# Homework 9 - Deep Learning (ECE 60146)

Souradip Pal
pal43@purdue.edu
PUID 0034772329

April 29, 2023

## 1 Introduction

In this homework, the programming task gives an overview of how to create a **Vision Transformer(ViT)** in PyTorch and use it for the classification of images. In this assignment, a custom Vision Transformer(ViT) network was created, consisting of multi-headed attention layers in an encoder network. The network was trained and validated using a subset of images present in the MS-COCO dataset(2014 version) which was downloaded with the help of **COCO API**, a library for managing the dataset. Finally, the performance of the network was compared by computing the confusion matrix to understand which images were correctly or incorrectly labeled for each of the classes. Some ideas related to the logic of downloading the COCO dataset, running the Transformer training and validation loops were taken from the source code of the **DLStudio** module for completing this assignment. Some help was also taken from this blog post on Vision Transformers.

## 2 Methodology

Initially, to get familiar with the code for training and testing transformer networks, the scripts *seq2seq_with_transformerFG.py* and *seq2seq_with_transformerPreLN.py* from the **ExamplesTransformers** example directory in the **DLStudio** module was run. It illustrated how the transformer architecture was constructed with different types of attention layers and how it can be operated on textual data for sequence-sequence learning. A similar approach was taken for the programming tasks on images where a custom Dataset and a Dataloader were created which were then used for the training and validation of ViT model. For comparing the model, a method was created to calculate the confusion matrix to measure the accuracy of the model and how it performed in classifying each of the classes. The following section gives a detailed description of each of the approaches and the results obtained from the experiments performed on the ViT model.

## 3 Implementation and Results

### 3.1 Task 3: Programming Tasks

#### 3.1.1 Building Custom ViT

- **ViT Architecture**: In this task, a vision transformer network *CustomViT* was created as per the instructions in the assignment. *CustomViT* has a projection layer that converts the flattened patches of the image into embedding vectors. These patch embeddings are then passed to


Figure 1: Sample of training images from the COCO dataset

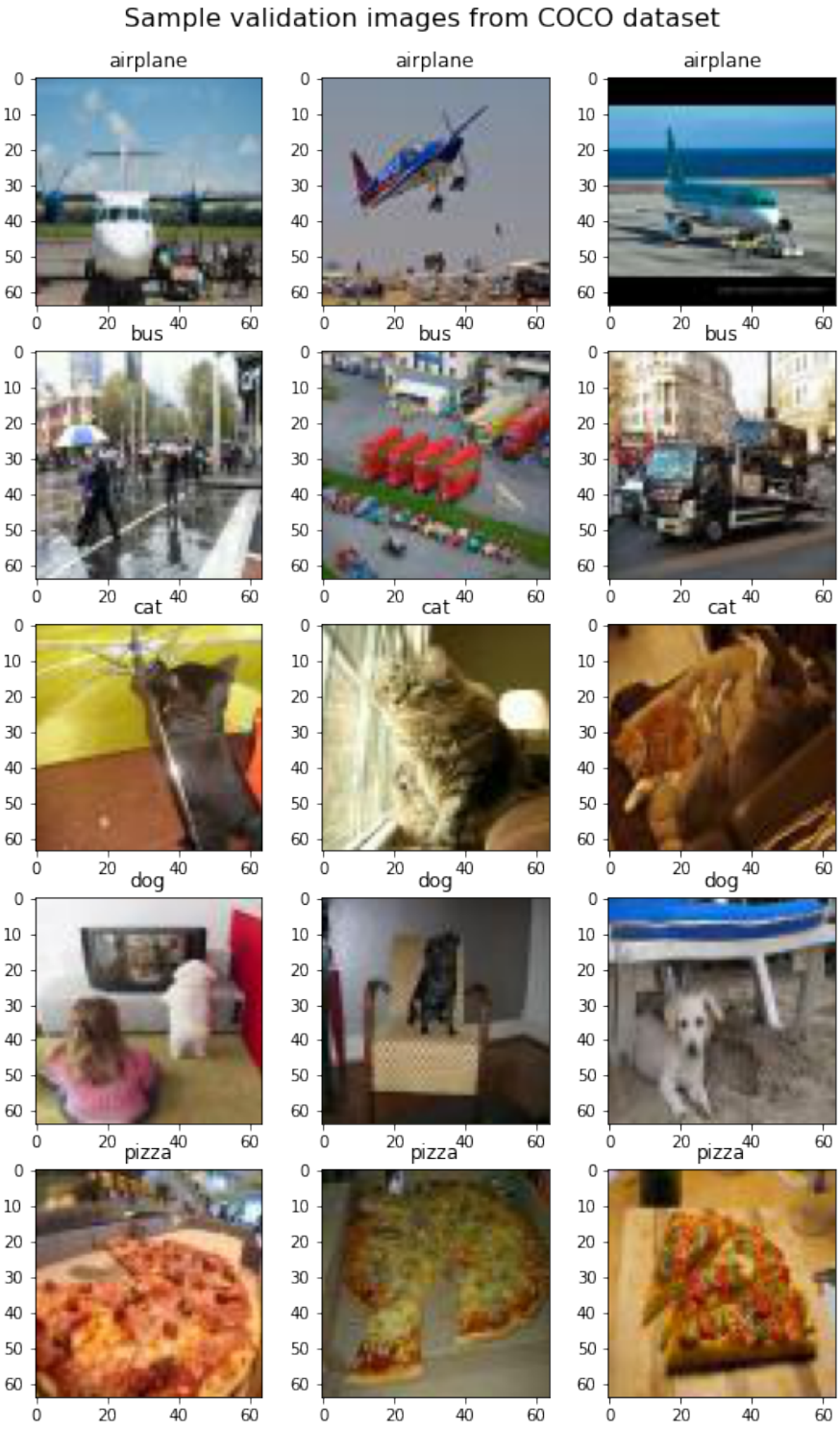



Figure 2: Sample of validation images from the COCO dataset

the Transformer Encoder network having Multi-Headed Self-Attention layers after prepending them with the class tokens and applying the appropriate positional embeddings. The positional embeddings and class tokens were defined as learnable parameters using the *nn.Parameter* module. The implementation of the Multi-Headed Self-Attention layers is taken directly from the **MasterEncoder** module present in the *VitHelper.py* file provided along with the instructions. The output of the encoder is then fed to an MLP head for classification. A 2D convolution layer was used to convert the patches of the images directly to embeddings. The input and output dimensions for the projection layer were calculated based on the number of output channels which essentially gives the length of the embedding vector used. The dimension of the output for a convolutional layer is calculated using the formula $\lfloor \frac{W-F+2P}{S} \rfloor + 1$ where W denotes the height or width, F denotes the kernel size, P denotes the padding and S denotes the stride. **Here, the convolution layer is set to have 128 output channels, a kernel size of 16, and a stride of 16**. Thus the dimension of the projection layer is calculated as follows:

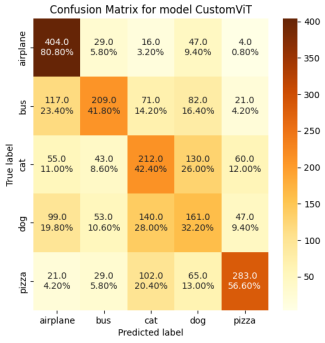
$$\begin{aligned}(B, 3, 64, 64) &\xrightarrow[in\_ch=3, out\_ch=128]{conv2D(kernel\_size=16, stride=16)} (B, 128, 4, 4) \\ &\xrightarrow{flatten} (B, 128, 16) \\ &\xrightarrow{permute} (B, 16, 128)\end{aligned}$$

Here $B$ denotes the batch size. The final output shape is of the form $(B, N_w, M)$ where $N_w = 16$ is the number of tokens and $M = 128$ is the length of the embedding vector. Also, only **2 basic encoder blocks** and **2 attention heads** were used for the *MasterEncoder* model. The total number of learnable layers of the entire network came out to be **46** and the total number of learnable parameters was ∼**52.2M** The following code block gives a description of the *CustomViT* network.

```
# Custom ViT Network
class CustomViT(nn.Module):
    def __init__(self, img_dim = 64,
                 in_channels=3,
                 patch_dim=16,
                 num_classes=5,
                 embed_dim=128,
                 num_blocks=2,
                 num_heads=2,
                 dropout=0.2):
        super(CustomViT, self).__init__()

        self.patch_dim = patch_dim
        # tokens = number of patches
        tokens = (img_dim // patch_dim) ** 2
        self.token_dim = in_channels * (patch_dim ** 2)
        self.embed_dim = embed_dim
        self.dim_head = (int(self.embed_dim / num_heads))

        # Projection and pos embeddings
        self.project_patches_2d = nn.Conv2d(in_channels,
                                            self.embed_dim,
                                            kernel_size = self.patch_dim,
                                            stride = self.patch_dim)
        self.emb_dropout = nn.Dropout(dropout)
        self.cls_token = nn.Parameter(torch.randn(1, 1, self.embed_dim))
        self.pos_emb = nn.Parameter(torch.randn(tokens + 1, self.embed_dim))

        # Master Encoder
        self.transformer = MasterEncoder(tokens + 1, self.embed_dim, num_blocks, num_heads)

        # MLP head
        self.mlp_head = nn.Linear(self.embed_dim, num_classes)

    def forward(self, img):
        # project patches with conv layer
```

```
img_patches = self.project_patches_2d(img)

img_patches = torch.permute(img_patches, (0, 2, 3, 1))
batch_size, num_patch_x, num_patch_y, dim = img_patches.shape

img_patches = img_patches.view(batch_size, -1, dim)
tokens = num_patch_x*num_patch_y

# Prepend class token parameter
class_tokens = self.cls_token.repeat(batch_size, 1, 1)
img_patches = torch.cat((class_tokens, img_patches), dim=1)

# add learnable postion embeddings + dropout
img_patches = img_patches + self.pos_emb[:tokens + 1, :]
patch_embeddings = self.emb_dropout(img_patches)

# feed patch_embeddings and output of transformer. shape: [batch, tokens, embed_dim]
y = self.transformer(patch_embeddings)

# only the cls token is used for classification
return self.mlp_head(y[:, 0, :])
```


(a) Training loss vs Iterations for *CustomViT*




(b) Confusion Matrix for *CustomViT* validations. Accuracy = 50.76%

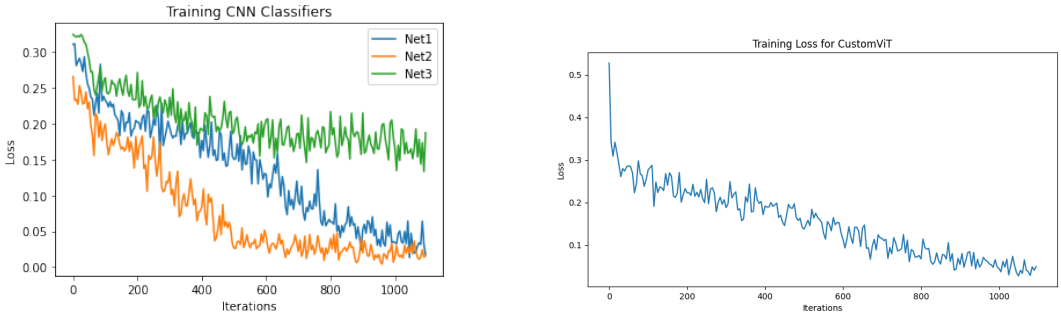
Figure 3: Training loss and the confusion matrix for vision transformer *CustomViT*

### 3.1.2 Image Classification using Custom ViT

- **Creating Dataset**: In order to use the **COCO API** for managing the MS-COCO dataset, the python version of the API called *pycocotools* was downloaded. Since the 2014 version of the COCO dataset was used for this homework, the 2014 Train/Val annotation zip file was downloaded from the MS-COCO website. It contained a JSON file named *instances_train2014.json* which contained details of the images present in the *train2014.zip* file including the image ids, image URLs, image categories, etc. Here, only the following five categories : [‘airplane’, ‘bus’, ‘cat’, ‘dog’, ‘pizza’] were selected, and the first 1500 image URLs for training and the last 500 image URLs for validation were taken from each category. The images are then downloaded using those URLs using the *requests* python library and saved in separate directories named after each of the categories. Moreover, the images were downsampled to a smaller size of $64 \times 64$ using the *PIL* library before saving. Shown in Fig. 1 and Fig. 2 are samples of the images from the training and validation sets for each of the five classes.

  A custom *CocoDataset* class was created from *torch.utils.data.Dataset* class to load the images

from each of the class directories after applying the necessary transforms. Finally, a *Dataloader* was created to wrap the dataset for processing the images in batches of 128 for training and validation.




(a) Training loss vs Iterations for *CNN* models

(b) Training loss vs Iterations for *CustomViT*

Figure 4: Training loss comparison of CNN models with *CustomViT*

- **CustomViT Training and Validation**: Using the training routine similar to the one created for Homework4, the network was trained for 20 epochs with optimizer as Adam having parameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$ and learning rate 0.001. In order to validate the network, a validation routine was created where the predicted labels were obtained via. model evaluation. These predictions were then used to calculate the confusion matrix. Fig. 3 shows the training loss and the confusion matrix for *CustomViT*.
- **Results & Comparison**: Table 1 shows the performance comparison of the *CustomViT* model with the different CNN architectures used for classifying a subset of the COCO image dataset. It can be seen the validation accuracies of the CNN models are higher than *CustomViT*. However, the training loss for *CustomViT* is comparable to that of the CNN models as seen in Fig 4. This implies that the vision transformer has overfitted the 7.5K training images. An additional dropout layer was also introduced to add some regularization to the model. However, a model with such complexity is unaffected by regularization and this issue can only be avoided by using more training data.

| **Model Type** | **Validation Accuracy(%)** |
|---|---|
| Net1 (CNN without padding) | 52.20 |
| Net2 (CNN with padding) | 57.56 |
| Net3 (CNN with padding - 10 layer) | 60.28 |
| CustomViT | 50.76 |

Table 1: Validation Accuracies of different models on MS-COCO dataset

### 3.1.3 Multi-Headed Self-Attention using torch.einsum (Extra Credit)

A simpler way to implement the Multi-Headed Self-Attention mechanism is by using the popular *torch.einsum*. The *torch.einsum* operation provides an easier way to perform batch matrix multiplication using symbolic equation strings. The operation is based on the famous Einstein notation used for tensor processing. The following code block gives the implementation of self-attention using *torch.einsum*.

```
# Multi-Headed Self-Attention using torch.einsum
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
        self.embedding_size = embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads

        self.Wqkv = nn.Linear(max_seq_length * self.embedding_size,
                              max_seq_length * self.qkv_size * self.num_atten_heads * 3)

        self.coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float())
        self.softmax = nn.Softmax(dim=3)


    def forward(self, sentence_tensor):
        # Output: [batch_size, seq_length * emb_size * 3]
        qkv = self.Wqkv(sentence_tensor.reshape(sentence_tensor.shape[0], -1).float())

        # Output: [batch_size, seq_length, qkv_size, num_heads, 3]
        qkv = qkv.view(-1, self.max_seq_length, self.qkv_size, self.num_atten_heads, 3)

        # Output: [3, batch_size, num_heads, seq_length, qkv_size]
        qkv = torch.permute(qkv, (4, 0, 3, 1, 2))
        q, k, v = qkv[0], qkv[1], qkv[2]

        # Output: [batch_size, num_heads, seq_length, seq_length]
        QK_dot_prod = torch.einsum('b h i d , b h j d -> b h i j', q, k)
        QK_dot_prod_norm = self.softmax(QK_dot_prod) * self.coeff

        # Output: [batch_size, num_heads, seq_length, qkv_size]
        Z = torch.einsum('b h i j , b h j d -> b h i d', QK_dot_prod_norm, v)

        # Output: [batch_size, seq_length, num_heads, qkv_size]
        Z = torch.permute(Z, (0, 2, 1, 3))

        return Z.reshape(sentence_tensor.shape[0], self.max_seq_length, -1)
```

# 4 Conclusion

In conclusion, a vision transformer can perform well in Image classification. However, the architecture of the transformer layers should be designed carefully to get better accuracy and faster training time. Although increasing the number of Basic Encoder blocks and the number of attention heads may result in higher classification accuracy, it may lead to slowness in training due to the increase in the number of trainable parameters and often leads to out-of-GPU memory. Also, some regularization in the form of dropout may control model overfitting to some extent. Hence, well-designed transformer architectures containing these practical layer elements like dropout or LayerNorm along with appropriate hyper-parameters are essential in making the training much more efficient and stable and getting higher classification accuracy.

# 5 Source Code

```
# -*- coding: utf-8 -*-
"""hw9_SouradipPal.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Bc7X7RYgHTixbdYQvODevXEyRkpH9QI6
"""
```

```
from google.colab import drive
drive.mount('/content/drive')

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/HW9

!wget -O DLStudio-2.2.7.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/DLStudio-2.2.7.tar.gz?download

!tar -xvf DLStudio-2.2.7.tar.gz -C /content/drive/MyDrive/Purdue/ECE60146/

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.7

!pip install pymsgbox

!python setup.py install

# %load_ext autoreload
# %autoreload 2

!pip install pycocotools

import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt

from tqdm import tqdm
from PIL import Image
from pycocotools.coco import COCO

seed = 0
random.seed(seed)
np.random.seed(seed)

from DLStudio import *

!wget -O /content/en_es_corpus_for_learning_with_Transformers.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/en_es_corpus_for_learning_with_Transformers.tar.gz

!tar -xvf /content/en_es_corpus_for_learning_with_Transformers.tar.gz -C \
    /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.7/ExamplesTransformers/

!python ExamplesTransformers/seq2seq_with_transformerFG.py

!python ExamplesTransformers/seq2seq_with_transformerPreLN.py

!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW9/coco
!wget --no-check-certificate http://images.cocodataset.org/annotations/annotations_trainval2014.zip \
    -O /content/drive/MyDrive/Purdue/ECE60146/HW9/coco/annotations_trainval2014.zip

!unzip /content/drive/MyDrive/Purdue/ECE60146/HW9/coco/annotations_trainval2014.zip \
    -d /content/drive/MyDrive/Purdue/ECE60146/HW9/coco/

!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW9/coco/train2014
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW9/coco/val2014
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW9/saved_models

# Image Downloader class to download COCO images
class ImageDownloader():
    def __init__(self, root_dir, annotation_path, classes, imgs_per_class):
        self.root_dir = root_dir
        self.annotation_path = annotation_path
        self.classes = classes
        self.images_per_class = imgs_per_class
        self.class_dir = {}
        self.coco = COCO(self.annotation_path)

    # Create directories same as category names to save images
    def create_dir(self):
        for c in self.classes:
            dir = os.path.join(self.root_dir, c)
```

```
            self.class_dir[c] = dir
            if not os.path.exists(dir):
                os.makedirs(dir)

    # Download images
    def download_images(self, download = True, val = False):
        img_paths = {}
        for c in tqdm(self.classes):
            class_id = self.coco.getCatIds(c)
            img_id = self.coco.getImgIds(catIds=class_id)
            imgs = self.coco.loadImgs(img_id)
            img_paths[c] = []

            indices = np.arange(0, self.images_per_class)
            if val == True:
                indices = np.arange(len(imgs) - self.images_per_class, len(imgs))

            for i in indices:
                img_path = os.path.join(self.root_dir, c, imgs[i]['file_name'])
                if download:
                    done = self.download_image(img_path, imgs[i]['coco_url'])
                    if done:
                        self.resize_image(img_path)
                        img_paths[c].append(img_path)
                else:
                    img_paths[c].append(img_path)

        return img_paths

    # Download image from URL using requests
    def download_image(self, path, url):
        try:
            img_data = requests.get(url).content
            with open(path, 'wb') as f:
                f.write(img_data)
            return True
        except Exception as e:
            print(f"Caught exception: {e}")
        return False

    # Resize image
    def resize_image(self, path, size = (64, 64)):
        im = Image.open(path)
        if im.mode != "RGB":
            im = im.convert(mode="RGB")
        im_resized = im.resize(size, Image.BOX)
        im_resized.save(path)

classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']
train_imgs_per_class = 1500
val_imgs_per_class = 500

# Download training images
train_downloader = ImageDownloader('/content/drive/MyDrive/Purdue/ECE60146/HW9/coco/train2014',
                '/content/drive/MyDrive/Purdue/ECE60146/HW9/coco/annotations/instances_train2014.json',
                classes, train_imgs_per_class)
train_downloader.create_dir()
train_img_paths = train_downloader.download_images(download = False)

# Download validation images
val_downloader = ImageDownloader('/content/drive/MyDrive/Purdue/ECE60146/HW9/coco/val2014',
                '/content/drive/MyDrive/Purdue/ECE60146/HW9/coco/annotations/instances_train2014.json',
                classes, val_imgs_per_class)
val_downloader.create_dir()
val_img_paths = val_downloader.download_images(download = False, val = True)

# Plotting sample training images
fig, axes = plt.subplots(5, 3, figsize=(9, 15))

for i, cls in enumerate(classes):
    for j, path in enumerate(train_img_paths[cls][:3]):
        im = Image.open(path)
        axes[i][j].imshow(im)
        axes[i][j].set_title(cls)

fig.suptitle('Sample training images from COCO dataset', fontsize=16, y=0.92)
```

```
plt.show()

# Plotting sample validation images
fig, axes = plt.subplots(5, 3, figsize=(9, 15))

for i, cls in enumerate(classes):
    for j, path in enumerate(val_img_paths[cls][:3]):
        im = Image.open(path)
        axes[i][j].imshow(im)
        axes[i][j].set_title(cls)

fig.suptitle('Sample validation images from COCO dataset', fontsize=16, y=0.92)
plt.show()

import os
import torch

# Custom dataset class for COCO
class CocoDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms=None):
        super().__init__()
        self.root_dir = root
        self.classes = os.listdir(self.root_dir)
        self.transforms = transforms
        self.img_paths = []
        self.img_labels = []
        self.class_to_idx = {'airplane':0, 'bus':1, 'cat':2, 'dog':3, 'pizza': 4}
        self.idx_to_class = {i:c for c, i in self.class_to_idx.items()}

        for cls in self.classes:
            cls_dir = os.path.join(self.root_dir, cls)
            paths = os.listdir(cls_dir)
            self.img_paths+= [os.path.join(cls_dir, path) for path in paths]
            self.img_labels+=[self.class_to_idx[cls]]*len(paths)

    def __len__(self):
        # Return the total number of images
        return len(self.img_paths)

    def __getitem__(self, index):
        index = index % len(self.img_paths)
        img_path = self.img_paths[index]
        img_label = self.img_labels[index]
        img = Image.open(img_path)
        img_transformed = self.transforms(img)
        return img_transformed, img_label

import torchvision.transforms as tvt

reshape_size = 64
transforms = tvt.Compose([
    tvt.ToTensor(),
    tvt.Resize((reshape_size, reshape_size), antialias=True),
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# Create custom training dataset
train_dataset = CocoDataset('/content/drive/MyDrive/Purdue/ECE60146/HW9/coco/train2014/', transforms=transforms)

len(train_dataset)

# Create custom validation dataset
val_dataset = CocoDataset('/content/drive/MyDrive/Purdue/ECE60146/HW9/coco/val2014/', transforms=transforms)

len(val_dataset)

# Create custom training/validation dataloader
train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True, num_workers=2)
val_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=128, shuffle=False, num_workers=2)

# Check dataloader
train_loader_iter = iter(train_data_loader)
img, target = next(train_loader_iter)

print('img has length: ', len(img))
print('target has length: ', len(target))
```

```
print(img[0].shape)

# Routin to train a neural network classifier
def train_net(device, net, optimizer, criterion, data_loader, model_name,
              epochs = 10, display_interval = 100):
    net = net.to(device)
    loss_running_record = []

    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(data_loader):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if (i+1) % display_interval == 0:
                avg_loss = running_loss / display_interval
                print ("[epoch : %d, batch : %5d] loss : %.3f" % (epoch + 1, i + 1, avg_loss))
                loss_running_record.append(avg_loss)
            running_loss = 0.0

    checkpoint_path = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW9/saved_models',
                                   f'{model_name}.pth')
    torch.save(net.state_dict(), checkpoint_path)

    return loss_running_record

# Plotting training loss
def plot_loss(loss, display_interval, model_name):
    plt.figure(figsize=(10,5))
    plt.plot(np.arange(len(loss))*display_interval, loss);
    plt.title(f'Training Loss for {model_name}')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.savefig(f"/content/drive/MyDrive/Purdue/ECE60146/HW9/training_loss_{model_name}.png")
    plt.show()

# Routine to validate a neural network classifier
def validate_net(device, net, data_loader, model_path = None):
    if model_path is not None:
        net.load_state_dict(torch.load(model_path))
    net = net.to(device)
    net.eval()
    running_loss = 0.0

    device_cpu = torch.device('cpu')

    iters = 0
    imgs = []
    all_labels = []
    all_pred = []

    with torch.no_grad():
        for i, data in enumerate(data_loader):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss
            iters += 1
            pred_labels = torch.argmax(outputs.data, axis = 1)
            all_labels += list(labels.to(device_cpu).numpy())
            all_pred += list(pred_labels.to(device_cpu).numpy())
            imgs += list(inputs.to(device_cpu))

        print(f"Validation Loss: {running_loss/iters}")

    return imgs, all_labels, all_pred

# Function to calculate confusion matrix
```

```
def calc_confusion_matrix(num_classes, actual, predicted):
    conf_mat = np.zeros((num_classes, num_classes))
    for a, p in zip(actual, predicted):
        conf_mat[a][p]+=1
    return conf_mat

import seaborn as sns

# Plotting confusion matrix
def plot_conf_mat(conf_mat, classes, model_name):
    labels = []
    num_classes = len(classes)
    labels_per_class =  np.sum(conf_mat) / num_classes
    for row in range(num_classes):
        rows = []
        for col in range(num_classes):
            count = conf_mat[row][col]
            percent = "%.2f%%" % (count / labels_per_class * 100)
            label = str(count) + '\n' + str(percent)
            rows.append(label)
        labels.append(rows)
    labels = np.asarray(labels)

    plt.figure(figsize=(6, 6))
    sns.heatmap(conf_mat, annot=labels, fmt="", cmap="YlOrBr", cbar=True,
                xticklabels=classes, yticklabels=classes)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title(f'Confusion Matrix for model {model_name}')
    plt.savefig(f"/content/drive/MyDrive/Purdue/ECE60146/HW9/conf_mat_{model_name}.png")
    plt.show()

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/HW9

import torch.nn as nn
import torch.nn.functional as F

from ViTHelper import MasterEncoder

# Custom ViT Network
class CustomViT(nn.Module):
    def __init__(self, img_dim = 64,
                 in_channels=3,
                 patch_dim=16,
                 num_classes=5,
                 embed_dim=128,
                 num_blocks=2,
                 num_heads=2,
                 dropout=0.2):
        super(CustomViT, self).__init__()

        self.patch_dim = patch_dim
        # tokens = number of patches
        tokens = (img_dim // patch_dim) ** 2
        self.token_dim = in_channels * (patch_dim ** 2)
        self.embed_dim = embed_dim
        self.dim_head = (int(self.embed_dim / num_heads))

        # Projection and pos embeddings
        self.project_patches_2d = nn.Conv2d(in_channels,
                                            self.embed_dim,
                                            kernel_size = self.patch_dim,
                                            stride = self.patch_dim)
        self.emb_dropout = nn.Dropout(dropout)
        self.cls_token = nn.Parameter(torch.randn(1, 1, self.embed_dim))
        self.pos_emb = nn.Parameter(torch.randn(tokens + 1, self.embed_dim))

        # Master Encoder
        self.transformer = MasterEncoder(tokens + 1, self.embed_dim, num_blocks, num_heads)

        # MLP head
        self.mlp_head = nn.Linear(self.embed_dim, num_classes)

    def forward(self, img):
        # project patches with conv layer
```

```
        img_patches = self.project_patches_2d(img)

        img_patches = torch.permute(img_patches, (0, 2, 3, 1))
        batch_size, num_patch_x, num_patch_y, dim = img_patches.shape

        img_patches = img_patches.view(batch_size, -1, dim)
        tokens = num_patch_x*num_patch_y

        # Prepend class token parameter
        class_tokens = self.cls_token.repeat(batch_size, 1, 1)
        img_patches = torch.cat((class_tokens, img_patches), dim=1)

        # add learnable postion embeddings + dropout
        img_patches = img_patches + self.pos_emb[:tokens + 1, :]
        patch_embeddings = self.emb_dropout(img_patches)

        # feed patch_embeddings and output of transformer. shape: [batch, tokens, embed_dim]
        y = self.transformer(patch_embeddings)

        # only the cls token is used for classification
        return self.mlp_head(y[:, 0, :])

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

# Initialize CustomViT
net = CustomViT()

# model_path = '/content/drive/MyDrive/Purdue/ECE60146/HW9/saved_models/net1_conv.pth'
# net.load_state_dict(torch.load(model_path))

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters() , lr=1e-3, betas=(0.5, 0.999))
epochs = 20
display_interval = 5

num_learnable_params= sum(p.numel() for p in net.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters: %d\n" % num_learnable_params)

num_layers = len(list(net.parameters()))
print("\nThe number of layers: %d\n" % num_layers)

# Train Net1
net1_losses = train_net(device, net, optimizer=optimizer, criterion=criterion,
                        data_loader = train_data_loader, model_name = 'net1_conv',
                        epochs=epochs, display_interval = display_interval)

# Plotting Net1 training loss
plot_loss(net1_losses, display_interval, 'CustomViT')

# Validate Net1
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW9/saved_models/net1_conv.pth'
imgs, actual, predicted = validate_net(device, net, val_data_loader, model_path = save_path)

# Calculate Net1 confusion matrix and accuracy
conf_mat = calc_confusion_matrix(5, actual, predicted)
print(conf_mat)
accuracy = np.trace(conf_mat) / float(np.sum(conf_mat))
print(accuracy)

# Plotting Net1 confusion matrix
plot_conf_mat(conf_mat, classes, 'CustomViT')

# Multi-Headed Self-Attention using torch.einsum
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
        self.embedding_size = embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads

        self.Wqkv = nn.Linear(max_seq_length * self.embedding_size,
                              max_seq_length * self.qkv_size * self.num_atten_heads * 3)

        self.coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float())
        self.softmax = nn.Softmax(dim=3)
```

```
def forward(self, sentence_tensor):
    # Output: [batch_size, seq_length * emb_size * 3]
    qkv = self.Wqkv(sentence_tensor.reshape(sentence_tensor.shape[0], -1).float())

    # Output: [batch_size, seq_length, qkv_size, num_heads, 3]
    qkv = qkv.view(-1, self.max_seq_length, self.qkv_size, self.num_atten_heads, 3)

    # Output: [3, batch_size, num_heads, seq_length, qkv_size]
    qkv = torch.permute(qkv, (4, 0, 3, 1, 2))
    q, k, v = qkv[0], qkv[1], qkv[2]

    # Output: [batch_size, num_heads, seq_length, seq_length]
    QK_dot_prod = torch.einsum('b h i d , b h j d -> b h i j', q, k)
    QK_dot_prod_norm = self.softmax(QK_dot_prod) * self.coeff

    # Output: [batch_size, num_heads, seq_length, qkv_size]
    Z = torch.einsum('b h i j , b h j d -> b h i d', QK_dot_prod_norm, v)

    # Output: [batch_size, seq_length, num_heads, qkv_size]
    Z = torch.permute(Z, (0, 2, 1, 3))

    return Z.reshape(sentence_tensor.shape[0], self.max_seq_length, -1)
```