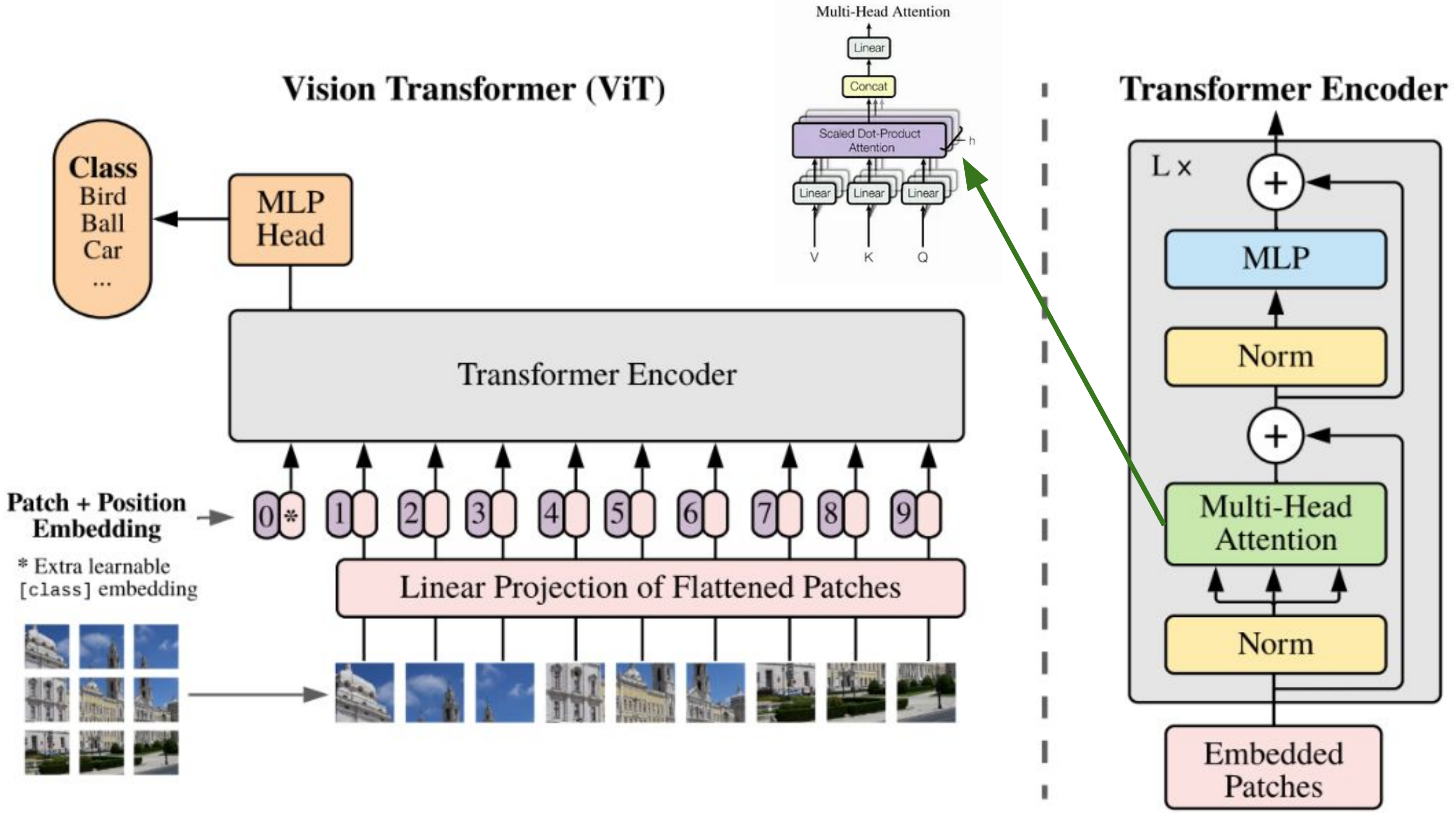
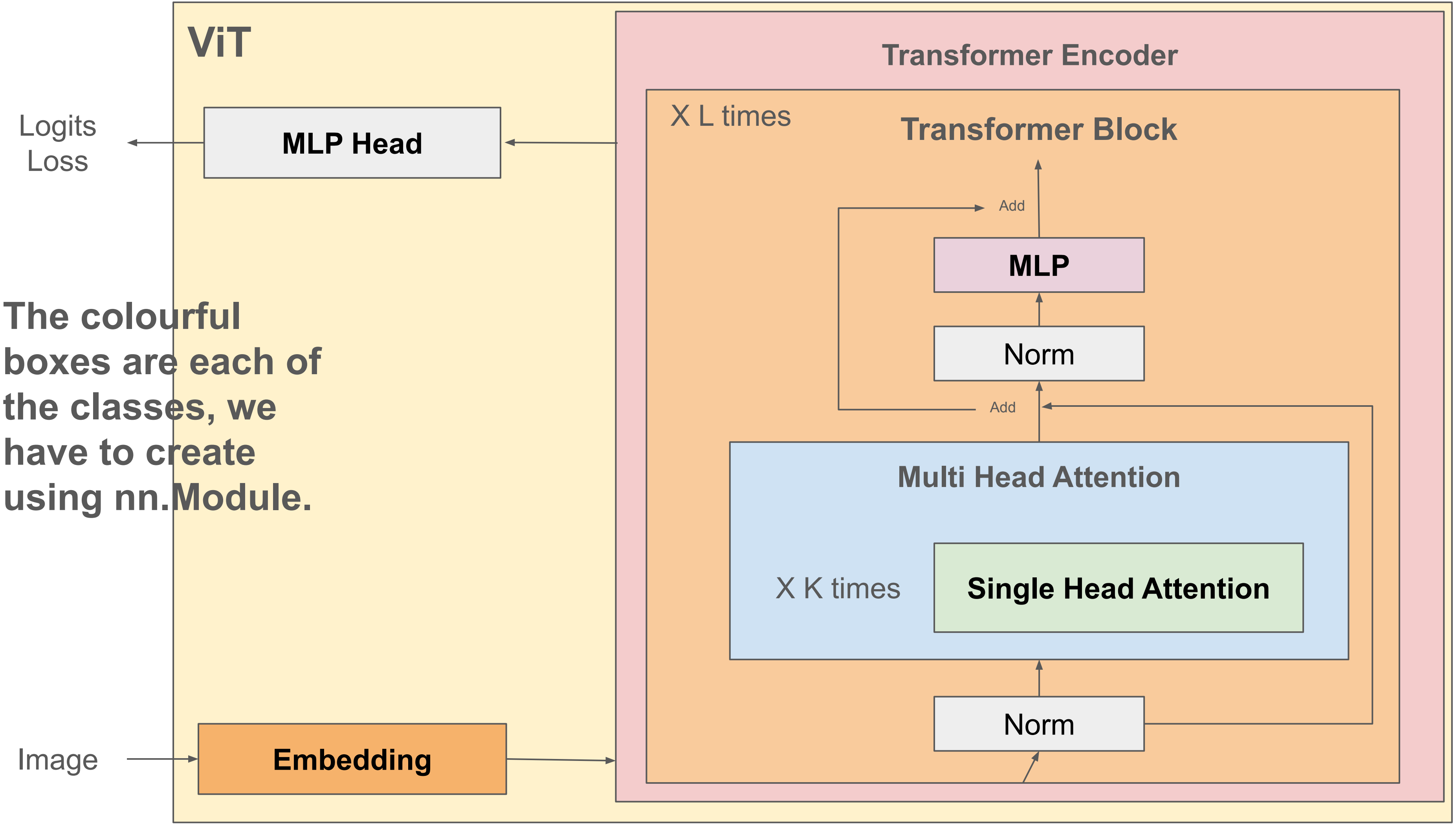
Deep Learning Papers from Scratch
Tutorial Series

Vision Transformer (ViT) *Model* from Scratch

Srijit Mukherjee
Pennsylvania State University

Understanding the Model


Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q
Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0
*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches
Transformer Encoder
L ×
+
MLP
Norm
+
Multi-Head Attention
Norm
Embedded Patches



The colourful boxes are each of the classes, we have to create using nn.Module.

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.

Similar to BERT's `[class]` token, we prepend a learnable embedding to the sequence of embedded patches ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$), whose state at the output of the Transformer encoder ($\mathbf{z}_L^0$) serves as the image representation $\mathbf{y}$ (Eq. 4). Both during pre-training and fine-tuning, a classification head is attached to $\mathbf{z}_L^0$. 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.

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.

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).

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

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

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

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

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

B.1 TRAINING

Table 3 summarizes our training setups for our different models. We found strong regularization to be key when training models from scratch on ImageNet. Dropout, when used, is applied after every dense layer except for the the qkv-projections and directly after adding positional- to patch embeddings. Hybrid models are trained with the exact setup as their ViT counterparts. Finally, all training is done on resolution 224.

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.

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

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

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

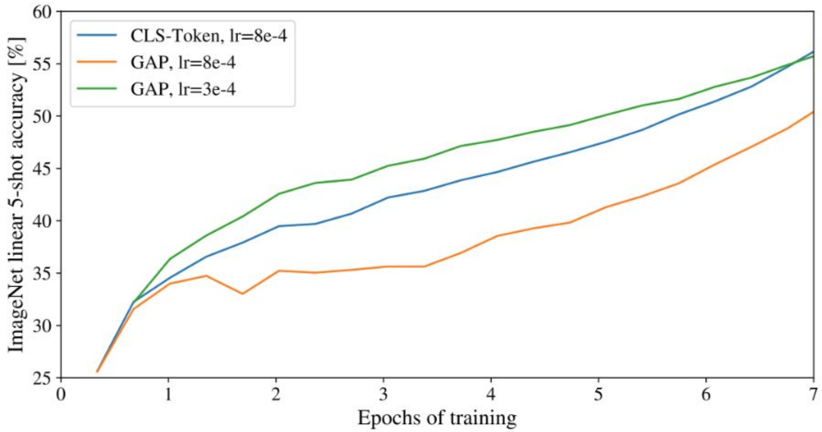
Multihead self-attention (MSA) is an extension of SA in which we run k self-attention operations, called "heads", in parallel, and project their concatenated outputs. To keep compute and number of parameters constant when changing k, D_h (Eq. 5) is typically set to D/k.

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

D.3 HEAD TYPE AND CLASS TOKEN

In order to stay as close as possible to the original Transformer model, we made use of an additional [class] token, which is taken as image representation. The output of this token is then transformed into a class prediction via a small multi-layer perceptron (MLP) with tanh as non-linearity in the single hidden layer.

This design is inherited from the Transformer model for text, and we use it throughout the main paper. An initial attempt at using only image-patch embeddings, globally average-pooling (GAP) them, followed by a linear classifier—just like ResNet's final feature map—performed very poorly. However, we found that this is neither due to the extra token, nor to the GAP operation. Instead, the difference in performance is fully explained by the requirement for a different learning-rate

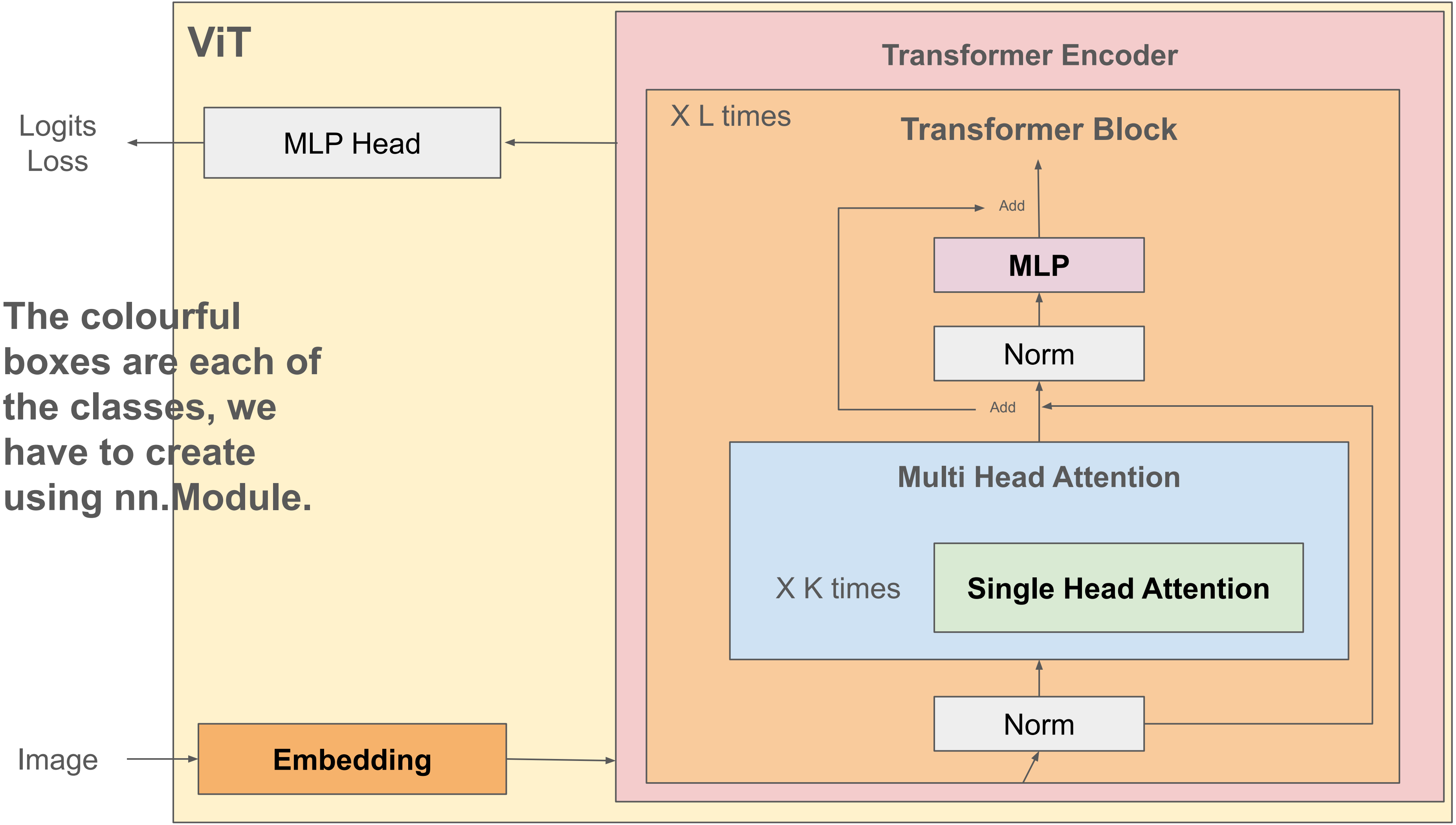


D.4 Positional Embedding

We ran ablations on different ways of encoding spatial information using positional embedding. We tried the following cases:

- Providing no positional information: Considering the inputs as a *bag of patches*.
- 1-dimensional positional embedding: Considering the inputs as a sequence of patches in the raster order (default across all other experiments in this paper).
- 2-dimensional positional embedding: Considering the inputs as a grid of patches in two dimensions. In this case, two sets of embeddings are learned, each for one of the axes, X-embedding, and Y-embedding, each with size $D/2$. Then, based on the coordinate on the path in the input, we concatenate the X and Y embedding to get the final positional embedding for that patch.
- Relative positional embeddings: Considering the relative distance between patches to encode the spatial information as instead of their absolute position. To do so, we use 1-dimensional Relative Attention, in which we define the relative distance all possible pairs of patches. Thus, for every given pair (one as query, and the other as key/value in the attention mechanism), we have an offset $p_q - p_k$, where each offset is associated with an embedding. Then, we simply run extra attention, where we use the original query (the content of query), but use relative positional embeddings as keys. We then use the logits from the relative attention as a bias term and add it to the logits of the main attention (content-based attention) before applying the softmax.

In addition to different ways of encoding spatial information, we also tried different ways of incorporating this information in our model. For the 1-dimensional and 2-dimensional positional embeddings, we tried three different cases: (1) add positional embeddings to the inputs right after the stem of them model and before feeding the inputs to the Transformer encoder (default across all other experiments in this paper); (2) learn and add positional embeddings to the inputs at the beginning of each layer; (3) add a learned positional embeddings to the inputs at the beginning of each layer (shared between layers).



The colourful boxes are each of the classes, we have to create using nn.Module.

Building the Model

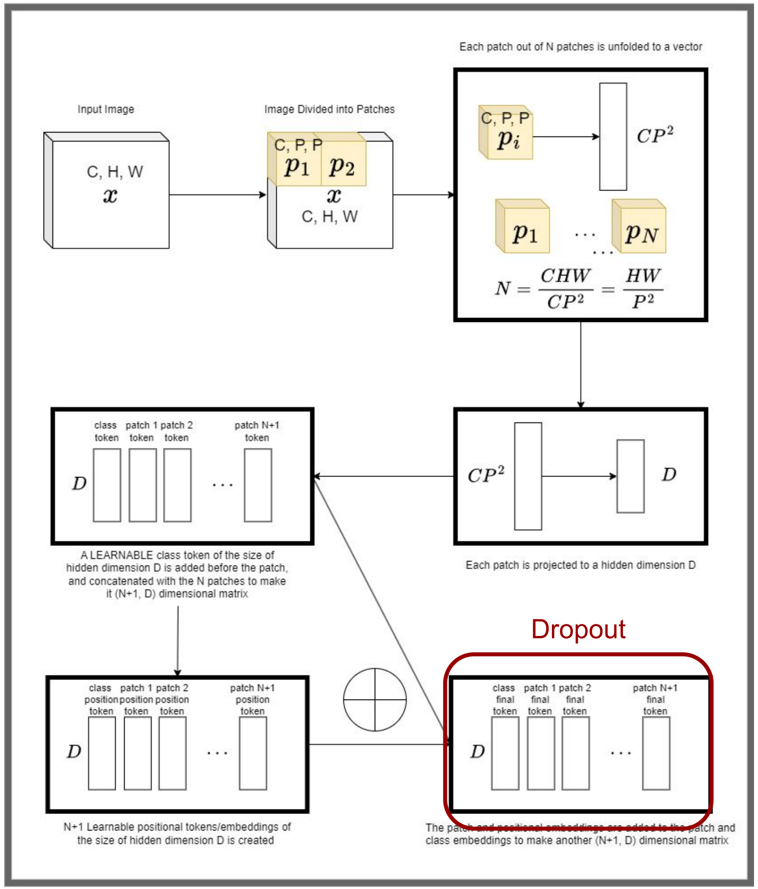
nn.Module classes to create

- Embedding
- Single_Head_Attention
- Multi_Head_Attention
- MLP
- Transformer_Block
- ViT (everything put together)

Parameters

```python
# training parameters
B = 32 # batch size

# image parameters
C = 3
H = 128
W = 128
x = torch.rand(B, C, H, W)

#model parameters
D = 64 # hidden size
P = 4 #patch size
N = int(H*W/P**2)#number of tokens
k = 4 # number of attention heads
Dh = int(D/k) # attention head size
p = 0.1 # dropout rate
mlp_size = D*4 # mlp size
L = 4 # number of transformer blocks
n_classes = 3 # number of classes
```

Input Image Embedding Creation



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

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

```python
# Image Embeddings [Patch, Class, with Position Embeddings]

class Embedding(nn.Module):
    def __init__(self):
        super(Embedding, self).__init__()

        self.unfold = nn.Unfold(kernel_size=P, stride=P) # function to create patch vectors (x_p^i)
        self.project = nn.Linear(P**2 * C, D) # patch tokens (E)
        self.cls_token = nn.Parameter(torch.randn((1, 1, D))) # function to create unbatched class token (x_class) as trainable parameter
        self.pos_embedding = nn.Parameter(torch.randn(1, N+1, D)) # function to create unbatched position embedding (E_pos) as trainable parameter
        self.dropout = nn.Dropout(p) #dropout

        #why unbatched? because we are setting the parameters and functions here.
        # giving batched will increase the parameter size without effectively increasing the parameters

    def forward(self, x):

        print("######")
        print("input image:", x.shape)
        x = self.unfold(x).transpose(1,2) # patch vectors (x_p^i)
        print("x_p^i:", x.shape)
        x = self.project(x)
        print("x_p^i*E: ", x.shape) # tokens for patches (x_p^i*E)
        cls_token = self.cls_token # unbatched class token (x_class)
        print("unbatched x_class:", cls_token.shape)
        cls_token = self.cls_token.expand(B, -1, -1) # batched class token (x_class)
        print("x_class:", cls_token.shape)
        x = torch.cat((cls_token, x), dim = 1) # final image token embedding
        print("patch embedding:", x.shape)
        pos_embedding = self.pos_embedding # unbatched position embedding (E_pos)
        print("unbatched E_pos:", pos_embedding.shape)
        pos_embedding = pos_embedding.expand(B, -1, -1) # batched position embedding (E_pos)
        print("E_pos:", pos_embedding.shape)
        z0 = x + pos_embedding # adding the batched position and image embedding
        print("z0:", z0.shape)
        z0 = self.dropout(z0) # dropout
        return z0
```

Single Head Attention

$$\text{input sequence } \mathbf{z} \in \mathbb{R}^{N \times D}$$

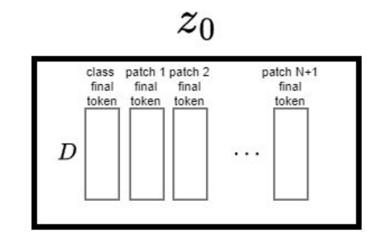
$$\mathbf{U}_{qkv} \in \mathbb{R}^{D \times 3D_h},$$

$$A \in \mathbb{R}^{N \times N},$$

$$[\mathbf{q}, \mathbf{k}, \mathbf{v}] = \mathbf{z}\mathbf{U}_{qkv}$$

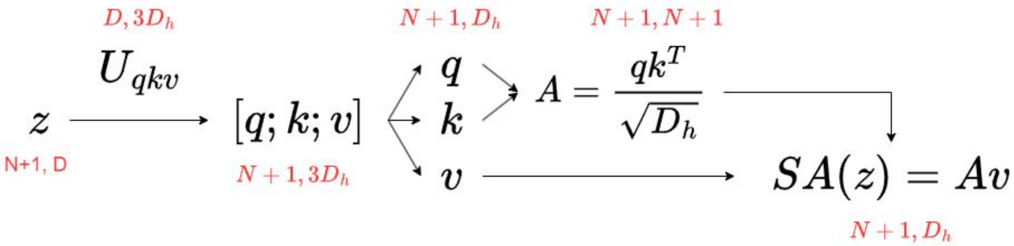
$$A = \text{softmax}\left(\mathbf{q}\mathbf{k}^\top / \sqrt{D_h}\right)$$

$$\text{SA}(\mathbf{z}) = A\mathbf{v}\,.$$



The patch and positional embeddings are added to the patch and class embeddings to make another (N+1, D) dimensional matrix

NO Dropout after Dense Layer in QKV projection

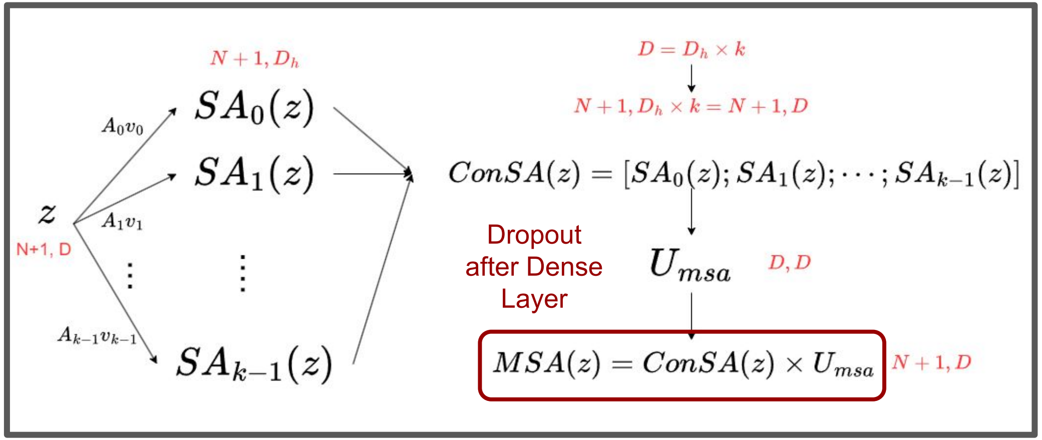



```python
# Single Head Attention


class Single_Head_Attention(nn.Module):
    def __init__(self):
        super(Single_Head_Attention, self).__init__()

        self.U_qkv = nn.Linear(D, 3*Dh) # U_qkv
        self.softmax = nn.Softmax(dim = -1) # softmax along the last dimension

    def forward(self, z):

      print("z:", z.shape)
      qkv = self.U_qkv(z) # qkv
      print("qkv:", qkv.shape)
      q = qkv[:, :, :Dh] # q
      print("q:", q.shape)
      k = qkv[:, :, Dh:2*Dh] # k
      print("k:", k.shape)
      v = qkv[:, :, 2*Dh:] # v
      print("v:", v.shape)
      qkTbysqrtDh = torch.matmul(q, k.transpose(-2, -1))/math.sqrt(Dh) # qk^T/sqrtDh
      print("qkTbysqrtDh:", qkTbysqrtDh.shape)
      A = self.softmax(qkTbysqrtDh) # A
      print("A:", A.shape)
      SAz = torch.matmul(A, v) # z = Av
      print("SA(z):", SAz.shape)

      return SAz
```

Multi Head Attention

$$\mathrm{MSA}(\mathbf{z}) = [\mathrm{SA}_1(z); \mathrm{SA}_2(z); \cdots ; \mathrm{SA}_k(z)]\, \mathbf{U}_{msa}$$

$$\mathbf{U}_{msa} \in \mathbb{R}^{k \cdot D_h \times D}$$; k, D_h (Eq. 5) is typically set to D/k.

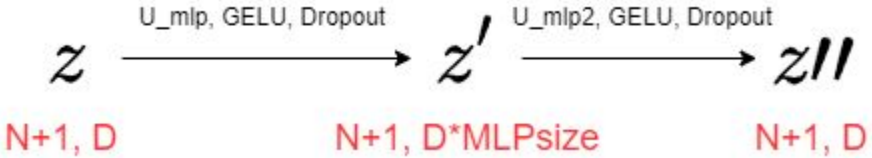



```python
# Multi Head Self Attention

class Multi_Head_Self_Attention(nn.Module):
    def __init__(self):
        super(Multi_Head_Self_Attention, self).__init__()

        self.heads = nn.ModuleList([Single_Head_Attention() for _ in range(k)]) # k heads
        self.U_msa = nn.Linear(D, D) # U_msa
        self.dropout = nn.Dropout(p) #dropout

    def forward(self, z):

      print("######")
      print("z:", z.shape)
      ConSAz = torch.cat([head(z) for head in self.heads], dim = -1)
      print("ConSA(z):", ConSAz.shape)
      msaz = self.U_msa(z) # MSA(z)
      print("MSA(z):", msaz.shape)
      msaz = self.dropout(msaz) # dropout

      return msaz
```

- A faster version of Multi Head Attention can be implemented where all the k single head attentions can have shared weights. This decreases the number of parameters and hence the training time.

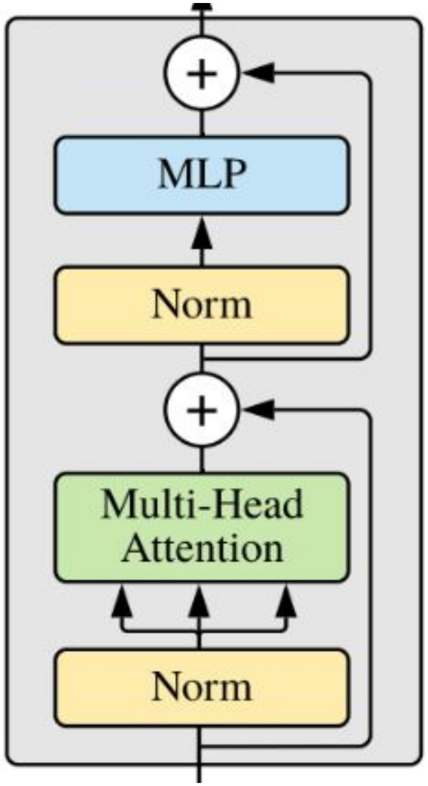
MLP

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




```python
# MLP

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()

        self.U_mlp = nn.Linear(D, mlp_size)
        self.gelu = nn.GELU()
        self.U_mlp2 = nn.Linear(mlp_size, D)
        self.dropout = nn.Dropout(p)

    def forward(self, z):

      print("###MLP###")
      print("z:", z.shape)
      z = self.U_mlp(z) # mlp
      print("mlp(z):", z.shape)
      z = self.gelu(z) # gelu
      print("gelu(mlp(z)):", z.shape)
      z = self.dropout(z) # dropout
      z = self.U_mlp2(z) # mlp2
      print("mlp2(gelu(mlp(z))):", z.shape)
      z = self.gelu(z) # gelu
      print("gelu(mlp2(gelu(mlp(z)))):", z.shape)
      z = self.dropout(z) # dropout

      return z
```

Transformer Block



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

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

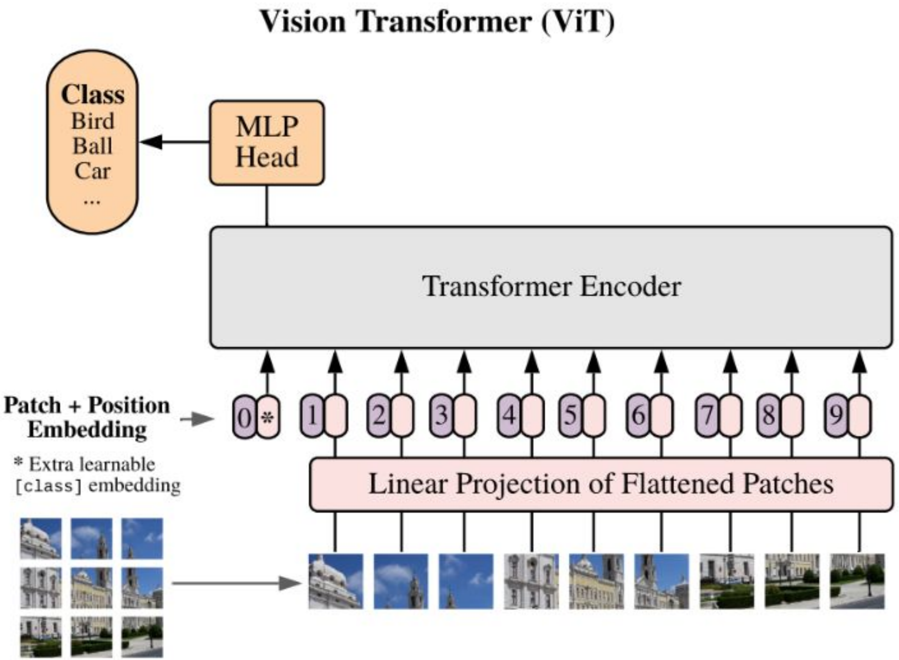
```python
# Transformer Block

class Transformer_Block(nn.Module):
    def __init__(self):
        super(Transformer_Block, self).__init__()

        self.layernorm_1 = nn.LayerNorm(D)
        self.msa = Multi_Head_Self_Attention()
        self.layernorm_2 = nn.LayerNorm(D)
        self.mlp = MLP()

    def forward(self, z):

      print("###Transformer Block###")
      print("z:", z.shape)
      z1 = self.layernorm_1(z) # layer norm 1 output
      print("layernorm_1(z):", z1.shape)
      z1 = self.msa(z1) # multi head self attention
      print("msa(layernorm_1(z)):", z1.shape)
      z2 = z + z1
      print("z + msa(layernorm_1(z)):", z2.shape)
      z3 = self.layernorm_2(z2) # layer norm 2 output
      print("layernorm_2(z + msa(layernorm_1(z))):", z3.shape)
      z3 = self.mlp(z3) # mlp
      print("mlp(layernorm_2(z + msa(layernorm_1(z)))):", z3.shape)
      z4 = z2 + z3
      print("z2 + mlp(layernorm_2(z + msa(layernorm_1(z)))):", z4.shape)

      return z4
```

ViT

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

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

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




```python
# ViT

class ViT(nn.Module):
    def __init__(self):
        super(ViT, self).__init__()

        self.embedding = Embedding()
        self.transformer_encoder = nn.ModuleList([Transformer_Block() for _ in range(L)])
        self.layernorm = nn.LayerNorm(D)
        self.U_mlp = nn.Linear(D, n_classes)

    def forward(self, x):

      print("###ViT###")
      print("input image:", x.shape)
      z = self.embedding(x)
      print("z:", z.shape)
      for block in self.transformer_encoder:
        z = block(z)
      print("z:", z.shape)
      z = self.layernorm(z)
      print("layernorm(z):", z.shape)
      z = z[:, 0, :]
      print("z:", z.shape)
      z = self.U_mlp(z)
      print("mlp(layernorm(z)):", z.shape)

      return z
```

Important Questions

- Use of Class Token in Vision Transformer (Stack Exchange)
- Use of Class Token in Vision Transformer (Reddit)
- My GitHub Repo for this ViT Model in Pytorch