Masked Autoencoders Are Scalable Vision Learners

The idea of masked autoencoder for computer vision got inspired from the concept of predicting a removed word from a sentence in NLP, but for image data. The Masked autoencoder for vision learning focuses on how to reconstruct images from the missing parts using autoencoder's conception with vision transformer.

If we consider a visual puzzle, how will a model just going to rightfully find the missing part by looking at the surrounding? The construction of masked autoencoders come up from the structure of attention mechanism and ViT.
The authors show how unsupervised learning of images can be achieved through autoencoders. As the purpose is to generate the removed part of the image, the model comes up with both encoder and a decoder.

The following sections cover the construction and explanation of MAE for visions in TensorFlow framework.

Patching

AT first the patches will be generated and applied over the entire image. The partches are non-overlapping. The authors specify RGB image with size of (224 x 224) pixels and patch size of (16 x 16). Each image patch is then converted to lower dimensional space based on the total patches and channels.

```
num_patches = (img_size[0] // patch_size[0]) * (img_size[1] // patch_size[1])
self.patch_shape = (img_size[0] // patch_size[0], img_size[1] // patch_size[1])
```

Now, if we look at the code below, the conv layer will create total numbers of filters based on given embed dim. It connects each patch to the embedding dimension space.

```
self.proj = Conv2D(filters=embed_dim,
                                                kernel_size=patch_size,
                                                strides=patch_size,
                                                padding='valid')
```

Mask Generator

The mask is generated which will be applied only on the missing part of the images, that is the part which needs to be predicted. The visible part is unmasked patches, which is only 25% of the whole image.
The masking ratio is set to 0.75, the masked part is 75% and 25% of the images remain unmasked that will be fed to the encoder.
So, we can see that, the encoder will be applied only on 25% of the entire image portion where mask generator won't be applied.

Mae Encoder

The encoder is basically a ViT, with self-attention. The encoder takes the position vectors of the patches.
The encoder block contains mlp, attention head and a transformer block.

Attention head:

To get the attentions on image, at first query, vkey, value needs to be obtained through linear transformation and reshaping

```
self.qkv = layers.Dense(all_head_dim * 3, use_bias=False)
```

to get the attention score just simply multiply query and keys

```
self.scale = qk_scale or head_dim ** -0.5
```

```
# Compute attention scores (QK^T)
    attn = tf.matmul(q, tf.transpose(k, perm=[0, 1, 3, 2]))

    # Apply softmax to get attention weights
    attn = tf.nn.softmax(attn, axis=-1)
    attn = self.attn_drop(attn, training=training)

    # Weighted sum using attention weights (AV)
    x = tf.matmul(attn, v)
```

Overall, the attention block is based on the structure that came from **Attention is All You Need**, which computes attention then reshapes it to get the attention weight and applies it to the image through matrix multiplication.

Encoder

Everything is put together for the encoder, which is a vit and returns a normalized layer.
The implemented encoder is a pretrained version of Vision transformer named **PretrainVisionTransformerEncoder.**
The class implements consists of patch embedding, positional encoding, a stack of transformer blocks, layer normalization, and a classification head.
Here, the transformer block is initialized earlier section. The block layer has the MLP and attention layer altogether, the we can see in the code below-

```
self.blocks = [Block(
                dim=embed_dim, num_heads=num_heads, mlp_ratio=mlp_ratio,
                qkv_bias=qkv_bias,
                qk_scale=qk_scale,
                drop=drop_rate, attn_drop=attn_drop_rate, drop_path=dpr[i],
                norm_layer=LayerNormalization,
                init_values=init_values)
                       for i in range(depth)]
```

The patch embeds and positional embeds constructed earlier are fed here. At first the patch embeds and positional embeds are fed to the layers. The authors decide to make the positional encoding optional between learning parameter or sine-cosine parameters.

After feeding them to the block layer we normalize the output values as shown in the following code.

```
    def call(self, x, mask):
        x = self.patch_embed(x)
        x = x + self.pos_embed
        for blk in self.blocks:
            x = blk(x)
        x = self.norm(x)
        x = self.head(x)
        return x
```

Decoder:

The decoder's construction is similar to the encoder. The only difference is what is being inserted to the layers. The inputs can be imparted in two ways

- Encoded, unmasked visible patches
- Mask tokens

The decoder has both classifier and a normalizer at the end of their building blocks.
Just like the encoder it also has a transformer block

```
# convert the list of drop probabilities to TensorFlow tensors
        dpr = [tf.convert_to_tensor(value=prob, dtype=tf.float32) for prob in
dpr]
            self.blocks = [Block(dim=embed_dim, num_heads=num_heads,
            mlp_ratio=mlp_ratio,
                qkv_bias=qkv_bias, qk_scale=qk_scale,
                drop=drop_rate, attn_drop=attn_drop_rate, drop_path=dpr[i],
                norm_layer=LayerNormalization,
                init_values=init_values)
                       for i in range(depth)]
```

Mae architecture:

After constructing encoder,decoder separately everything is put together for the **PretrainVisionTransformer class**. One thing to address in this case, is that here the full architecture is built upon pretrained frameworks. It encodes input features using the encoder and uses the decoder to generate the masked tokens based on the encoded inputs.

```
def forward(self, x, mask):

        x_vis = self.encoder(x, mask) # [B, N_vis, C_e]
        x_vis = self.encoder_to_decoder(x_vis) # [B, N_vis, C_d]

        B, N, C = x_vis.shape

        # we don't unshuffle the correct visible token order,
        # but shuffle the pos embedding accorddingly.
        expand_pos_embed = self.pos_embed.expand(B, -1, -
1).type_as(x).to(x.device).clone().detach()
        pos_emd_vis = expand_pos_embed[~mask].reshape(B, -1, C)
```

```
        pos_emd_mask = expand_pos_embed[mask].reshape(B, -1, C)
        x_full = torch.cat([x_vis + pos_emd_vis, self.mask_token + pos_emd_mask],
dim=1)
        # notice: if N_mask==0, the shape of x is [B, N_mask, 3 * 16 * 16]
        x = self.decoder(x_full, pos_emd_mask.shape[1]) # [B, N_mask, 3 * 16 *
16]
        return x
```

Conclusion:

The model is still under developing, as some errors haven't been figured out yet. Hence, it was not possible to see the total number of parameters in the TensorFlow implementation.

References:

1. Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. *arXiv:2111.06377*, 2021.
2. Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In ICLR, 2021.