PRACTICE #03

Vision Transformer

(*Keyword:* ViT)

I. Goals

The objective of this assignment is to introduce the Vision Transformer model, how to implement a ViT Model from scratch using PyTorch, and train your own model for image classification tasks.

II. Introduction

Inspired by Transformer, one of the most successful deep learning models in natural language processing, machine translation, etc. Vision Transformer (ViT) has recently demonstrated its effectiveness in computer vision tasks such as image classification, object detection, etc. This is the link to the original vision transformer paper: https://arxiv.org/abs/2010.11929. A vision transformer model (ViT) is made up of three primary modules: a linear projection for patch embedding, a sequence of transformer blocks, and several fully connected layers for the classification head.

- Firstly, ViT takes an input image of size W x H x U where W, H are the spatial sizes, U is the number of channels. After that, we split the input image into a sequence of patches, each of which is of size WH/P^2 x UP^2 where P is the patch size

- Secondly, these patches are linearly transformed into a D-dimensional space to produce N patch embeddings $z_n \in \mathbb{R}^D$, where D is the embedding dimension. To capture positional information, positional encoding is added to z_n. Consequently, we collect N embedding patches z_n to generate a matrix $Z \in \mathbb{R}^{N \times D}$ before feeding it into transformer blocks, each of which is composed of a multi-head self-attention layer (MSA) and feed-forward blocks
- Finally, a sequence of fully connected layers is applied to the class token to calculate the prediction score for classification.

III. Content

1. Implement attention module
   - Input: X of size `(batch_size, seq_len, embed_dim)` where `seq_len` is the number of tokens, embed_dim is the dimension of embedding
   - To compute the multihead self-attention, we perform the following steps:

- Step 1: Compute Q = XW_Q, K=XW_K, V = XWV where WQ, WK, WV are trainable weights of size `(embed_dim, embed_dim)`
- Step 2: Resize Q, K, V to size `(batch_size, seq_len, heads, embed_dim // heads)` where heads are number of heads, and then permute it to size `(batch_size, heads, seq_len, embed_dim // heads)`
- Step 3: Compute attention

$$attention = softmax(\frac{QK^T}{\sqrt{embed\_\mathrm{dim}\ //\ heads}})V$$

- Step 4: return output = attention . V

- Output is of size (batch_size, seq_len, embed_dim)

```
class Attention(nn.Module):
    '''
    Attention Module is used to perform self-attention operation allowing the
model to attend information from different representation subspaces on an input
sequence of embeddings.

    Args:
        embed_dim: Dimension size of the hidden embedding
        heads: Number of parallel attention heads
    Methods:
        forward(inp) :-
        Performs the self-attention operation on the input sequence embedding.
        Returns the output of self-attention can be seen as an attention map
        inp (batch_size, seq_len, embed_dim)
        out:(batch_size, seq_len, embed_dim),
    Examples:
        >>> attention = Attention(embed_dim, heads, activation, dropout)
        >>> out = attention(inp)
      def __init__(self, heads, embed_dim):
        super(Attention, self).__init__()

      def forward(self, inp):
        # inp: (batch_size, seq_len, embed_dim)
```

2. Implement TransformerBlock

```
class TransformerBlock(nn.Module):
    '''
    Transformer Block combines both the attention module and the feed-forward
module with layer normalization, dropout and residual connections. The sequence
of operations is as follows :-

Inp -> LayerNorm1 -> Attention -> Residual -> LayerNorm2 -> FeedForward -> Out
 |                                    |   |                                 |
 |-------------Addition--------------|   |---------------Addition------------|

    Args:
        embed_dim: Dimension size of the hidden embedding
```

```
        heads: Number of parallel attention heads (Default=8)
        mlp_dim: The higher dimension is used to transform the input embedding
and then resized back to embedding dimension to capture richer information.
        dropout: Dropout value for the layer on attention_scores (Default=0.1)

    Methods:
        forward(inp) :-
        Applies the sequence of operations mentioned above.
        (batch_size, seq_len, embed_dim) -> (batch_size, seq_len, embed_dim)

    Examples:
        >> TB = TransformerBlock(embed_dim, mlp_dim, heads, activation, dropout)
        >> out = TB(inp)
    '''
    def __init__(self, embed_dim, mlp_dim, heads, dropout=0.1):
        super(TransformerBlock, self).__init__()
        self.attention = MultiheadAttention(heads, embed_dim)
        self.fc1 = nn.Linear(embed_dim, mlp_dim)
        self.fc2 = nn.Linear(mlp_dim, embed_dim)
        self.activation = nn.ReLU()
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)

    def forward(self, inp):
          # inp: (batch_size, seq_len, embed_dim)
```

3. Implement class Transformer

```
class Transformer(nn.Module):
    '''
    Transformer combines multiple layers of Transformer Blocks in a sequential
manner. The sequence
    of the operations is as follows -

    Input -> TB1 -> TB2 -> .......... -> TBn (n being the number of layers) ->
Output

    Args:
        embed_dim: Dimension size of the hidden embedding in the TransfomerBlock
        mlp_dim: Dimension size of MLP layer in the TransfomerBlock
        layers: Number of Transformer Blocks in the Transformer
        heads: Number of parallel attention heads (Default=8)
        dropout: Dropout value for the layer on attention_scores (Default=0.1)

    Methods:
        forward(inp) :-
        Applies the sequence of operations mentioned above.
        (batch_size, seq_len, embed_dim) -> (batch_size, seq_len, embed_dim)
```

```
    Examples:
        >>> transformer = Transformer(embed_dim, layers, heads, activation,
forward_expansion, dropout)
        >>> out = transformer(inp)
    '''
    def __init__(self, embed_dim, layers, heads=8, dropout=0.1):
        super(Transformer, self).__init__()
        self.embed_dim = embed_dim
        self.trans_blocks = nn.ModuleList(
            [TransformerBlock(embed_dim, mlp_dim, heads, dropout)
            for i in range(layers)]
        )

    def forward(self, inp):
        # inp: (batch_size, seq_len, embed_dim)
```

4. Implement ClassificationHead

```
class ClassificationHead(nn.Module):
    '''
    Classification Head attached to the first sequence token which is used as
the arbitrary
    classification token and used to optimize the transformer model by applying
Cross-Entropy
    loss. The sequence of operations is as follows :-

    Input -> FC1 -> GELU -> Dropout -> FC2 -> Output

    Args:
        embed_dim: Dimension size of the hidden embedding
        classes: Number of classification classes in the dataset
        dropout: Dropout value for the layer on attention_scores (Default=0.1)

    Methods:
        forward(inp) :-
        Applies the sequence of operations mentioned above.
        (batch_size, embed_dim) -> (batch_size, classes)

    Examples:
        >>> CH = ClassificationHead(embed_dim, classes, dropout)
        >>> out = CH(inp)
    '''
    def __init__(self, embed_dim, classes, dropout=0.1):
        super(ClassificationHead, self).__init__()
        self.embed_dim = embed_dim
        self.classes = classes
        self.fc1 = nn.Linear(embed_dim, embed_dim // 2)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(embed_dim // 2, classes)
        self.softmax = nn.Softmax(dim=-1)
        self.dropout = nn.Dropout(dropout)

    def forward(self, inp):

        # inp: (batch_size, embed_dim)
```

5. Complete VisionTransformer

How to train a vision transformer model:

- Step 1: Extract patches by cutting the input image into patches `(batch_size, num_patches, channels)`
- Step 2: Convert patches into a sequence of embedding vectors `(batch_size, num_patches, embed_dim)`
- Step 3: Prepend a classification token to embedding vectors
  `torch.cat([class_token, patches], dim=1)`
- Step 4: Add positional embeddings to embedding vectors `(batch_size, seq_len, embed_dim)` (seq_len = num_patches + 1)
- Step 5: Pass the embedding vectors through a sequence of Transformer Blocks `(batch_size, seq_len, embed_dim)`
- Step 6: Extract the classification token from final output of the Transformer Blocks to pass through a ClassificationHead

```
class VisionTransformer(nn.Module):
    '''
    Vision Transformer is the complete end-to-end model architecture that
combines all the above modules in a sequential manner. The sequence of the
operations is as follows -

 Input -> CreatePatches -> ClassToken, PatchToEmbed , PositionEmbed -> Transformer -> ClassificationHead -> Output
                                   |             | |               |
                                   |---Concat---| |----Addition----|


    Args:
        patch_size: Length of square patch size
        max_len: Max length of learnable positional embedding
        embed_dim: Dimension size of the hidden embedding
        mlp_dim: Dimension size of MLP embedding
        classes: Number of classes in the dataset
        layers: Number of Transformer Blocks in the Transformer
        channels: Number of channels in the input (Default=3)
        heads: Number of parallel attention heads (Default=8)


        dropout: Dropout value for the layer on attention_scores (Default=0.1)

    Methods:
        forward(inp) :-
        Applies the sequence of operations mentioned above.
        It outputs the classification output as well as the sequence output of
the transformer
        (batch_size, channels, width, height) -> (batch_size, classes),
(batch_size, seq_len+1, embed_dim)

    Examples:
```

```
        >>> ViT = VisionTransformer(inp_channels, patch_size, max_len, heads, 
classes, layers, embed_dim, mlp_dim, channels,dropout)
        >>> class_out, hidden_seq = ViT(inp)
    '''
    def __init__(self, inp_channels, patch_size, max_len, heads, classes, 
layers, embed_dim, mlp_dim, dropout):
        super(VisionTransformer, self).__init__()


    def forward(self, inp):

        # inp: (batch_size, channels, width, height)
```

IV. Requirements

1. In this assignment, you are required to implement from scratch a ViT model for image classification task on CIFAR10 dataset.
2. Train ViT model from scratch on CIFAR-10 dataset, and evaluate your trained model using top-1 accuracy metric
3. Run some experiments by changing hyperparameters (embed_dim, mlp_dim, patch_size, heads, layers), compare the performance of each experiment, and give your comment
4. The directory structure of the compressed submission
   - *doc*: a report file *StudentID_report_p03.doc/docx/pdf*. The report file includes your personal information (StudentID, name), describes the architecture of the ViT you have built, your experiments, and presents the results of these experiments, and gives your observations about those results.
   - *source*: contains entire source code
   - *bonus*: optional, for plus points, if available
   A separate report doc file is required even if students use Jupyter Notebook file (named StduentID_p03).
5. Other requirements
   - The report should be presented clearly and intuitively: a self-scoring table (assessment) of the results of the work compared to the corresponding requirements (0-100%), a list of the functions included in the program with proof images, summarize the usage and implementation (for example: through pseudo-code, description of methods, or how to do it, *do not copy the source code into the report*).
   - The source code needs to be commented on the corresponding lines.