BME646 and ECE60146: Homework 9

Spring 2023

Arghadip Das

das169@purdue.edu

1. Introduction

Over the past few years, convolutional neural networks (CNNs) have dominated the field of computer vision, achieving state-of-the-art performance on a wide range of visual recognition tasks. However, these networks have several limitations, such as a fixed receptive field and a lack of attention mechanisms for modeling long-range dependencies. To address these issues, a new type of neural network architecture called the Vision Transformer (ViT) has been proposed. The ViT is based on the Transformer architecture originally developed for natural language processing (NLP) and uses self-attention mechanisms to model long-range dependencies between image patches. The ViT has quickly gained attention in the computer vision community due to its impressive performance on various image classification tasks, surpassing the previous state-of-the-art results achieved by CNNs. In addition, the ViT has shown promising results in other computer vision tasks such as object detection, segmentation, and generation.
In this homework, we have several goals. Firstly, we aim to gain a deeper understanding of the multi-headed self-attention mechanism and the transformer architecture. Secondly, we hope to comprehend how the transformer architecture, which was originally developed for language translation, can be readily adapted to process images in the Vision Transformer (ViT). Finally, we are required to implement our own ViT for image classification. By achieving these goals, we will not only learn more about the ViT architecture but also gain valuable insights into the underlying principles of modern neural networks for image processing.

2. Methodology

In this report, I detail the steps taken to complete the homework assignment. To prepare for this homework, we first reviewed Professor Kak's slides on self-attention and gained a better understanding of the self-attention mechanism and its implementation through matrix multiplication. Additionally, we learned how multiple self-attention heads can work in parallel in multi-headed attention to capture inter-word dependencies. We also studied the encoder-decoder structure of a transformer for sequence-to-sequence translation and experimented with `seq2seq_with_transformerFG.py` and `seq2seq_with_transformerPreLN.py`. Finally, we went through the ViT paper to understand the fundamental concepts behind the Vision Transformer (ViT) for image classification. In particular, we closely examined Figure 1, which provides a clear illustration of how an image can be transformed into a sequence of embeddings and processed using a transformer encoder. We paid special attention to how the class token is prepended as a learnable parameter to the input patch embedding sequence, as well as how the same token is taken from the final output sequence to produce the predicted label.

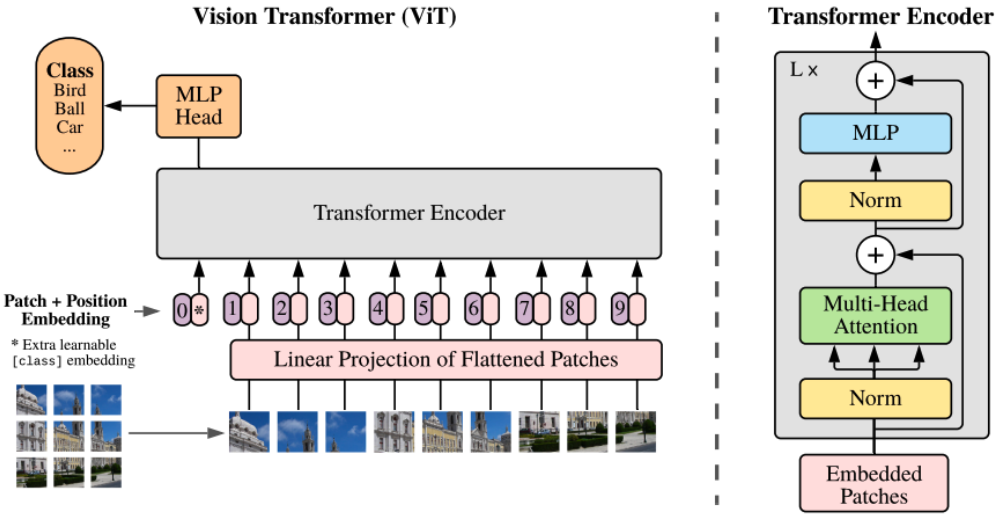

Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification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

To implement our ViT, we utilized the Google Colab platform and followed the instructions provided in the homework manual. We reused the dataset, training, and evaluation scripts from HW4, but replaced the HW4 CNN with our ViT implementation. After training the ViT, we generated a confusion matrix on our test set to evaluate its performance and compared it to CNN-based networks. In addition, we also attempted to implement a multi-headed self-attention mechanism using `torch.einsum`, which we were able to accomplish within just 10 lines of code.

3. Programming Tasks

3.1. *Building Our Own ViT*

Before starting, I reviewed the provided VitHelper.py file. This file includes all the necessary classes for us to build a transformer from scratch quickly. We can find these classes in `DLStudio`'s `Transformers.TransformerPreLN` module. These classes are now standalone so that we can use them directly in our ViT implementation. Specifically, we will use the `MasterEncoder` class as the "Transformer Encoder" block, as shown in Figure 1.

Initially, I utilized a linear layer for converting the patch into an embedding, as illustrated in Figure 1. But then, I incorporated a Conv2D based embedding which turned out to be significantly faster than the linear layer-based conversion. This is because we can apply Conv2D directly on the complete image without dividing it into patches first. However, it requires appropriate tensor reshaping to work effectively. We are working with images of size 64x64 and a patch size of 16x16, resulting in a total of 16 patches. To prepare the input sequence for ViT, we *prepend a*

class token to our patch sequence. To account for this, we set the maximum sequence length of the transformer to 17. For the class token, it must be set as learnable parameter, which is implemented using `nn.Parameter`. In contrast to the sinusoid-based position embedding commonly used in language processing, ViT makes use of *learnable position embeddings*. Therefore, besides initializing the class token, I have also defined the position embeddings as learnable parameters. We obtain the *final class predic*tion by *extracting the class token from the output sequence* generated by the Transformer Encoder block. The class token is then passed through a Multilayer Perceptron (MLP) layer to obtain the logits for the 5 classes. Here in this report, we have included the libraries and helper functions provided in `ViTHelper.py` to ensure that the report can be fully understood without referring to any external files. Following this, the source code for `ADVit` is presented and it is sufficiently commented to make it easy to comprehend.

Source Code:

Importing libraries and getting the device:

```
# Importing required libraries
%matplotlib inline
from pycocotools.coco import COCO          # For MS-COCO API
import numpy as np                         # numpy
from PIL import Image                      # For image resizing
import os                                  # Creating folders
import random                              # Shuffle the images
import torch                               # For building net, training and tesing routines
import torchvision.transforms as tvt       # Convert PIL to tensor and augmentation (if needed)
from torch.utils.data import DataLoader    # Parallel processing of data loading
import torch.nn as nn                      # Neural network layers
import torch.nn.functional as F            # Non-liner activations (ReLU)
import skimage.io as io                    # Image loading through coco_url
import matplotlib.pyplot as plt            # Plotting
import seaborn as sns                      # Colormap of confusion matrix
from torchsummary import summary
# Determining device (CPU or CUDA)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

Helper classes and functions:

```
##  This code is from the Transformers co-class of DLStudio:

##           https://engineering.purdue.edu/kak/distDLS/
class MasterEncoder(nn.Module):
    """
    The purpose of the MasterEncoder is to invoke a stack of BasicEncoder instances on a
    source-language sentence tensor. The output of each BasicEncoder is fed as input to the
    next BasicEncoder in the cascade, as illustrated in the loop in Line B below.  The stack
    of BasicEncoder instances is constructed in Line A.
    """
    def __init__(self, max_seq_length, embedding_size, how_many_basic_encoders, num_atten_heads):
        super().__init__()
        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)

    def forward(self, sentence_tensor):
        out_tensor = sentence_tensor
        for i in range(len(self.basic_encoder_arr)):  # (B)
            out_tensor = self.basic_encoder_arr[i](out_tensor)
```

```
        return out_tensor


class BasicEncoder(nn.Module):
    """
    The BasicEncoder in TransformerPreLN consists of a layer of self-attention (SA) followed
    by a purely feed-forward layer (FFN).  The job of the SA layer is for the network to
    figure out what parts of an input sentence are relevant to what other parts of the
    same sentence in the process of learning how to translate a source-language sentence into
    a target-language sentence. The output of SA goes through FFN and the output of FFN becomes
    the output of the BasicEncoder.  To mitigate the problem of vanishing gradients in the PreLN
    transformer design, the input to each of the two components of a BasicEncoder --- SA and
    FFN --- is subject to LayerNorm and a residual connection used that wraps around both the
    LayerNorm and the component as shown in the keystroke diagram in the comment block associated
    with the definition of TranformerPreLN. Deploying a stack of BasicEncoder instances becomes
    easier if the output tensor from a BasicEncoder has the same shape as its input tensor.

    The SelfAttention layer mentioned above consists of a number of AttentionHead instances,
    with each AttentionHead making an independent assessment of what to say about the
    inter-relationships between the different parts of an input sequence. It is the embedding
    axis that is segmented out into disjoint slices for each AttentionHead instance.The
    calling SelfAttention layer concatenates the outputs from all its AttentionHead instances
    and presents the concatenated tensor as its own output.
    """
    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

##################################  Self Attention Code TransformerPreLN
########################################

class SelfAttention(nn.Module):
    """
    As described in the doc section of the BasicEncoder class, in each BasicEncoder you have
    a layer of SelfAttention followed by a Fully-Connected layer.  The SelfAttention layer
    concatenates the outputs from all AttentionHead instances and presents that concatenated
    output as its own output. If the input sentence consists of W words at most and if the
    embedding size is M, the sentence_tensor at the input to forward() in Line B below will
    be of shape [W,M].  This tensor will be fed into each AttentionHead instance constructed
    in Line A.  If K is the size of the output from an AttentionHead instance, the output of each
    such instance will be of shape [W,K].  The SelfAttention instance concatenates A of those
    AttentionHead outputs and returns a tensor of shape [W,K*A] to the BasicEncoder.  This
    concatenation is carried in the loop in Lines C and D below.
    """
    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)
        # batch_size, max_seq_length, embedding_size
        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):
    """
    An AttentionHead (AH) instance does its job by first matrix-multiplying the embedding
    vector FOR EACH WORD in an input sentence by the following three matrices:

    -- a matrix Wq of size PxM to produce the query Vector q where P is the desired size
        for the matrix-vector product and M the size of the embedding.  For each word, this
        will yield a query vector of size P elements at each word position in the input
        sentence.

    -- a matrix Wk, also of size PxM, to produce a key vector k of size P elements at
        each word position

    -- a matrix Wv, also of size PxM, to produce a value vector v of size P elements at
        each word position.
    """
    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
```

Vision Transformer (`ADVit`) Implementation:

```
class ADVit(nn.Module):
  """
  The ADVit class is a PyTorch module that implements an attention-based deep learning
architecture called Vision Transformer (ViT)
  for image classification. It takes as input an image tensor x of size (batch_size,
num_channels, image_height, image_width).
  The class constructor initializes the model's hyperparameters including image_size, patch_size,
num_atten_heads, embedding_size,
  how_many_basic_encoders, mlp_dim, num_classes, and patch_to_embed_method.
  The model converts the image tensor into a sequence of patch embeddings using either a linear
layer or a convolutional layer,
  based on the patch_to_embed_method argument. If the linear method is used, the input tensor x
is first unfolded into patches
  of size (batch_size, num_channels, num_patches_vertically, num_patches_horizontally,
patch_size, patch_size) using the unfold
  method of PyTorch. Then, the patches are reshaped into a tensor of size (batch_size,
num_patches, patch_dim), where patch_dim
  is the size of each patch embedding. If the convolutional method is used, the input tensor x is
passed through a convolutional
  layer with embedding_size output channels and a kernel size and stride equal to the patch_size.
The resulting tensor is then
  reshaped into a tensor of size (batch_size, num_patches, embedding_size).
  The patch embeddings are then augmented with a learnable class token and learnable positional
embeddings,
  resulting in a tensor of size (batch_size, max_seq_length, embedding_size), where
max_seq_length is the number of patches plus one.
  The resulting tensor is then fed into a stack of transformer encoders implemented in the
MasterEncoder class.
  Finally, the class token output of the last transformer encoder is extracted and passed through
a multi-layer perceptron (MLP) consisting of three
  fully connected layers with embedding_size, mlp_dim, and num_classes output channels,
respectively. The output of the MLP is a
  tensor of size (batch_size, num_classes), which represents the model's predicted probabilities
for each of the input images.
  """
  def __init__(self, image_size, patch_size, num_atten_heads, embedding_size,
how_many_basic_encoders, mlp_dim, num_classes=5, patch_to_embed_method='Conv'):
    super().__init__()
    self.image_size = image_size                                # Image height and width (if 64,
image is 3x64x64)
    self.patch_size = patch_size                                # Size of each patch (if 16, patch
is 3x16x16)
    self.embedding_size = embedding_size                        # Size of embedding for each patch
(chosen 128)
    self.how_many_basic_encoders = how_many_basic_encoders      # Number of basic encoders in
cascade (chosen 2)
    self.mlp_dim = mlp_dim                                      # dimension of the first FC layer
output (chosen 96)
    self.num_classes = num_classes                              # Number of classes (for image
classification application, here 5)
    self.num_atten_heads = num_atten_heads                      # Number of attention heads (chosen
as 8), each attention
                                                                # head will work on the embedding
of size (128/8 = 16)
    self.patch_to_embed_method = patch_to_embed_method          # The method to convert a patch to
embedding (Linear or Conv2d)

    # calculate number of patches (Assuming square images)
    self.num_patches = (image_size // patch_size) ** 2          # Number of patches = (64/16)^2 =
16
    self.max_seq_length = self.num_patches + 1                  # Length of the sequence = Number
of patches + 1 (for prepended class token)
    self.patch_dim = 3 * patch_size ** 2                        # Patch dimension (3*16*16 = 768),
assuming RGB images with 3 channels

    if patch_to_embed_method == 'Lin':
      # patch embedding layer (using nn.Linear)
      self.patch_embeddings = nn.Linear(self.patch_dim, embedding_size)
```

```
    elif patch_to_embed_method == 'Conv':
      # patch embedding layer (using nn.Conv2d)
      # using a conv layer instead of a linear one -> performance gains
      # kernel_size = stride = patch_size, So, basically it is operating on each patch separately
      self.patch_embeddings = nn.Conv2d(3, embedding_size, kernel_size=patch_size,
stride=patch_size)

    # learnable positional embeddings (nn.Parameter)
    self.positional_embeddings = nn.Parameter(torch.zeros(1, self.num_patches + 1,
embedding_size))

    # Encoder layers
    self.encoder = MasterEncoder(self.max_seq_length, embedding_size, how_many_basic_encoders,
num_atten_heads)

    # class token (learnable) (nn.Parameter)
    self.class_token = nn.Parameter(torch.zeros(1, 1, embedding_size))

    # class prediction head (Multi-Layer Perceptron, i.e., MLP)
    self.fc = nn.Sequential(
                            nn.Linear(embedding_size, self.mlp_dim),
                            nn.Linear(self.mlp_dim, 64),
                            nn.Linear(64, num_classes),
    )

  # Forward pass through ViT
  def forward(self, x):                          # Example sizes are given considering the shape
of x as (1,3,64,64)
    if self.patch_to_embed_method == 'Lin':      # If embeddings for patches are created using
linear layers
      # extract patches
      x = x.unfold(2, self.patch_size, self.patch_size).unfold(3, self.patch_size,
self.patch_size) # (1, 3, 4, 4, 16, 16)
      x = x.contiguous().view(x.size(0), -1, self.patch_dim)  # (1, 16, 768)

      # patch embeddings
      x = self.patch_embeddings(x)               # (1, 16, 768)
    elif self.patch_to_embed_method == 'Conv':
      x = self.patch_embeddings(x)               # (1, 128, 4, 4)
      x = x.view(x.size(0), -1, x.size(1))       # (1, 16, 128)

    # add class token
    class_token = self.class_token.expand(x.size(0), -1, -1)
    x = torch.cat((class_token, x), dim=1)       # (1, 17, 128)

    # add positional embeddings
    x = x + self.positional_embeddings           # (1, 17, 128)

    # transformer layers
    x = self.encoder(x)                          # (1, 17, 128)

    # extract class token output
    class_output = x[:, 0]                       # (1, 128)

    # class prediction head
    output = self.fc(class_output)               # (1, 5)
    return output
```

`ADVit` takes in an image and outputs a prediction of its class. The input image is assumed to be square with a specified image size and three channels for RGB. The image is divided into non-overlapping patches of a specified patch size, and each patch is embedded into a vector of a specified embedding size using *either a linear layer or a convolutional layer.* The module also learns learnable positional embeddings of the same size as the patch embeddings, and it includes an encoder made up of a specified number of basic encoders that each have a specified number of attention heads. Additionally, a learnable class token is added to the input embeddings, and the output of the class token is fed through an MLP to produce a prediction of the input image's class.

In the `__init__` function, the relevant parameters are set, and the various components of the module are defined. The forward function takes in an input image and performs the necessary operations to convert the image into a sequence of embeddings that can be fed into the encoder. If linear patch embeddings are used, the input image is divided into patches using the unfold function, and each patch is embedded using the linear layer. If convolutional patch embeddings are used, the input image is passed through a convolutional layer, resulting in a tensor of size `(batch_size, embedding_size, num_patches, num_patches)`, and the tensor is reshaped to `(batch_size, num_patches, embedding_size)`. In both cases, the class token is added to the sequence of embeddings, and the positional embeddings are added to the embeddings. The resulting sequence is fed through the encoder, and the output of the class token is passed through an MLP to produce a prediction of the input image's class.

We get the following details of the `ADVit`. In the following code, we can see the ***hyperparameters used in `ADVit`.*** *The mentioned* ***shapes of the tensors and variables*** *in comments in the earlier `ADVit` code are based on these hyperparameters.*

```
model = ADVit(image_size=64, \
              patch_size=16, \
              num_atten_heads=8,
              embedding_size=128, \
              how_many_basic_encoders=2, \
              mlp_dim=96, \
              num_classes=5,\
              patch_to_embed_method='Conv').to(device)

number_of_learnable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
num_layers = len(list(model.parameters()))
print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" % number_of_learnable_params)
image_size = 64
input_size = (3,image_size,image_size)
summary(model, input_size)
```

```
The number of layers in the model: 122

The number of learnable parameters in the model: 41577829
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 128, 4, 4]          98,432
         LayerNorm-2            [-1, 17, 128]             256
            Linear-3               [-1, 272]          74,256
            Linear-4               [-1, 272]          74,256
            Linear-5               [-1, 272]          74,256
           Softmax-6            [-1, 17, 17]               0
    AttentionHead-7            [-1, 17, 16]               0
            Linear-8               [-1, 272]          74,256
            Linear-9               [-1, 272]          74,256
           Linear-10               [-1, 272]          74,256
          Softmax-11            [-1, 17, 17]               0
   AttentionHead-12            [-1, 17, 16]               0
           Linear-13               [-1, 272]          74,256
           Linear-14               [-1, 272]          74,256
           Linear-15               [-1, 272]          74,256
          Softmax-16            [-1, 17, 17]               0
   AttentionHead-17            [-1, 17, 16]               0
           Linear-18               [-1, 272]          74,256
           Linear-19               [-1, 272]          74,256
           Linear-20               [-1, 272]          74,256
          Softmax-21            [-1, 17, 17]               0
```

```
   AttentionHead-22              [-1, 17, 16]                   0
          Linear-23                 [-1, 272]             74,256
          Linear-24                 [-1, 272]             74,256
          Linear-25                 [-1, 272]             74,256
         Softmax-26              [-1, 17, 17]                   0
   AttentionHead-27              [-1, 17, 16]                   0
          Linear-28                 [-1, 272]             74,256
          Linear-29                 [-1, 272]             74,256
          Linear-30                 [-1, 272]             74,256
         Softmax-31              [-1, 17, 17]                   0
   AttentionHead-32              [-1, 17, 16]                   0
          Linear-33                 [-1, 272]             74,256
          Linear-34                 [-1, 272]             74,256
          Linear-35                 [-1, 272]             74,256
         Softmax-36              [-1, 17, 17]                   0
   AttentionHead-37              [-1, 17, 16]                   0
          Linear-38                 [-1, 272]             74,256
          Linear-39                 [-1, 272]             74,256
          Linear-40                 [-1, 272]             74,256
         Softmax-41              [-1, 17, 17]                   0
   AttentionHead-42              [-1, 17, 16]                   0
   SelfAttention-43             [-1, 17, 128]                   0
       LayerNorm-44             [-1, 17, 128]                 256
          Linear-45                [-1, 4352]       9,474,304
          Linear-46                [-1, 2176]       9,472,128
    BasicEncoder-47             [-1, 17, 128]                   0
       LayerNorm-48             [-1, 17, 128]                 256
          Linear-49                 [-1, 272]             74,256
          Linear-50                 [-1, 272]             74,256
          Linear-51                 [-1, 272]             74,256
         Softmax-52              [-1, 17, 17]                   0
   AttentionHead-53              [-1, 17, 16]                   0
          Linear-54                 [-1, 272]             74,256
          Linear-55                 [-1, 272]             74,256
          Linear-56                 [-1, 272]             74,256
         Softmax-57              [-1, 17, 17]                   0
   AttentionHead-58              [-1, 17, 16]                   0
          Linear-59                 [-1, 272]             74,256
          Linear-60                 [-1, 272]             74,256
          Linear-61                 [-1, 272]             74,256
         Softmax-62              [-1, 17, 17]                   0
   AttentionHead-63              [-1, 17, 16]                   0
          Linear-64                 [-1, 272]             74,256
          Linear-65                 [-1, 272]             74,256
          Linear-66                 [-1, 272]             74,256
         Softmax-67              [-1, 17, 17]                   0
   AttentionHead-68              [-1, 17, 16]                   0
          Linear-69                 [-1, 272]             74,256
          Linear-70                 [-1, 272]             74,256
          Linear-71                 [-1, 272]             74,256
         Softmax-72              [-1, 17, 17]                   0
   AttentionHead-73              [-1, 17, 16]                   0
          Linear-74                 [-1, 272]             74,256
          Linear-75                 [-1, 272]             74,256
          Linear-76                 [-1, 272]             74,256
         Softmax-77              [-1, 17, 17]                   0
   AttentionHead-78              [-1, 17, 16]                   0
          Linear-79                 [-1, 272]             74,256
          Linear-80                 [-1, 272]             74,256
          Linear-81                 [-1, 272]             74,256
         Softmax-82              [-1, 17, 17]                   0
   AttentionHead-83              [-1, 17, 16]                   0
          Linear-84                 [-1, 272]             74,256
          Linear-85                 [-1, 272]             74,256
          Linear-86                 [-1, 272]             74,256
         Softmax-87              [-1, 17, 17]                   0
   AttentionHead-88              [-1, 17, 16]                   0
   SelfAttention-89             [-1, 17, 128]                   0
       LayerNorm-90             [-1, 17, 128]                 256
          Linear-91                [-1, 4352]       9,474,304
          Linear-92                [-1, 2176]       9,472,128
```

```
     BasicEncoder-93              [-1, 17, 128]               0
    MasterEncoder-94              [-1, 17, 128]               0
          Linear-95                   [-1, 96]          12,384
          Linear-96                   [-1, 64]           6,208
          Linear-97                    [-1, 5]             325
================================================================
Total params: 41,575,525
Trainable params: 41,575,525
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 0.43
Params size (MB): 158.60
Estimated Total Size (MB): 159.08
----------------------------------------------------------------
```

3.2. *Image Classification with `ADVit`*

To accomplish this task, we will reuse the training and evaluation scripts we developed in HW4. Instead of using the HW4 CNN, we will replace it with our ViT model. We will also use the COCO-based dataset that we created for HW4, which contains 64 × 64 images from five classes. *To ensure that our report is self-contained and can be used to reproduce our results, we have included the entire process of creating the dataset, defining the dataloader class, and the training and testing routines from HW4. We will present each step in detail below.*

Creating Our Own Image Classification Dataset

In this task we need to create our own image classification dataset by taking images from the MS-COCO dataset. For that, I took the help of python version of the COCO API. We are using 2014 Train images. I downloaded all the images from the following link and uploaded it to the Google Drive. I also downloaded the annotation files: 2014 Train/Val annotations. For image classification task, `instances_train2014.json` file is used. The following script is used to read the images from the MS-COCO dataset, resize it to 64x64 images using PIL module and save the images to another directory. It is made sure that there are no duplicate images.

Source Code:

```
dataDir = '/content/drive/MyDrive/Arghadip/DL/Datasets'              # Directory of annotation and
entire MS-COCO dataset
dataType='train2014'                                                 # We are working with
"train2014" version
annFile='{}/annotations/instances_{}.json'.format(dataDir,dataType) # Name of the annotation file
coco=COCO(annFile)                                                   # initialize COCO api for
instance annotations

classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']                 # We are interested in these
5 classes

for cat in classes:                                                  # Loop over all the classes
  catIds = coco.getCatIds(catNms=[cat])                              # Get class ids
  imgIds = coco.getImgIds(catIds=catIds)                             # Get image ids corresponding
to the class

  # Shuffle images
  indices = list(range(len(imgIds)))
  random.shuffle(indices)
```

```
  # Take first 1500 images for training
  train_dir = os.path.join(dataDir, 'coco', 'train', cat)           # Directory to write training
images
  os.mkdir(train_dir)                                               # Make the directory
  for i in range(1500):
    img = coco.loadImgs(imgIds[indices[i]])[0]                      # Get image details
    I = Image.open(os.path.join(dataDir, dataType, img['file_name'])) # Load as PIL image from
the MS-COCO directory
    I = I.convert('RGB')                                            # Convert to RGB
    im_resized = I.resize((64,64), Image.BOX)                       # Resize to 64 x 64
    im_resized.save(os.path.join(train_dir, cat + '_' + str(i) + '.jpg')) # Save with class name
and sequence number

  # Take next 500 images for testing
  test_dir = os.path.join(dataDir, 'coco', 'test', cat)             # Directory to write test
images
  os.mkdir(test_dir)                                                # Make the directory
  for i in range(1500,2000):
    img = coco.loadImgs(imgIds[indices[i]])[0]                      # Get image details
    I = Image.open(os.path.join(dataDir, dataType, img['file_name'])) # Load as PIL image from
the MS-COCO directory
    I = I.convert('RGB')                                            # Convert to RGB
    im_resized = I.resize((64,64), Image.BOX)                       # Resize to 64 x 64
    im_resized.save(os.path.join(test_dir, cat + '_' + str(i-1500) + '.jpg')) # Save with class
name and sequence number
```

Output Directory Structure:

Figure 2. Dataset directory structure

Once the images are saved in proper directory, the next step is to implement our own dataset class to provide the necessary support to the `torch.utils.data.DataLoader` class. The source code is given below. The dataset class `My_COCO_Dataset` contains the relevant information such as root directory, split (train or test), number of images per class etc. It also performs proper transform to covert PIL images to CNN supported tensor input. I haven't used any data augmentation techniques, but the scope is already present in the given source code. One method named `_get_filenames_and_labels()` is defined to get all the filenames and corresponding labels, so that later it can be used during query (i.e. `__getitem__()`). The `__len__()` method is also overwritten to return the total number of images in the dataset. During training feeding images from different classes in a single batch leads to better training. Therefore, shuffling support is also provided (`shuffle=True`). The labels are integers (0 to 4) for 5 classes.

Source Code:

```
# Custom dataset class definition
class My_COCO_Dataset(torch.utils.data.Dataset):
  def __init__(self, root='/content/drive/MyDrive/Arghadip/DL/Datasets/coco', split='train',
shuffle=False):
    super().__init__()  # Part of the definition is obtained from parent class
    self.split = split                                          # 'train' or 'test'
    self.path = os.path.join(root,self.split)                   # Assign path as per the split
    self.shuffle = shuffle                                      # If shuffling is needed
    self.classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']   # Name of 5 classes
    if self.split == 'train':
      self.num_images_per_class = 1500                          # 1500 images per class for
training
    elif self.split == 'test':
      self.num_images_per_class = 500                           # 500 images per class for
testing
    # Initialize data augmentation transforms , etc.
    # tvt.Compose collates multiple transforms and perform them sequentially
    self.xform = tvt.Compose([
        # # ------------------- Uncomment following transforms for improved performance ---------
----------
        # # ColorJitter deals with altering the color properties of an image by changing its
pixel values.
        # tvt.ColorJitter(brightness=1, contrast=0, saturation=0, hue=0),
        # # Converted into grayscale with probability 0.5 for augmentation.
        # tvt.RandomGrayscale(p=0.5),
        # # Flipped horizontally with probability 0.5
        # tvt.RandomHorizontalFlip(p=0.5),
        # Conversion from PIL to floating point Tensor
        tvt.ToTensor(),
        # Normalize
        tvt.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))
    ])
    self._get_filenames_and_labels()                            # Form a list with filenames and
corresponding labels

  # Internal method to get filenames and labels of all the classes
  def _get_filenames_and_labels(self):
    self.list_of_files_and_labels = []                          # Empty list initialization
    for i in range(len(self.classes)):                          # Start loop for each class
      for j in range(self.num_images_per_class):                # Start loop for images in each
class
        # Append the filenames and labels to the "list_of_files_and_labels"
        self.list_of_files_and_labels.append([os.path.join(self.path, self.classes[i],
self.classes[i] + '_' + str(j) + '.jpg'), i])
    # Shuffling
    if self.shuffle:
      random.shuffle(self.list_of_files_and_labels)

  def __len__ (self):
    # Return the total number of images in the dataset = number of files in the
"list_of_files_and_labels"
    return len(self.list_of_files_and_labels)

  def __getitem__(self, index):
  # Read an image at index and perform augmentations
  # Return the tuple : ( augmented tensor , integer label )
    image = Image.open(self.list_of_files_and_labels[index][0])   # Load image as PIL object
    image = self.xform(image)                                     # Apply transform
    return (image, self.list_of_files_and_labels[index][1])       # Return the image tensor and
label (integer)
```

Training routine:

We use `Adam` optimizer with ***learning rate = 1e-3, $\beta_1 = 0.9$ and $\beta_2 = 0.99$*** for training. It also saves the model parameters in every epoch as a dictionary that is later used for testing using validation dataset. The model is trained for ***50 epochs***.

```
def run_training(net, train_data_loader, learning_rate, num_epochs, print_frequency=100):
  '''
  This is the method used to run training. It returns the training loss as a list.
  '''
  net = net.to(device)                                # Move 'net' to device (either CPU or CUDA)
  criterion = torch.nn.CrossEntropyLoss()             # Loss function
  optimizer = torch.optim.Adam(net.parameters(), lr = learning_rate, betas = (0.9, 0.99))    #
Adam optimizer
  loss_for_plot = []                                  # Empty list to store losses
  for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(train_data_loader):  # Loop for every batch in the train_data_loader
      inputs, labels = data
      inputs = inputs.to(device)                      # Move input batch to the device
      labels = labels.to(device)                      # Move corresponding labels to the device
      optimizer.zero_grad()                           # Clear the stored gradient at nodes
      outputs = net(inputs)                           # Forward pass
      loss = criterion(outputs, labels)               # Loss calculation
      loss.backward()                                 # Gradient calculation
      optimizer.step()                                # Back propagation and update of parameters
      running_loss += loss.item()                     # Update loss
      if (i+1) % print_frequency == 0:                # Print and keep record for every
'print_frequency'
        print("[epoch: %d, batch: %5d] loss: %.3f" \
              % (epoch + 1, i + 1, running_loss / print_frequency))
        loss_for_plot.append(running_loss/print_frequency)
        running_loss = 0.0
    # Save model parameters in every epoch
    torch.save(net.state_dict(), os.path.join('/content/drive/MyDrive/Arghadip/DL/HW9/models',
"ADVit_{}.pth".format(str(learning_rate) + '_' + str(num_epochs))))
  return loss_for_plot                                # Return loss vector
```

Test routine and script to plot confusion matrix:

The test routine is inspired from the routine present in `DLStudio`. The `seaborn` package is useful for the proper display of the confusion matrix.

```
def run_testing(net, test_data_loader, num_classes=5):
  '''
  This function is used for validation/testing. It returns the confusion matrix.
  '''
  confusion_matrix = np.zeros([num_classes, num_classes])   # Initialization of confusion matrix
as ndarray
  with torch.no_grad():                                     # With no gradient calculation at
nodes
    for i, data in enumerate(test_data_loader):             # For every batch in test_data_loader
      inputs, labels = data                                 # Get input images (as tensors) and
labels
      inputs = inputs.to(device)
      labels = labels.to(device)
      outputs = net(inputs)                                 # Forward pass
      _, predicted = torch.max(outputs, 1)                  # Predicted class (class for which
the output is max)
      for label,prediction in zip(labels,predicted):        # Confusion matrix formulation (Taken
from DLStudio)
        confusion_matrix[label][prediction] += 1
  return confusion_matrix                                   # Return confusion matrix
```

```
def plot_confusion_matrix(confusion_matrix, classes, title='Confusion Matrix'):
  '''
  This function is defined to plot the confusion matrix with proper colormap and labels.
  '''
  labels = []                                                 # List of empty labels for all the
positions in matrix
  for row in range(confusion_matrix.shape[0]):                # Loop for every row
    rows = []
    for col in range(confusion_matrix.shape[1]):              # Loop for every column
      pred_count = confusion_matrix[row][col]                 # Counts for predicted classes for
this particular class
      percentage = "%0.2f%%" % (pred_count*100/(np.sum(confusion_matrix[:][col])))  # Percentage
      box_label = str(pred_count) + '\n' + str(percentage)  # Label for the box with both count
and percentage
      rows.append(box_label)                                  # Append to the list
    labels.append(rows)                                       # Append the row to the labels
  labels = np.asarray(labels)                                 # Form numpy array from list
  test_accuracy = np.trace(confusion_matrix) * 100 / np.sum(confusion_matrix) # Overall test
accuracy in percentage
  # Plot of confusion matrix
  # plt.figure(figsize=(5,6))
  plt.figure()
  sns.heatmap(confusion_matrix, annot=labels, fmt="", cmap="Blues", cbar=True,
xticklabels=classes, yticklabels=classes)
  plt.title(title)
  plt.xlabel('Predicted label' + '\n\nAccuracy = %0.3f'% test_accuracy)
  plt.ylabel('True label')
  plt.savefig(os.path.join('/content/drive/MyDrive/Arghadip/DL/HW9/confusion_matrices', title +
'.jpg'), dpi=300)  # save
  plt.show()
```

Main code:

```
# Main code
model = ADVit(image_size=64, \
              patch_size=16, \
              num_atten_heads=8,
              embedding_size=128, \
              how_many_basic_encoders=2, \
              mlp_dim=96, \
              num_classes=5,\
              patch_to_embed_method='Conv').to(device)

# Dataset
batch_size = 64          # Batch size
num_workers = 2          # Number of parallel threads for data loading
train_dataset = My_COCO_Dataset(split='train', shuffle=True)     # Training dataset
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
num_workers=num_workers) # Loader
test_dataset = My_COCO_Dataset(split='test', shuffle=False)      # Testing/Validation dataset
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,
num_workers=num_workers)  # Loader

# Run training
learning_rate = 1e-3     # Learning rate
num_epochs = 50          # Number of training epochs
print_frequency = 10     # Frequency of logging loss
loss = run_training(model, train_dataloader, learning_rate, num_epochs=num_epochs,
print_frequency=print_frequency)     # Loss

label = model.__class__.__name__ + '_' + str(learning_rate) + '_' + str(num_epochs)
# Plot training loss
fig, axes = plt.subplots(ncols=1, figsize = (8,4))
axes.plot(loss, label=label)
axes.set_title('Training loss vs. batches')
axes.set_xlabel('Batches in hundreds, batch size = ' + str(batch_size))
axes.set_ylabel('Training loss')
fig.legend()
```

```
fig.tight_layout()
plt.show();
fig.savefig(os.path.join("/content/drive/MyDrive/Arghadip/DL/HW9/training_losses", label +
'.jpg'),dpi=300)             # Save
# # Validation and Confusion matrix plot
# Network shell
net = ADVit(image_size=64, \
              patch_size=16, \
              num_atten_heads=8,
              embedding_size=128, \
              how_many_basic_encoders=2, \
              mlp_dim=96, \
              num_classes=5,\
              patch_to_embed_method='Conv').to(device)
modelfile = os.path.join('/content/drive/MyDrive/Arghadip/DL/HW9/models', label + '.pth') # Param
dictionary path
if device.type == 'cpu':
  net.load_state_dict(torch.load(modelfile, map_location=torch.device('cpu')))  # Load in CPU
else:
  net.load_state_dict(torch.load(modelfile))                                    # Load in GPU
net.to(device)                                                                  # Mode the model
to 'device'
net.eval()                                                                      # Setting models
to eval mode
confusion_matrix = run_testing(net, test_dataloader, num_classes=5)             # Testing
plot_confusion_matrix(confusion_matrix, test_dataset.classes, title='Confusion Matrix (' + label
+ ')')  # Plot confusion matrix
```

Training Loss over Training Iterations:

The following plots show how training progresses along the epochs.

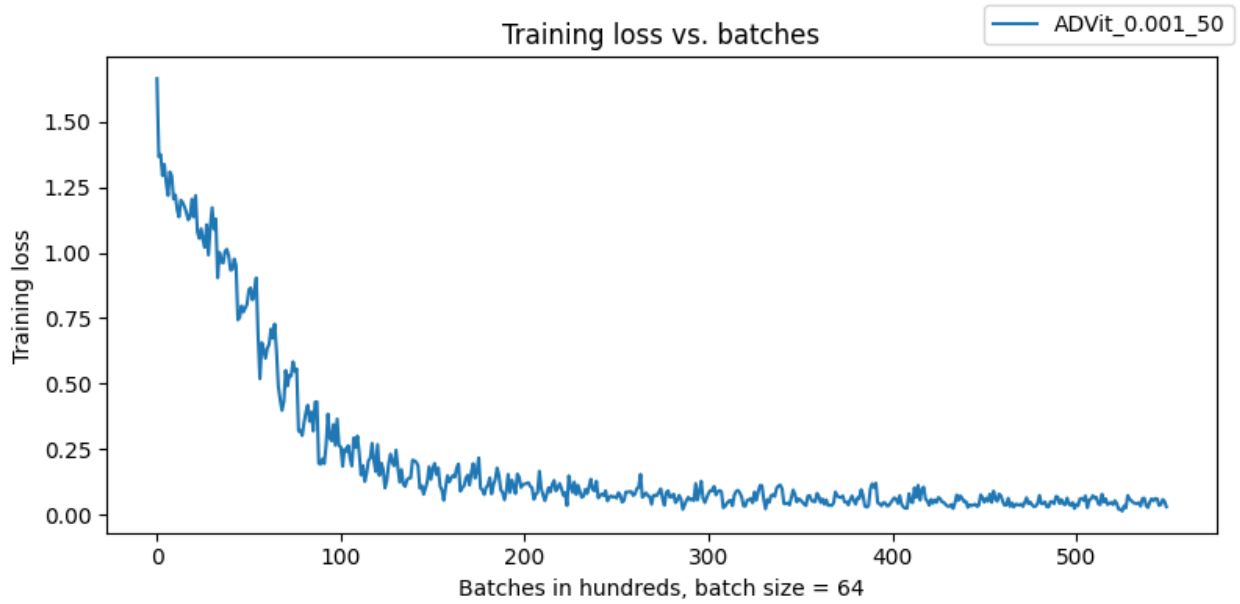

Fig.3. Training loss vs. iterations for `ADVit`

Confusion matrix and overall accuracy:

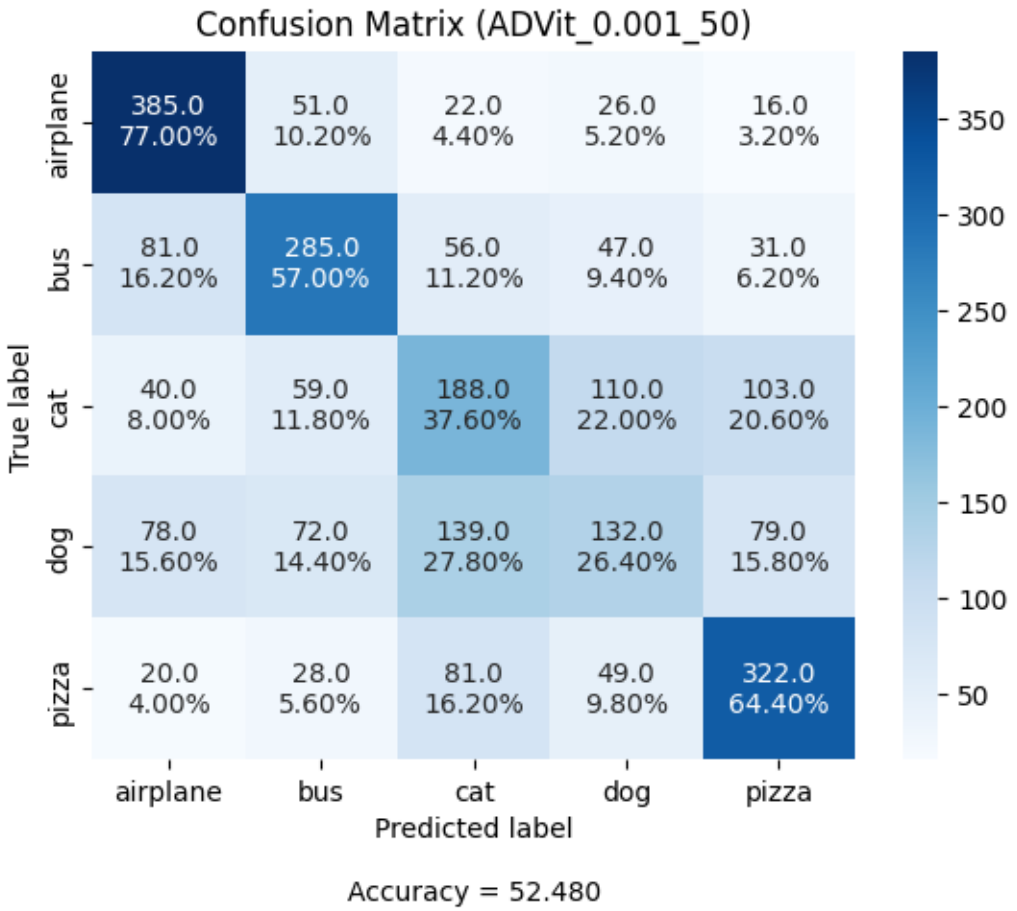

Fig.6. Confusion matrix for `ADVit` over HW4-COCO testset

Overall accuracy: 52.48%

Comparison with HW4 CNN based network:

For better comparison, I present the confusion matrices of 3 different CNN tasks from HW4. These networks were trained with the same hyperparameters (Adam optimizer with learning rate 1e-3, $\beta_1 = 0.9$ and $\beta_2 = 0.99$) for 50 epochs with batch size of 8.

P.T.O.

Confusion Matrix for HW4 Task 1:

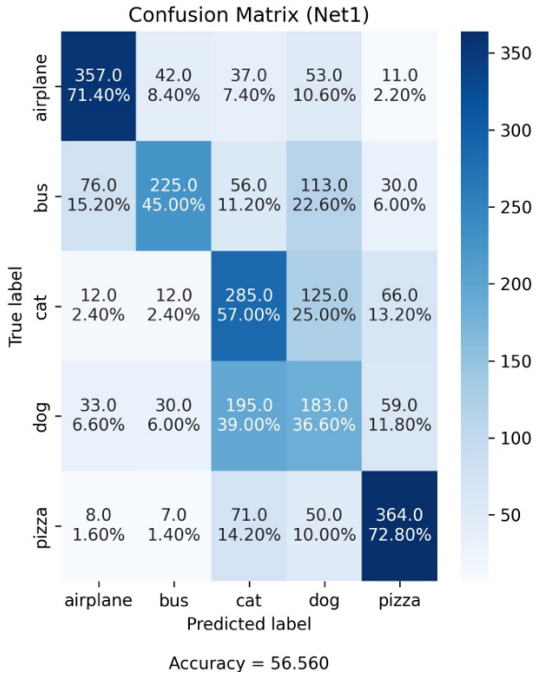

Fig.4. Confusion matrix for HW4 Net1 (Overall test accuracy = 56.56%)

Confusion Matrix for HW4 Task 2:

Fig.5. Confusion matrix for HW4 Net2 (Overall test accuracy = 57.56%)

Confusion Matrix for HW4 Task 3:

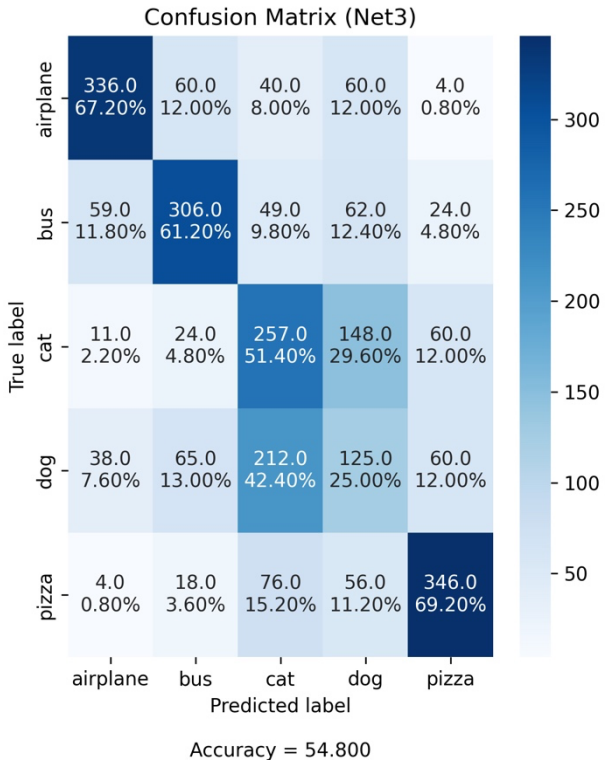

Fig.6. Confusion matrix for HW4 Net3 (Overall test accuracy = 54.80%)

The transformer model (`ADVit`) performs worse than the CNN-based models, with only 52.48% accuracy on the test set. In comparison, the CNNs had the following accuracy in decreasing order: `Net2` (57.56%) > `Net1` (56.56%) > `Net3` (54.80%). ***This is because `ADVit` is overfitting the small training set,*** which consists of only 7500 images of size 64x64. The total number of training points is 7500x(3x64x64) = 92160000, while the number of parameters in the transformer model (41577829) is comparable with the size of the training set. Although I have tried to reduce the number of parameters by adjusting the hyperparameters, the model still overfits. To address this issue, regularization, dropout, and early termination methods can be used, but they are not within the scope of this homework. On the other hand, the low accuracies of the CNN models in HW4 were due to underfitting caused by their small model orders.

3.3. *Extra Credit: Multi-headed self-attention using `torch.einsum`*

The implementation code with detailed comments is given below.

```
class SelfAttention_einsum(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super().__init__()
        self.max_seq_length = max_seq_length                        # Sequence length (here 17)
        self.embedding_size = embedding_size                        # Emdedding size (here 128)
        self.num_atten_heads = num_atten_heads                      # Number of attention heads
(here 8)
        self.qkv_size = self.embedding_size // num_atten_heads      # size of Q, K and V (here
128/8 = 16)
        self.attention_heads_arr = nn.ModuleList([AttentionHead_einsum(self.max_seq_length,
                                                                       self.qkv_size) for _ in
range(num_atten_heads)]) # Forming multiple attention heads

    def forward(self, sentence_tensor):
        # batch_size, max_seq_length, embedding_size i.e. (1, 17, 128)
        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):
            sentence_tensor_portion = sentence_tensor[:,
                                                      :, i * self.qkv_size: (i+1) *
self.qkv_size]  # (1, 17, 16)
            concat_out_from_atten_heads[:, :, i * self.qkv_size: (i+1) * self.qkv_size] =
\
                self.attention_heads_arr[i](sentence_tensor_portion)    # Concatenate attention
head outputs
        return concat_out_from_atten_heads


class AttentionHead_einsum(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)
        self.WK = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)
        self.WV = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, sentence_portion):    # The below sizes are mentioned assuming the shape of
sentence_portion = (1, 17, 16)
        Q = self.WQ(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)   # Query (1, 272)
        K = self.WK(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)   # Key (1, 272)
        V = self.WV(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)   # Value (1, 272)
        Q = Q.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)           # (1, 17, 16)
        K = K.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)           # (1, 17, 16)
        V = V.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)           # (1, 17, 16)

        ########################################## Using einsum ############################
        # This line creates a tensor A with the shape (1, 16, 16) by performing a batch matrix
multiplication
        # between the key tensor K (shape: (1, 17, 16)) and a tensor of ones with the same shape
as the query
        # tensor Q (shape: (1, 17, 16)).
        A = torch.einsum("bnm,bnk->bmk", K, torch.ones_like(Q))     # (1, 16, 16)
```

```
        # This line performs a batch matrix multiplication between the query tensor Q (shape: (1,
17, 16))
        # and the attention weight tensor A (shape: (1, 16, 16)) to obtain a dot product tensor
of shape
        # (1, 17, 16). This tensor represents the pairwise dot product similarity scores between
each query
        # vector and key vector for each position in the input sentence.
        QK_dot_prod = torch.einsum("bnm,bmk->bnk", Q, A)            # (1, 17, 16)
        # This line applies the softmax function along the second dimension of the dot product
tensor,
        # resulting in a row-wise normalized tensor with shape (1, 17, 16). Each row in the
resulting tensor
        # represents the attention distribution for a given query vector, i.e. how much attention
to pay to
        # each key vector.
        rowwise_softmax_normalizations = self.softmax(QK_dot_prod)  # (1, 17, 16)
        # This line performs a batch matrix multiplication between the attention weight tensor
        # (shape: (1, 16, 16)) and the value tensor V (shape: (1, 17, 16)) to obtain the output
        # tensor Z with shape (1, 17, 16). This tensor represents the final output of the
attention head.
        Z = torch.einsum("bnm,bnk->bnk", rowwise_softmax_normalizations, V)     # (1, 17, 16)
        # This line computes a scalar coefficient to normalize the output of the attention head
by dividing
        # by the square root of the dimension of the key, query and value vectors.
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(device) # (1)
        # This line normalizes the output tensor Z by scaling it with the coefficient.
        Z = coeff * Z
        return Z
```

The shape of each tensor is given in the comments assuming the following hyperparameter values.

- **`max_seq_length = 17`**
- **`embedding_size = 128`**
- **`num_atten_heads = 8`**

The above code implements a multi-headed attention mechanism for a given input sentence portion. It does this by matrix-multiplying the embedding vector for each word in the sentence by the WQ, WK, and WV matrices to produce the query vector Q, key vector K, and value vector V for each word in the input sentence. The dot product of Q and K is then used to calculate the attention weights, which are applied to the value vectors to get the final attention output.

Suggested improvements:

To improve the performance of a transformer on a small training set, several techniques can be employed. Firstly, transfer learning can be used by pretraining the transformer on a larger dataset and then fine-tuning it on the small dataset. This can lead to better generalization and faster convergence on the small dataset. Another technique is to use data augmentation to artificially increase the size of the training set. This can be done by applying random transformations to the input data such as cropping, flipping, or adding noise. This can help the model learn more robust features and reduce overfitting. Regularization techniques such as dropout and weight decay can also be used to prevent overfitting on the small training set. Dropout randomly drops out some neurons during training, while weight decay adds a penalty to the loss function for large weights. Finally, ensembling multiple models can also help improve performance on a small dataset. By training multiple models with different initializations or architectures and combining their predictions, the overall performance can be boosted. However, this approach may require more computational resources.

4. Lessons Learned

In this Vision transformer homework, we learned how to implement a transformer model for image classification. We used the `PyTorch` library to build a transformer model that consists of an encoder, where the encoder extracts features from an image. We also learned about the importance of attention mechanisms in transformer models, which allow the model to focus on important parts of the input. Additionally, we experimented with different hyperparameters and techniques to improve the performance of our model. Overall, this homework provided a practical introduction to transformer models and their application in computer vision tasks. Through experimentation and fine-tuning, we gained insights into how different components and hyperparameters of the model affect its performance. These lessons can be applied to other transformer-based models and tasks and can help us develop more accurate and efficient deep learning models.

--- End of the document ---