VIT:

- Vision Transformer (ViT) is an innovative deep learning architecture designed to process visual data using the same transformer architecture that revolutionized natural language processing (NLP).
- Unlike convolutional neural networks (CNNs), which rely on convolutions to capture local spatial features, Vision Transformers adopt the self-attention mechanism to model global relationships across image patches.
- This architecture has demonstrated state-of-the-art performance in many computer vision tasks such as image classification, object detection, and segmentation.

Vision Transformer (ViT) Architecture Overview

- The Vision Transformer builds upon the transformer architecture initially introduced by Vaswani et al. in 2017 for NLP tasks.
- Transformers are highly effective at processing sequential data, utilizing self-attention to model dependencies between different parts of the input.
- Vision Transformers apply this architecture to image data, treating the image as a sequence of patches instead of a grid of pixels.

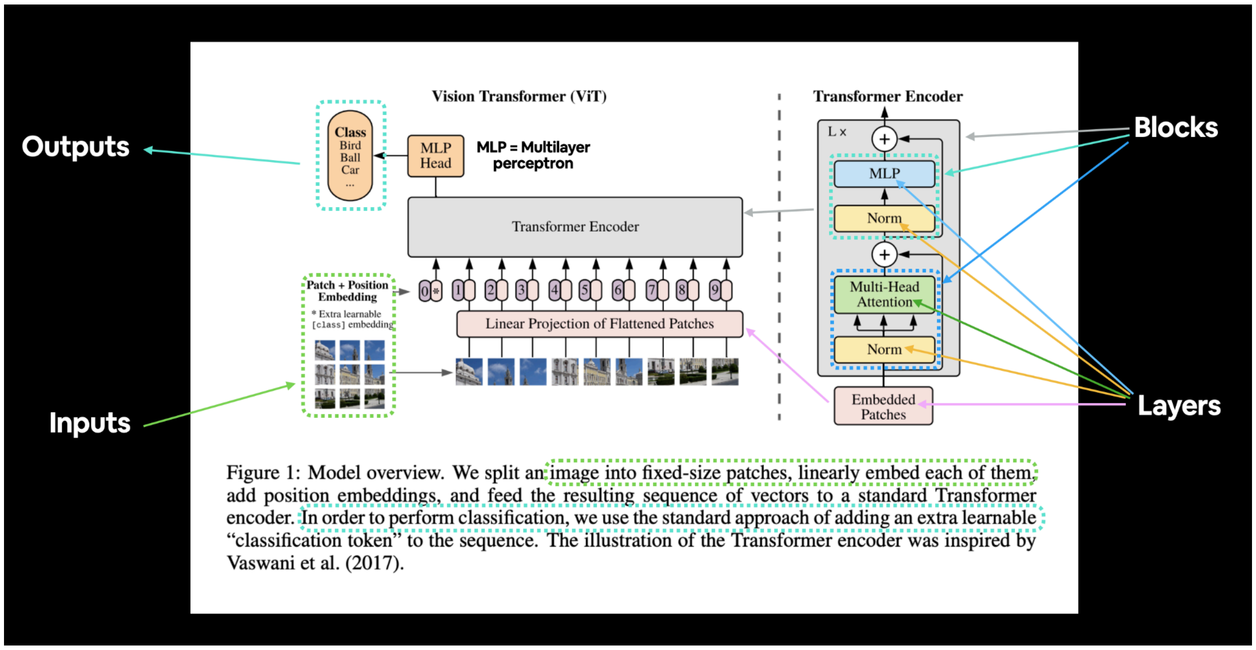


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “classification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

Vision Transformer architecture comprises several key stages:

- Image Patching and Embedding
- Positional Encoding
- Transformer Encoder
- Classification Head (MLP Head)

Architecture and Working of Vision Transformer (ViT)

- Input Image Processing: The input image is divided into patches, which are flattened and embedded using a linear projection.
- Positional Encoding: Positional encodings are added to the patch embeddings to retain spatial information.
- Transformer Encoder: The patch embeddings (along with the CLS token) are passed through multiple transformer encoder layers, which include multi-head self-attention and feed-forward networks.
- Classification Head: The CLS token's output is extracted and fed into an MLP for final classification.
- Vision Transformer (ViT) vs. Convolutional Neural Networks (CNNs)
- Local vs. Global Attention: CNNs capture local spatial features through convolutions, whereas ViTs capture global relationships using self-attention.
- Inductive Biases: CNNs have built-in inductive biases, such as locality and translation invariance, which make them effective at image tasks with smaller datasets. Vision Transformers, on the other hand, rely on data to learn these patterns, making them more flexible but data-hungry.

Training Data Requirements: Vision Transformers typically require large amounts of training data to achieve their best performance, while CNNs can perform well even with smaller datasets.

Strengths of Vision Transformer Architecture

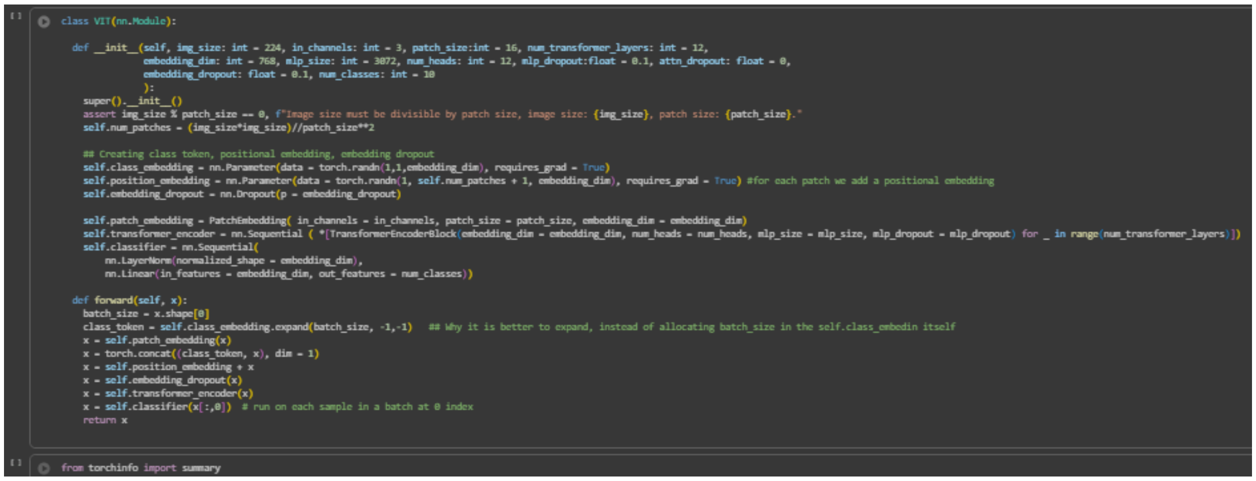
- Global Context: ViTs are excellent at capturing long-range dependencies between image patches, giving them a better understanding of the global context of images.
- Scalability: ViTs scale well with larger datasets and deeper architectures, making them highly effective for large-scale vision tasks.
- Flexibility: Since ViTs don't rely on convolutions, they can be easily adapted to different tasks and domains beyond image classification, such as video analysis and object detection.

Challenges and Limitations

- Data Requirements: ViTs are data-hungry and typically require large-scale datasets for effective training. Fine-tuning ViTs pre-trained on large datasets (like ImageNet) is a common practice when data is limited.
- Computational Resources: Vision Transformers demand significant computational power, especially during training, due to the quadratic complexity of the self-attention mechanism.

Vision Transformer architecture marks a significant shift in how visual data is processed by leveraging the self-attention mechanism of transformers. It excels in capturing global relationships between image patches, providing a powerful alternative to traditional convolutional neural networks.
While ViTs are highly flexible and scalable, they require large datasets and computational resources to realize their full potential. As research and development in this field continue, ViTs are expected to play an increasingly important role in the future of computer vision tasks.

Implementation:



```
class VIT(nn.Module):

  def __init__(self, img_size: int = 224, in_channels: int = 3, patch_size:int = 16, num_transformer_layers: int = 12,
               embedding_dim: int = 768, mlp_size: int = 3072, num_heads: int = 12, mlp_dropout:float = 0.1, attn_dropout: float = 0,
               embedding_dropout: float = 0.1, num_classes: int = 10
               ):
    super().__init__()
    assert img_size % patch_size == 0, f"Image size must be divisible by patch size, image size: {img_size}, patch size: {patch_size}."
    self.num_patches = (img_size*img_size)//patch_size**2

    ## Creating class token, positional embedding, embedding dropout
    self.class_embedding = nn.Parameter(data = torch.randn(1,1,embedding_dim), requires_grad = True)
    self.position_embedding = nn.Parameter(data = torch.randn(1, self.num_patches + 1, embedding_dim), requires_grad = True) #for each patch we add a positional embedding
    self.embedding_dropout = nn.Dropout(p = embedding_dropout)

    self.patch_embedding = PatchEmbedding( in_channels = in_channels, patch_size = patch_size, embedding_dim = embedding_dim)
    self.transformer_encoder = nn.Sequential ( *[TransformerEncoderBlock(embedding_dim = embedding_dim, num_heads = num_heads, mlp_size = mlp_size, mlp_dropout = mlp_dropout) for _ in range(num_transformer_layers)])
    self.classifier = nn.Sequential(
        nn.LayerNorm(normalized_shape = embedding_dim),
        nn.Linear(in_features = embedding_dim, out_features = num_classes))

  def forward(self, x):
    batch_size = x.shape[0]
    class_token = self.class_embedding.expand(batch_size, -1,-1)   ## Why it is better to expand, instead of allocating batch_size in the self.class_embedin itself
    x = self.patch_embedding(x)
    x = torch.concat((class_token, x), dim = 1)
    x = self.position_embedding + x
    x = self.embedding_dropout(x)
    x = self.transformer_encoder(x)
    x = self.classifier(x[:,0])  # run on each sample in a batch at 0 index
    return x
```

```
from torchinfo import summary
```

We trained this architecture on both CIFAR - 10 dataset and MNIST for just 15 epochs to gauge performance and we achieved an accuracy of 57 % both test and training on CIFAR 10 and 98% on MNIST classification.

This was expected as CIFAR - 10 is a more complex dataset - 60,000 colored images across 10 classes that are vastly different from each other

(a) Part images of CIFAR-10

(b) Part images of MNIST

Hence to perform better on CIFAR - 10, we had to increase the number of epochs.