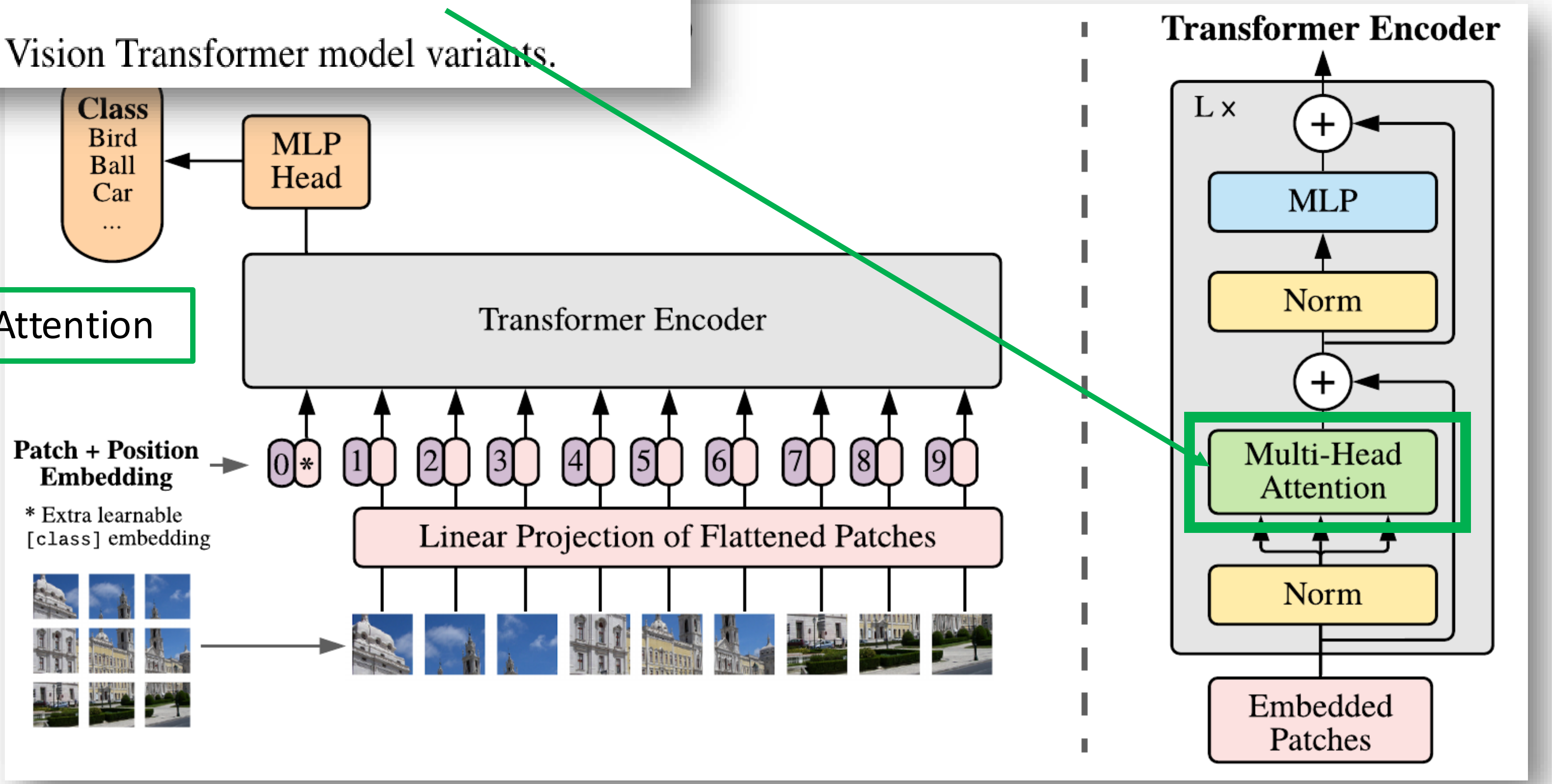
Table 1: Details of Vision Transformer model variants.

$$\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \quad (1)$$

$$\ell = 1 \ldots L \quad (2)$$

$$\ell = 1 \ldots L \quad (3)$$

(4)



Number of Multi-Head Attention

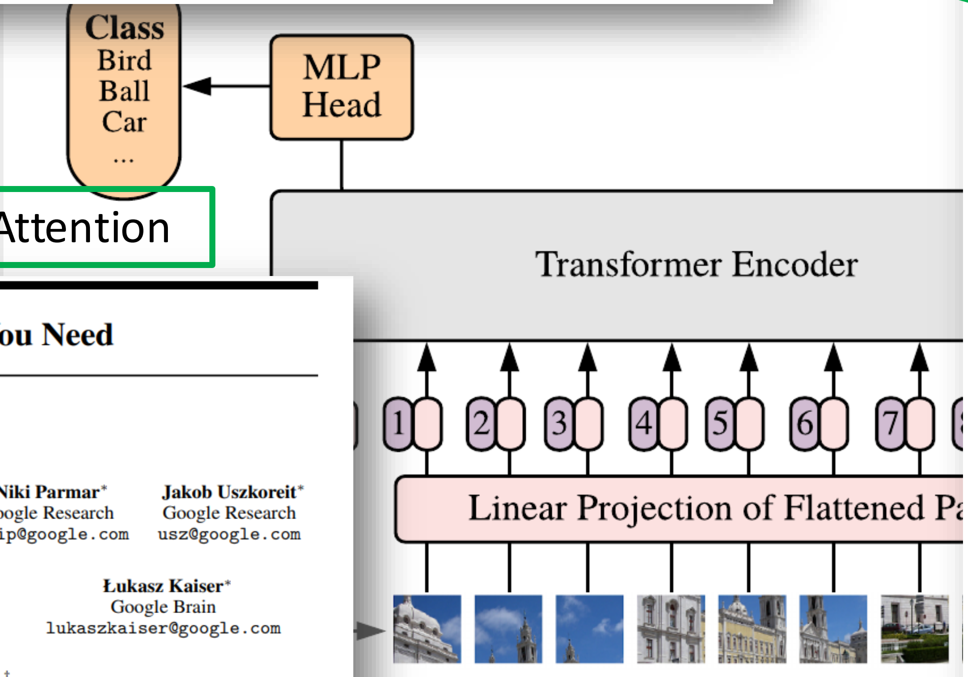
https://arxiv.org/abs/2010.11929

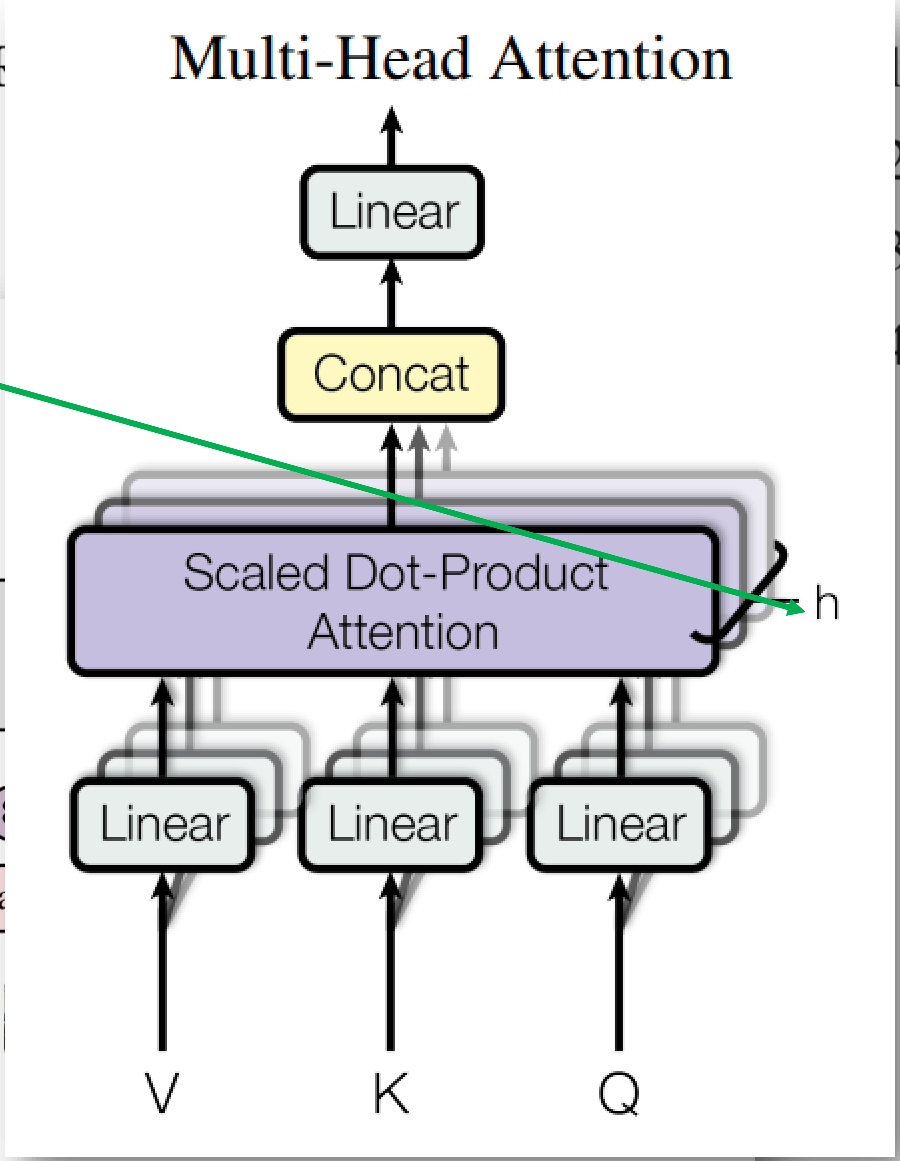
ViT (Vision Transformer)

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

Number of Multi-Head Attention





Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

https://arxiv.org/pdf/1706.03762

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

A Multihead Self-Attention

Standard **qkv** self-attention (SA, Vaswani et al. (2017)) is a popular building block for neural architectures. For each element in an input sequence $\mathbf{z} \in \mathbb{R}^{N \times D}$, we compute a weighted sum over all values $\mathbf{v}$ in the sequence. The attention weights A_{ij} are based on the pairwise similarity between two elements of the sequence and their respective query $\mathbf{q}^i$ and key $\mathbf{k}^j$ representations.

$$[\mathbf{q}, \mathbf{k}, \mathbf{v}] = \mathbf{z}\mathbf{U}_{qkv} \qquad \mathbf{U}_{qkv} \in \mathbb{R}^{D \times 3D_h}, \tag{5}$$

$$A = \mathrm{softmax}\left(\mathbf{q}\mathbf{k}^\top / \sqrt{D_h}\right) \qquad A \in \mathbb{R}^{N \times N}, \tag{6}$$

$$\mathrm{SA}(\mathbf{z}) = A\mathbf{v}\,. \tag{7}$$

Multihead self-attention (MSA) is an extension of SA in which we run k self-attention operations, called "heads", in parallel, and project their concatenated outputs. To keep compute and number of parameters constant when changing k, D_h (Eq. 5) is typically set to D/k.

$$\mathrm{MSA}(\mathbf{z}) = [\mathrm{SA}_1(z); \mathrm{SA}_2(z); \cdots; \mathrm{SA}_k(z)]\,\mathbf{U}_{msa} \qquad \mathbf{U}_{msa} \in \mathbb{R}^{k \cdot D_h \times D} \tag{8}$$

illia.polosukhin@gmail.com

V K Q

https://arxiv.org/pdf/1706.03762

https://arxiv.org/abs/2010.11929

ViT (Vision Transformer)

VisionTransformer

The VisionTransformer model is based on the An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale paper.

Model builders

The following model builders can be used to instantiate a VisionTransformer model, with or without pre-trained weights. All the model builders internally rely on the `torchvision.models.vision_transformer.VisionTransformer` base class. Please refer to the source code for more details about this class.

`vit_b_16`(*[, weights, progress])	Constructs a vit_b_16 architecture from An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
`vit_b_32`(*[, weights, progress])	Constructs a vit_b_32 architecture from An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
`vit_l_16`(*[, weights, progress])	Constructs a vit_l_16 architecture from An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
`vit_l_32`(*[, weights, progress])	Constructs a vit_l_32 architecture from An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
`vit_h_14`(*[, weights, progress])	Constructs a vit_h_14 architecture from An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.

Published as a conference paper at ICLR 2021

		ViT-B/16	ViT-B/32	ViT-L/16	ViT-L/32	ViT-H/14
ImageNet	CIFAR-10	98.13	97.77	97.86	97.94	-
	CIFAR-100	87.13	86.31	86.35	87.07	-
	ImageNet	77.91	73.38	76.53	71.16	-
	ImageNet ReaL	83.57	79.56	82.19	77.83	-
	Oxford Flowers-102	89.49	85.43	89.66	86.36	-
	Oxford-IIIT-Pets	93.81	92.04	93.64	91.35	-
ImageNet-21k	CIFAR-10	98.95	98.79	99.16	99.13	99.27
	CIFAR-100	91.67	91.97	93.44	93.04	93.82
	ImageNet	83.97	81.28	85.15	80.99	85.13
	ImageNet ReaL	88.35	86.63	88.40	85.65	88.70
	Oxford Flowers-102	99.38	99.11	99.61	99.19	99.51
	Oxford-IIIT-Pets	94.43	93.02	94.73	93.09	94.82
JFT-300M	CIFAR-10	99.00	98.61	99.38	99.19	99.50
	CIFAR-100	91.87	90.49	94.04	92.52	94.55
	ImageNet	84.15	80.73	87.12	84.37	88.04
	ImageNet ReaL	88.85	86.27	89.99	88.28	90.33
	Oxford Flowers-102	99.56	99.27	99.56	99.45	99.68
	Oxford-IIIT-Pets	95.80	93.40	97.11	95.83	97.56

Table 5: Top1 accuracy (in %) of Vision Transformer on various datasets when pre-trained on ImageNet, ImageNet-21k or JFT300M. These values correspond to Figure 3 in the main text. Models are fine-tuned at 384 resolution. Note that the ImageNet results are computed without additional techniques (Polyak averaging and 512 resolution images) used to achieve results in Table 2.

https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html

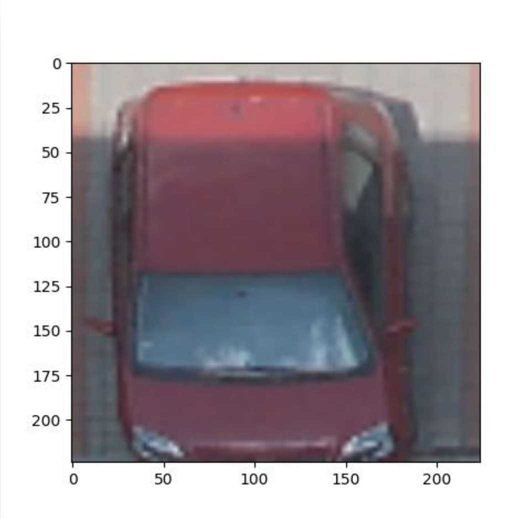
ViT (Vision Transformer)

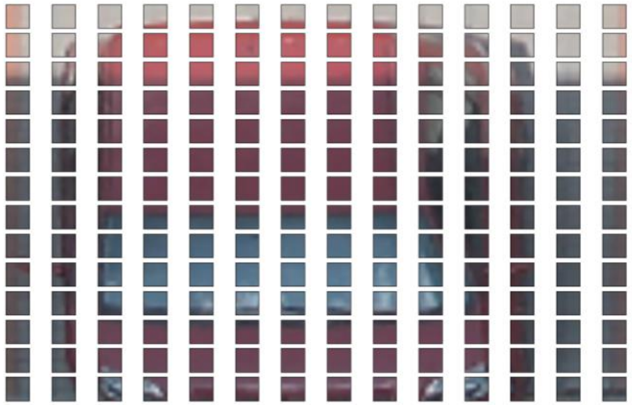
Part 3. Selected Implementation Details

https://arxiv.org/abs/2010.11929
https://arxiv.org/abs/1512.03385

ViT – Selected Implementation Details – Equation 1.

We can use the **nn.Conv2d** layer to split the input image into patches (blocks) and create the embedding vector (with size = 768, in this example – ViT Base model) for each patch.





```
h = 224
w = 224
c = 3
ps = 16

conv = nn.Conv2d(
in_channels  = 3,
out_channels = 768,
kernel_size  = (ps,ps),
stride       = ps,
padding      = 0,
bias         = False)
```

shape of input 224 x 244 x 3 image after nn.Conv2d:
([768, 14, 14])

https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html

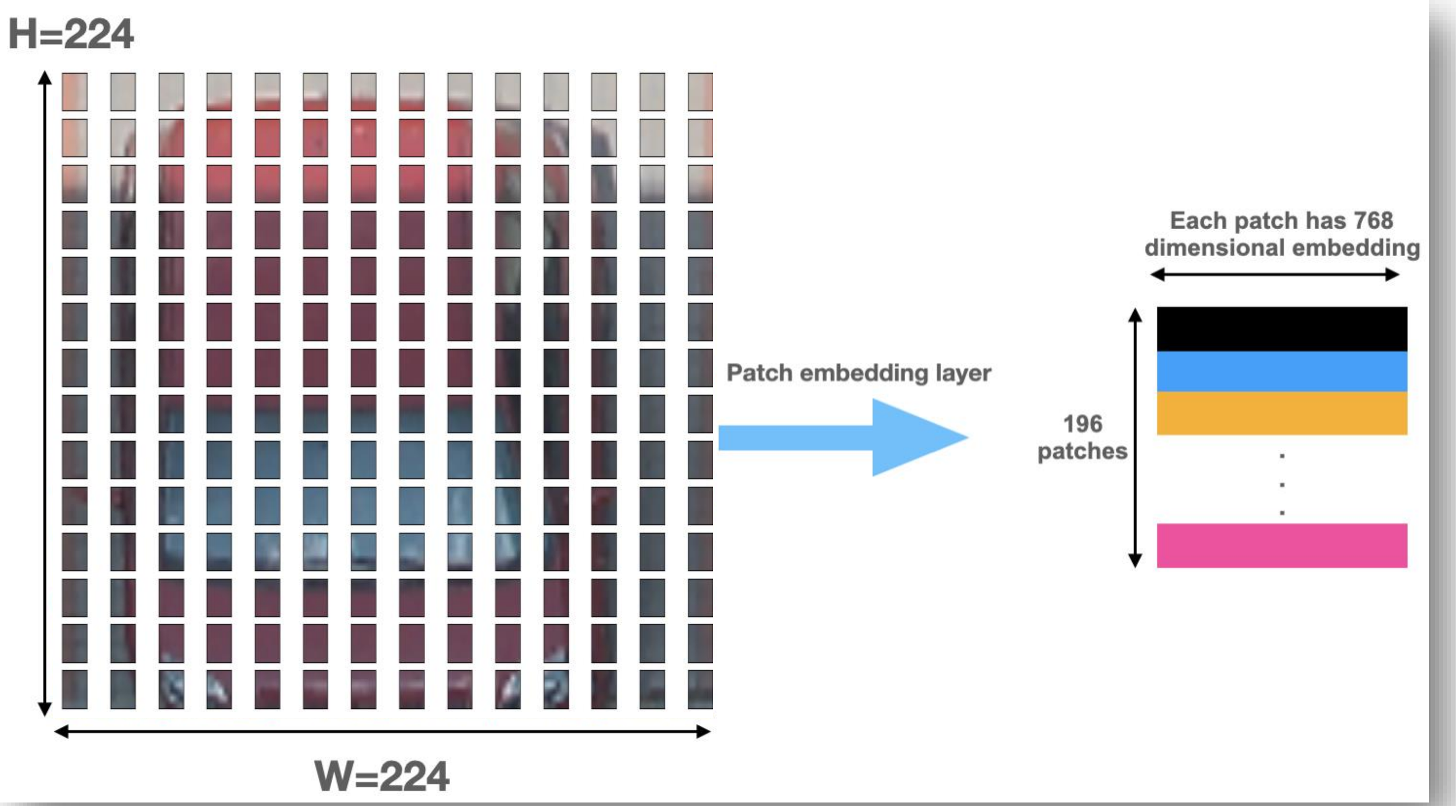
ViT – Selected Implementation Details – Equation 1.

- 768 feature maps of size 14 x 14 are generated
- we can use flatten on these 14 x 14 feature maps to obtain: ([batch_size, 768, 196])
- it is necessary to switch dimensions so that the number of patches is in the second place: ([batch_size, 196, 768])

```
block_permute = nn.Sequential(

        nn.Conv2d(
        in_channels=3,
        out_channels=768,
        kernel_size=(ps,ps),
        stride=ps
        padding = 0,
        bias=False),

        nn.Flatten(start_dim=2, end_dim=-1),
        MyPermute((0, 2, 1))
    )
```



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html
https://learnopencv.com/the-future-of-image-recognition-is-here-pytorch-vision-transformer/

ViT – Selected Implementation Details – Equation 1.

- class_token = nn.Parameter(torch.ones(batch_size, 1, 768))
- z_0_with_class = torch.cat((class_token, block_out_permute), dim=1)
- position_embeddings = nn.Parameter(torch.ones(1, 197, 768))
- z_0_with_class_with_poss = z_0_with_class + position_embeddings

$$\mathbf{z}_0 = [\mathbf{x}_\text{class}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \tag{1}$$

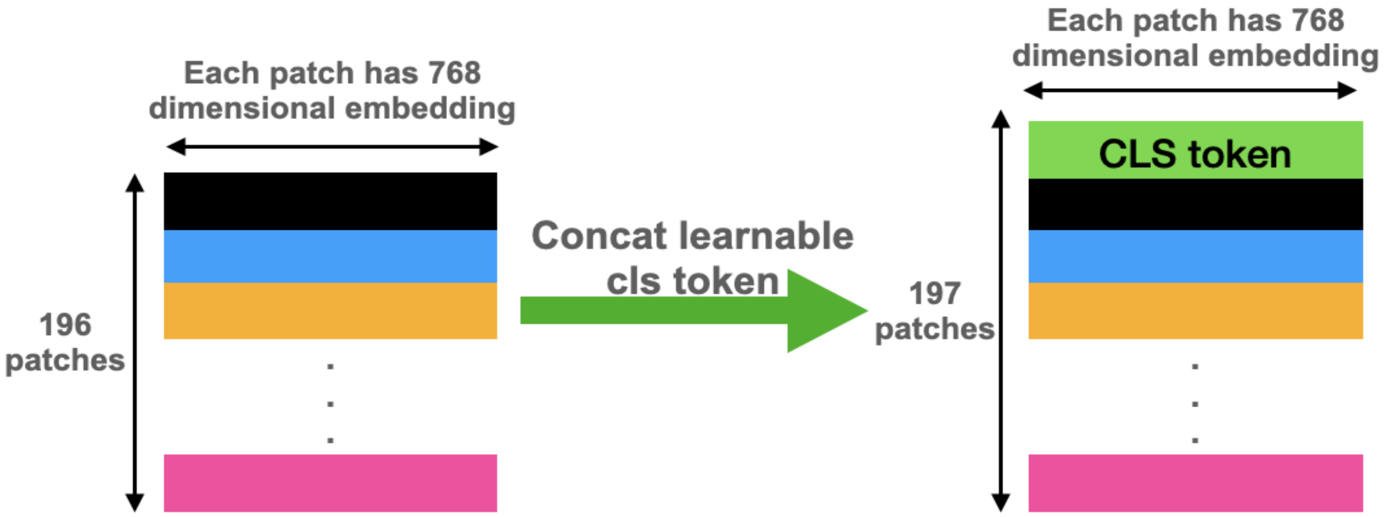
$$\mathbf{z'}_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \dots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \dots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0)$$

```
block_permute = nn.Sequential(
    nn.Conv2d(
    in_channels=3,
    out_channels=768,
    kernel_size=(ps,ps),
    stride=ps,
    padding = 0,
    bias=False),

    nn.Flatten(start_dim=2, end_dim=-1),
    MyPermute((0, 2, 1))
)
```



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html
https://learnopencv.com/the-future-of-image-recognition-is-here-pytorch-vision-transformer/

ViT – Selected Implementation Details – Equation 1.

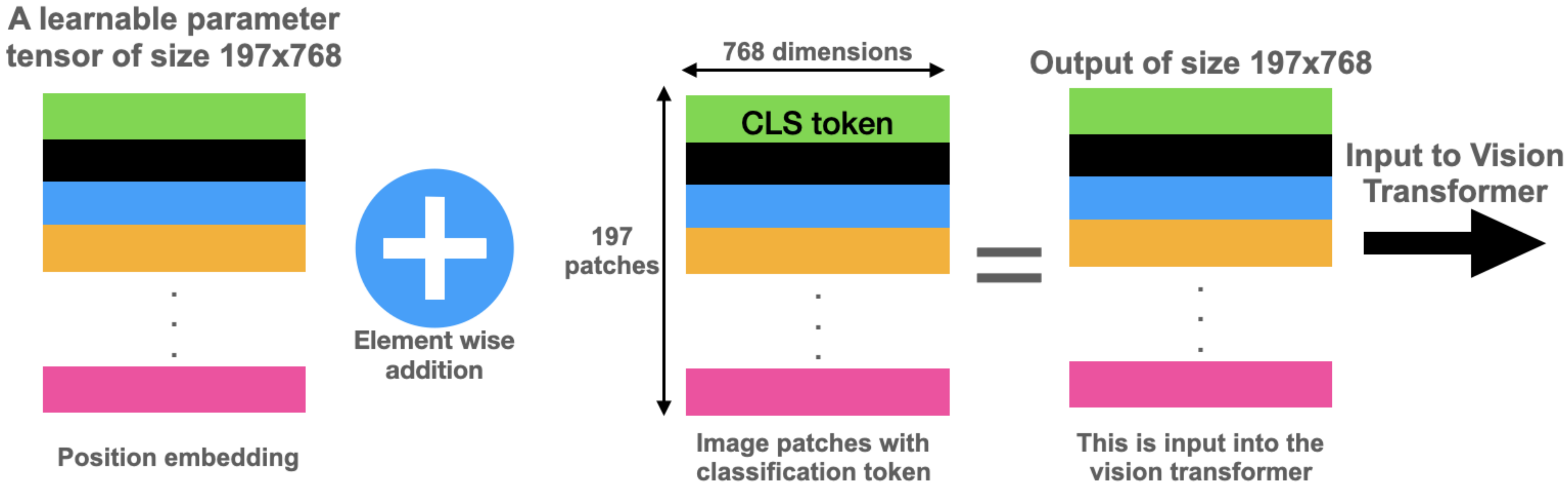
- class_token = nn.Parameter(torch.ones(batch_size, 1, 768))
- z_0_with_class = torch.cat((class_token, block_out_permute), dim=1)
- position_embeddings = nn.Parameter(torch.ones(1, 197, 768))
- z_0_with_class_with_poss = z_0_with_class + position_embeddings

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \tag{1}$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \tag{2}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1 \ldots L \tag{3}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \tag{4}$$



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html
https://learnopencv.com/the-future-of-image-recognition-is-here-pytorch-vision-transformer/

ViT – Selected Implementation Details – Equation 1.

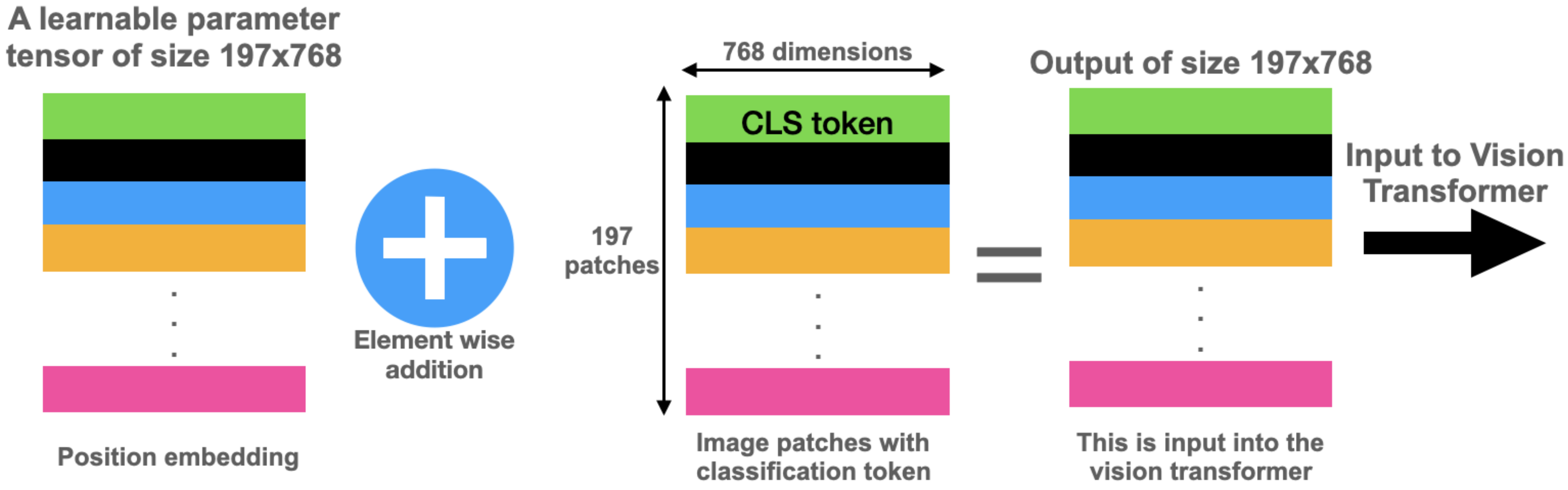
- class_token = nn.Parameter(torch.ones(batch_size, 1, 768))
- z_0_with_class = torch.cat((class_token, block_out_permute), dim=1)
- position_embeddings = nn.Parameter(torch.ones(1, 197, 768))
- z_0_with_class_with_poss = z_0_with_class + position_embeddings

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

$$\mathbf{z'}_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L \quad (2)$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z'}_\ell)) + \mathbf{z'}_\ell, \qquad \ell = 1 \ldots L \quad (3)$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0) \quad (4)$$



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html
https://learnopencv.com/the-future-of-image-recognition-is-here-pytorch-vision-transformer/

...sformer)

MultiheadAttention

CLASS torch.nn.MultiheadAttention(*embed_dim, num_heads, dropout=0.0, bias=True, add_bias_kv=False, add_zero_attn=False, kdim=None, vdim=None, batch_first=False, device=None, dtype=None*) [SOURCE]

Allows the model to jointly attend to information from different representation subspaces.

Method described in the paper: Attention Is All You Need.

Multi-Head Attention is defined as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h)W^O$$

where $head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

`nn.MultiHeadAttention` will use the optimized implementations of `scaled_dot_product_attention()` when possible.

In addition to support for the new `scaled_dot_product_attention()` function, for speeding up Inference, MHA will use fastpath inference with support for Nested Tensors, iff:

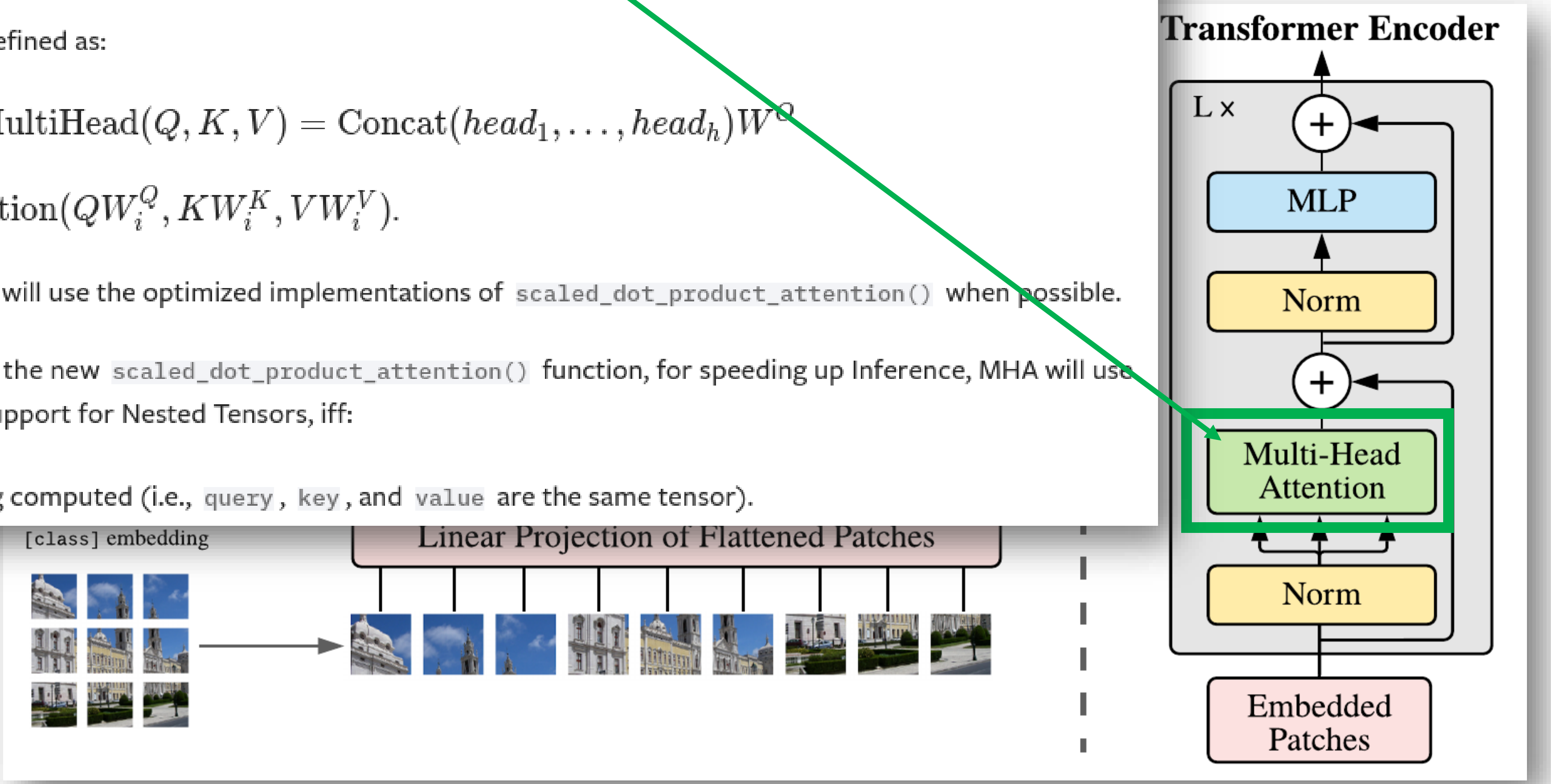
- self attention is being computed (i.e., `query`, `key`, and `value` are the same tensor).

$$\ldots_{pos} \in \mathbb{R}^{(N+1)\times D} \quad (1)$$

(2)

(3)

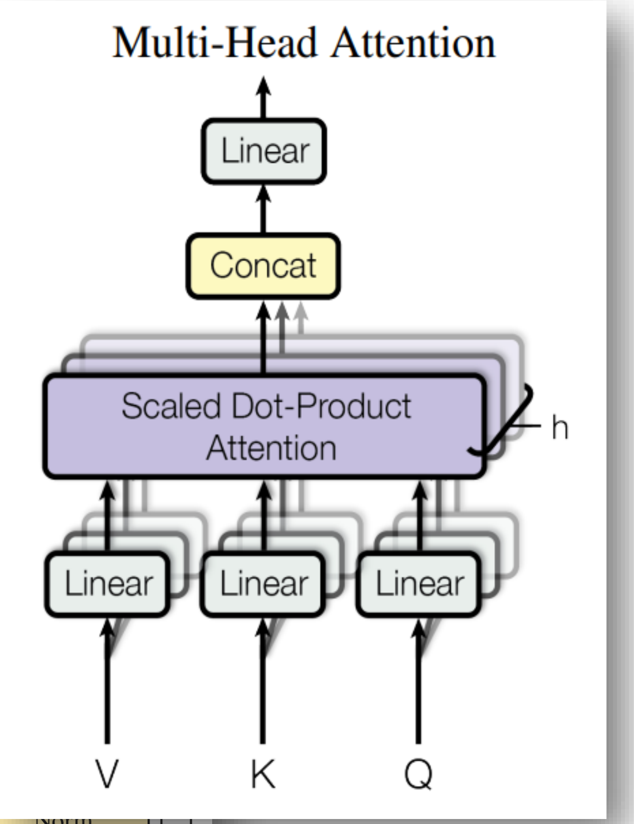
(4)



https://arxiv.org/abs/2010.11929
https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html

ViT – Selected Implementation Details – Equation 2.

```
layer_norm = nn.LayerNorm(768)
layer_norm_out = layer_norm(z_0_with_class_with_poss)
multihead_attn = nn.MultiheadAttention(embed_dim = 768, num_heads = 12, batch_first = True)
attn_output, attn_output_weights = multihead_attn(query = layer_norm_out,
                                                  key = layer_norm_out,
                                                  value = layer_norm_out)
```



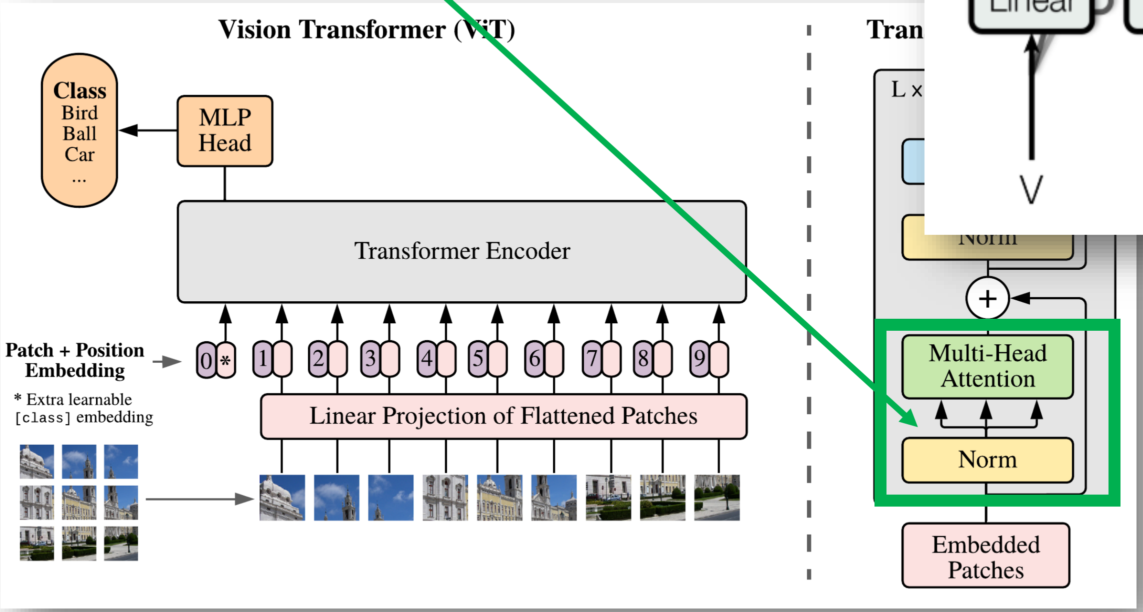
$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\, \mathbf{E}_{pos} \in$$

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1\ldots L$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1\ldots L$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0)$$

You can continue in a similar way with equations 3 and 4 during the exercise (it is appropriate to organise the ViT blocks into individual classes)



https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html

Where to go from here?

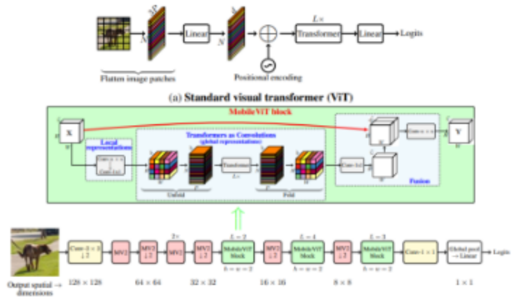
https://github.com/lucidrains/vit-pytorch

README | MIT license

Table of Contents

README | MIT license

MobileViT



(b) MobileViT. Here, Conv-n × n in the MobileViT block represents a standard n × n convolution and MV2 refers to MobileNetv2 block. Blocks that perform down-sampling are marked with ↓ 2.

This paper introduce MobileViT, a light-weight and general purpose vision transformer for mobile devices. MobileViT presents a different perspective for the global processing of information with transformers.

You can use it with the following code (ex. mobilevit_xs)

```
import torch
from vit_pytorch.mobile_vit import MobileViT

mbvit_xs = MobileViT(
    image_size = (256, 256),
    dims = [96, 120, 144],
    channels = [16, 32, 48, 48, 64, 64, 80, 80, 96, 96, 384],
    num_classes = 1000
)

img = torch.randn(1, 3, 256, 256)

pred = mbvit_xs(img) # (1, 1000)
```

https://arxiv.org/abs/2010.11929
https://pytorch.org/vision/main/models/vision_transformer.html