Vision Transformers

Part 1: Introduction

In this lab, we are asked to implement a Vision Transformer (ViT) to classify images in a 5 class dataset. We reuse the data loading code that was created in Lab 4. Furthermore, we reuse the image dataset that was created in Lab 4. After implementing the ViT, we train for 10 epochs and then provide a confusion matrix of the resulting model on the test dataset. We compare the results from the ViT with that of the CNN based model from Lab 4.

Part 2: Transformer Implementation

Transformer Implementation - Embedding Generation:

In order to implement the ViT on a dataset of 64 x 64 x 3 (H x W x C) images, we need to represent each “image patch” as an embedding. An image patch is a square region within the image. In this lab, 16 image patches are generated by applying a 3 channel to 100 channel convolution using a stride of 16 and kernel size of 16. The 100 channels are important because that will represent the size of the embedding for a particular patch. This will use a 3 x 100 x 16 x 16 kernel. Since the kernel is 16 x 16 with a stride of 16, there are 16 pixels in each resulting channel of the convolution. Then, a stack of pixels along the channel dimension is taken, resulting in a 1 x 100 embedding vector for each patch. In total, this process creates 16 distinct 1 x 100 embedding vectors. Put into a tensor together, that is a 16 x 100 tensor. This process is achieved via the following code:

```
class PatchEmbed(nn.Module): #Note, from Meta DINO ViT code ()
    """ Image to Patch Embedding
    """
    def __init__(self, img_size=64, patch_size=16, in_chans=3, embed_dim=100):
        super().__init__()
        num_patches = (img_size // patch_size) * (img_size // patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.num_patches = num_patches

        self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)

    def forward(self, x):
        B, C, H, W = x.shape
        y = self.proj(x)
        #print('conv output before flattening and transpose', y.shape)
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```

Transformer Implementation - Adding Positional and Class Embeddings

The ViT paper specifies to add a learnable class embedding as a row on top of the 16 x 100 tensor. This class embedding is created via the nn.Parameter command using an input as a random vector of size 1 x 100. Furthermore, a learnable position embedding is added to the now 17 x 100 tensor using nn.Parameter (but an input as a random tensor of the size 17 x 100).

Code for this is below:

```
class MasterEncoder(nn.Module):
    def __init__(self, max_seq_length, embedding_size, how_many_basic_encoders, num_atten_heads):
        super().__init__()
        self.class_embedding = nn.Parameter(torch.rand(size = (1, 1, embedding_size))) #create class embedding.
        #this class_embedding will be the first row of the embedding matrix, where axis 0 is each patch embedding
        #Note that the first 1 in the size parameter above corresponds to the batch size.
        self.pos_embedding = nn.Parameter(torch.rand(size = (1, max_seq_length, embedding_size)))
        self.patch_generator = PatchEmbed()
        self.max_seq_length = max_seq_length
        self.basic_encoder_arr = nn.ModuleList([BasicEncoder(
            max_seq_length, embedding_size, num_atten_heads) for _ in range(how_many_basic_encoders)])  # (A)
        self.mlp_head = nn.Linear(embedding_size, 5)
```

Transformer Implementation - Query, Key, Value logic with multiple attention heads

Now, we enrich these embeddings using multiple head Query, Key, and Value logic specified in the original transformer paper. The learnable Query, Key, and Value matrices are embedding_size x embedding_size dimension (WQ, WK, QV). Specifically, Q = embedding · WQ, K = embedding · WK, and V = embedding · WV. The resulting Q, K, and V tensors are then used to product a probability mass function interpretation of embeddings by applying a softmax to Q · K.T and dividing the result by the length of the square root of the embedding. That result is then multiplied by V.

After the attention blocks are processed and concatenated (because of the multiple heads), the output is subject to a Layer Normalization and then followed by a fully connected network that preserves the shape of the input and another Layer Normalization. A series of four consecutive attention blocks, layer normalizations, fully connected layers, and layer normalizations comprise the entire encoder network.

Note, most of the code for this was taken from ViTHelper.py which was provided in the lab documents.

Transformer Implementation - MLP Class Prediction

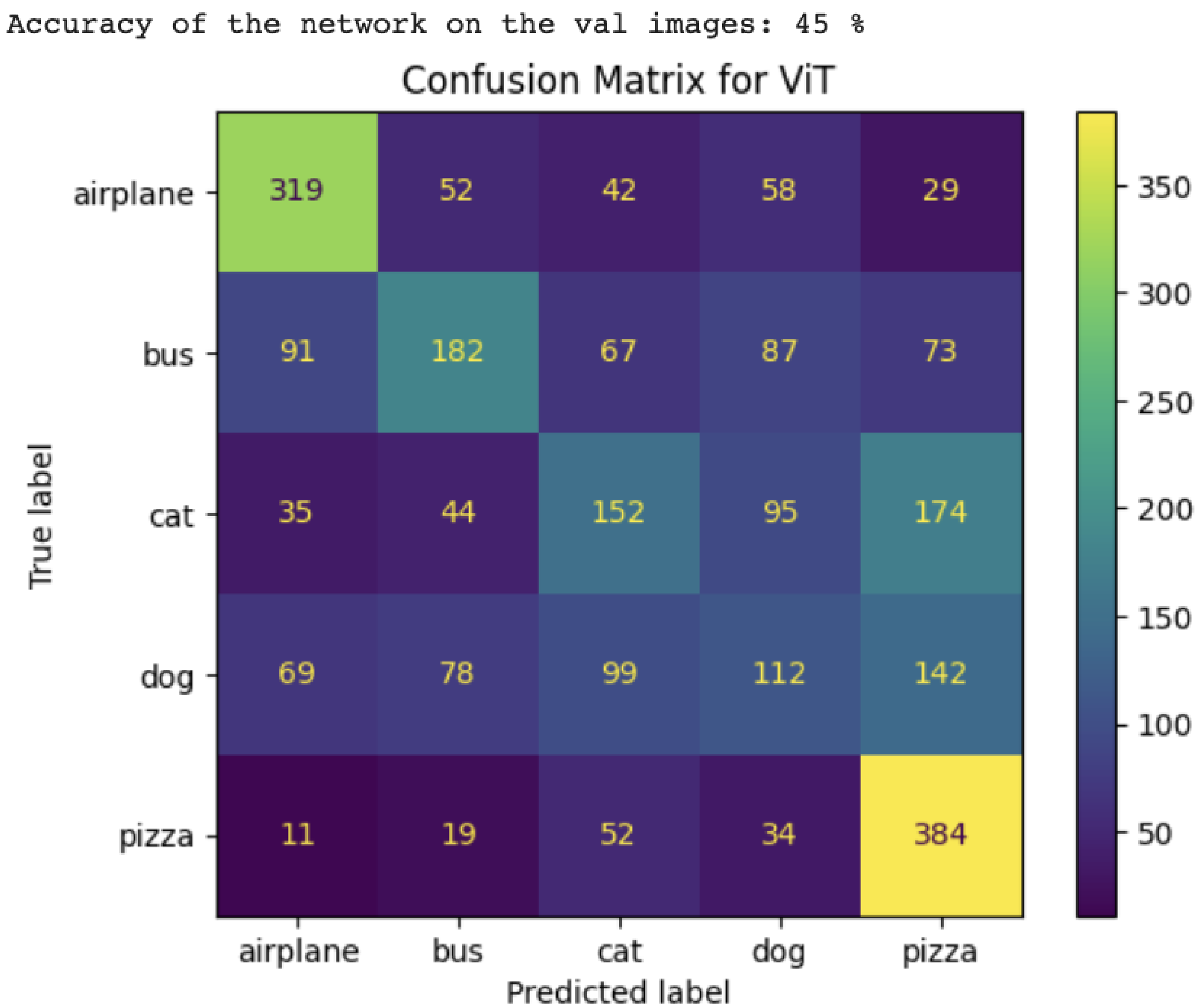
After the entire encoder network is processed, the resulting class embedding is transformed from a 100 dimensional vector to a 5 dimensional vector representing the class probabilities. This is achieved using the nn.Linear() function. During training, the resulting 5 dimensional vector is used in the Cross Entropy loss computation in order to optimize the network parameters. For testing, the maximum argument of the resulting 5 dimensional vector is used as the class prediction from the network.

Note that unlike sequence to sequence predictions, the ViT only has an encoder network. The decoder is not needed because we are transforming the original data to the same shape - we are just doing a classification problem.

Part 3: Results

Results on ViT

After training the network, the following confusion matrix is obtained:

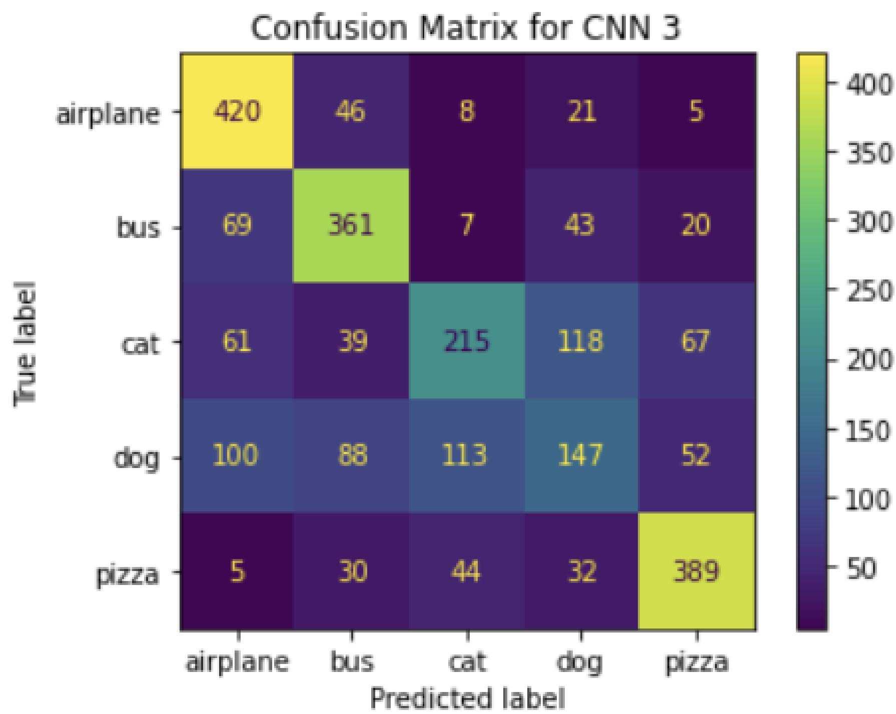


This is slightly worse performance than the CNN based network we did in Lab 4 (45% overall accuracy vs 61% overall accuracy). The reason is likely due to the large increase in parameter size while keeping the training data the same size, which is resulting in overfitting the dataset and reducing generalizability during test time.

Results on CNN from Lab 4:

Below is the Confusion Matrix for the best performing CNN in Lab 4:

```
Accuracy of the network on the val images: 61 %
```



Part 4: Source Code

Source code - Master Encoder Network (mostly from ViTHelper.py)

```python
class MasterEncoder(nn.Module):
    def __init__(self, max_seq_length, embedding_size, how_many_basic_encoders, num_atten_heads):
        super().__init__()
        self.class_embedding = nn.Parameter(torch.rand(size = (1, 1, embedding_size))) #create class embedding.
        #this class_embedding will be the first row of the embedding matrix, where axis 0 is each patch embedding
        #Note that the first 1 in the size parameter above corresponds to the batch size.
        self.pos_embedding = nn.Parameter(torch.rand(size = (1, max_seq_length, embedding_size)))
        self.patch_generator = PatchEmbed()
        self.max_seq_length = max_seq_length
        self.basic_encoder_arr = nn.ModuleList([BasicEncoder(
            max_seq_length, embedding_size, num_atten_heads) for _ in range(how_many_basic_encoders)])  # (A)
        self.mlp_head = nn.Linear(embedding_size, 5)

    def forward(self, img_patch):
        #out_tensor = sentence_tensor
        img_embedding = self.patch_generator(img_patch) #pass in img_patch, which has convo2d applied to it and
        #results in the img embedding.

        img_embedding = torch.cat(tensors = (self.class_embedding, img_embedding), dim = 1)
        img_embedding = img_embedding + self.pos_embedding
        for i in range(len(self.basic_encoder_arr)):  # (B)
            img_embedding = self.basic_encoder_arr[i](img_embedding)


        img_embedding = self.mlp_head(img_embedding[0,0,:]) #use the class embedding vector to do the prediction.
        #here, we take the 100-dimension embedding to 5d, so we can get log probabilities and then do NLLoss
        # all using CE loss in the training loop.
        return img_embedding


class BasicEncoder(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super().__init__()
        self.max_seq_length = max_seq_length
        self.embedding_size = embedding_size
        self.qkv_size = self.embedding_size // num_atten_heads
        self.num_atten_heads = num_atten_heads
        self.self_attention_layer = SelfAttention(
            max_seq_length, embedding_size, num_atten_heads)  # (A)
        self.norm1 = nn.LayerNorm(self.embedding_size)  # (C)
        self.W1 = nn.Linear(self.max_seq_length * self.embedding_size,
                            self.max_seq_length * 2 * self.embedding_size)
        self.W2 = nn.Linear(self.max_seq_length * 2 * self.embedding_size,
                            self.max_seq_length * self.embedding_size)
        self.norm2 = nn.LayerNorm(self.embedding_size)  # (E)

    def forward(self, sentence_tensor):
        input_for_self_atten = sentence_tensor.float()
        normed_input_self_atten = self.norm1(input_for_self_atten)
        output_self_atten = self.self_attention_layer(
            normed_input_self_atten).to(device)  # (F)
        input_for_FFN = output_self_atten + input_for_self_atten
        normed_input_FFN = self.norm2(input_for_FFN)  # (I)
        basic_encoder_out = nn.ReLU()(
            self.W1(normed_input_FFN.view(sentence_tensor.shape[0], -1)))  # (K)
        basic_encoder_out = self.W2(basic_encoder_out)  # (L)
        basic_encoder_out = basic_encoder_out.view(
            sentence_tensor.shape[0], self.max_seq_length, self.embedding_size)
        basic_encoder_out = basic_encoder_out + input_for_FFN
        return basic_encoder_out

```

Attention Logic - from ViTHelper.py

```python
class SelfAttention(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super().__init__()
        self.max_seq_length = max_seq_length
        self.embedding_size = embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads
        self.attention_heads_arr = nn.ModuleList([AttentionHead(self.max_seq_length,
                                                        self.qkv_size) for _ in range(num_atten_heads)])  # (A)

    def forward(self, sentence_tensor):  # (B)
        concat_out_from_atten_heads = torch.zeros(sentence_tensor.shape[0], self.max_seq_length,
                                          self.num_atten_heads * self.qkv_size).float()
        for i in range(self.num_atten_heads):  # (C)
            sentence_tensor_portion = sentence_tensor[:,
                                              :, i * self.qkv_size: (i+1) * self.qkv_size]
            concat_out_from_atten_heads[:, :, i * self.qkv_size: (i+1) * self.qkv_size] =          \
                self.attention_heads_arr[i](sentence_tensor_portion)  # (D)
        return concat_out_from_atten_heads




class AttentionHead(nn.Module):
    def __init__(self, max_seq_length, qkv_size):
        super().__init__()
        self.qkv_size = qkv_size
        self.max_seq_length = max_seq_length
        self.WQ = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)  # (B)
        self.WK = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)  # (C)
        self.WV = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)  # (D)
        self.softmax = nn.Softmax(dim=1)  # (E)

    def forward(self, sentence_portion):  # (F)
        Q = self.WQ(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)  # (G)
        K = self.WK(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)  # (H)
        V = self.WV(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)  # (I)
        Q = Q.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)  # (J)
        K = K.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)  # (K)
        V = V.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)  # (L)
        A = K.transpose(2, 1)  # (M)
        QK_dot_prod = Q @ A  # (N)
        rowwise_softmax_normalizations = self.softmax(QK_dot_prod)  # (O)
        Z = rowwise_softmax_normalizations @ V
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(device)  # (S)
        Z = coeff * Z  # (T)
        return Z
```

Dataset Class (which is later wrapped in the Dataloader class):

```
### Create data_loader ###
root_train = 'train/'
root_val = 'val/'
catNms=['airplane','bus','cat', 'dog', 'pizza']

class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root, catNms):
        super(MyDataset).__init__()
        self.root = {} #dictionary for main directory which holds all the images of a category
        self.filenames = {} #dictionary for filenames of a given category
        for cat in catNms:
            self.root[cat] = root + cat + '/'
        for cat in catNms:
        #create list of image files in each category that can be opened by __getitem__
            self.filenames[cat] = os.listdir(self.root[cat])

        self.rand_max = len(os.listdir(self.root[catNms[0]])) - 1 #number of files in directory

        self.mapping = {0 : 'airplane',
                        1: 'bus',
                        2: 'cat',
                        3: 'dog',
                        4: 'pizza'} #makes it easy to convert between index and name of a category.

        self.one_hot_encoding = {0: torch.tensor(np.array([1, 0, 0, 0, 0])),
                                1: torch.tensor(np.array([0, 1, 0, 0, 0])),
                                2: torch.tensor(np.array([0, 0, 1, 0, 0])),
                                3: torch.tensor(np.array([0, 0, 0, 1, 0])),
                                4: torch.tensor(np.array([0, 0, 0, 0, 1]))} #one hot encode each category.


        self.to_Tensor_and_Norm = tvt.Compose([tvt.ToTensor(),tvt.Resize((64,64)) ,
                                        tvt.Normalize([0], [1]) ,  tvt.ColorJitter(0.75, 0.75) ,
                                        tvt.RandomHorizontalFlip( p = 0.75),
                                        tvt.RandomRotation(degrees = 45)]) #normalize and resize in case the resize op
#         wasn't done. Note that resizing here may not have any impact as the resizing was done previously.


    def __len__(self):
        count = 0
        for cat in catNms:
            temp_num = os.listdir(self.root[cat])
            count = count + len(temp_num)
        return count #. Will be 2500 if the root=val/ and 7500 if root=train/

```

```python
    def __len__(self):
        count = 0
        for cat in catNms:
            temp_num = os.listdir(self.root[cat])
            count = count + len(temp_num)
        return count #. Will be 2500 if the root=val/ and 7500 if root=train/

    def __getitem__(self, index):
        file_index = index % self.rand_max + 1
        class_index = index % 5

        img_file = self.filenames[self.mapping[class_index]]

        try:
            item = Image.open(self.root[self.mapping[class_index]] + img_file[file_index])
        except IndexError: #for debugging
            print('these are the indices for the line above when shape is correct', class_index , file_index)

        np_img = np.array(item)
        shape = np_img.shape
        while shape != (64, 64 ,3): #handle if the image from COCO is grayscale.
            #print('found a grayscale image, fetching an RGB!')
            another_rand = random.randint(0,self.rand_max)  #generate another rand num
            #print('another_rand is', another_rand)
            try:
                item = Image.open(self.root[self.mapping[class_index]] + img_file[another_rand])
            except IndexError: #for debugging
                print('these are the indices for the line above when shape is incorrect', another_rand , class_index)
            np_img = np.array(item)
            shape = np_img.shape

        img = self.to_Tensor_and_Norm(item)
        class_label = self.one_hot_encoding[class_index].type(torch.FloatTensor) #convert to Float
        return img, class_label

```

Training Logic:

```python
epochs = 10
criterion = nn.CrossEntropyLoss()
encoder = MasterEncoder(max_seq_length=17, embedding_size=100, how_many_basic_encoders=4, num_atten_heads=4)
optimizer = torch.optim.Adam(encoder.parameters(), lr = 1e-3, betas = (0.9, 0.99))
loss_running_list_encoder = []
running_loss = 0.0
encoder = encoder.to(device)
for i in range(epochs):
    for n, data in enumerate(my_train_dataloader):
        #Create encoder network:
        #print(n)
        optimizer.zero_grad() #Sets gradients of all model parameters to zero. We want to compute fresh gradients
        #based on the new forward run.
        img, GT = data
        GT = torch.argmax(GT)

        img = img.to(device)
        GT = GT.to(device)

        out = encoder(img)
        loss = criterion(out, GT) #input, then target for arg order

        loss.backward() #compute derivative of loss wrt each gradient.
        optimizer.step() #takes a step on hyperplane based on derivatives
        running_loss += loss.item()
        if (n+1) % 500 == 0:
            print("[epoch: %d, batch: %5d] loss: %3f" % (i + 1, n + 1, running_loss / 500))
            loss_running_list_encoder.append(running_loss/500)
            running_loss = 0.0

```

Testing Logic and Confusion Matrix generation:

```
correct = 0
total = 0
y_pred = []
y_label = []
mapping = { 0: 'airplane',
            1: 'bus',
            2: 'cat',
            3: 'dog',
            4: 'pizza'}


with torch.no_grad():
    for n, data in enumerate(my_val_dataloader):
        images, labels = data
        images = images.to(device)
        outputs = encoder(images)

        predicted = torch.max(outputs.data)
        predicted_class = torch.argmax(outputs.data)

        total += labels.size(0) #add to total count of ground truth images so we can calculate total accuracy
        #print("total images in val set", total)
        for n, i in enumerate(labels):
            temp = np.array(i) #arrays are one hot encoded, we need to convert it into a human readable label for
            #display in the confusion matrix
            label_arg = np.argmax(temp) #get the argument of the one hot encoding
            y_label.append(mapping[label_arg]) #apply the argument to the mapping dictionary above. For example
            # if the argument is 3, then, that corresponds to a label of dog in the mapping dictionary
            t = int(predicted_class) #get integer representation of prediction from network (will
            #be an int from 0 to 4.
            y_pred.append(mapping[t]) #append the predicted output of this label to the prediction list, but,
            #via the mapping dictionary definition so that the y_pred list is human readable.

            if label_arg == t:
                correct = correct + 1 #add to total count of correct predictions so we can calculate total accurac


print('Accuracy of the network on the val images: %d %%' % (
    100 * correct / total))
from sklearn.metrics import confusion_matrix

y_true = y_label
y_pred = y_pred
confusion_matrix=confusion_matrix(y_true, y_pred, labels = [ "airplane", "bus", "cat", "dog", "pizza"])
disp = ConfusionMatrixDisplay(confusion_matrix, display_labels = [ "airplane", "bus", "cat", "dog", "pizza"])
disp.plot()
disp.ax_.set_title("Confusion Matrix for ViT")
plt.show()
```