SPRING 2023 ECE 60146 – Homework 9

Sahithi Kodali – 34789866
kodali1@purdue.edu

3.1 Building Your Own ViT

ViT (Vision Transformers) is implemented in this homework on the COCO dataset from HW4 consisting of 64X64 images from five classes for image classification task. This dataset has about 7.5k images for training and 2.5k images for evaluation. The ViT network implemented can be seen in Figure 1 which is commented in detail for understanding its working.

```
class ViT(nn.Module):
    def __init__(self):
        super().__init__()

        #initialize class token and postion embeddings as learnable parameters
        self.class_token = nn.Parameter(torch.zeros(1, 1, hidden_size))
        self.position_embeddings = nn.Parameter(torch.zeros(1, num_patches + 1, hidden_size))

        #the parameters adjusted to normal distribution and a standard deviation 0.02
        nn.init.trunc_normal_(self.class_token, std=0.02)
        nn.init.trunc_normal_(self.position_embeddings, std=0.02)

        #convolutional layer to extract patches from input image and apply linear transformation
        self.conv = nn.Conv2d(3, hidden_size, kernel_size=patch_size, stride=patch_size)

        #the masterencoder that creates a stack of basic encoder layers using Multi-head self attention
        #max_seq_length  = num_patches+1
        self.encoder = MasterEncoder(max_seq_length = 17, embedding_size = 256,
                                     how_many_basic_encoders = 6, num_atten_heads = 8 )

        #final linear layer to map the hidden states to the number of classes i.e 5 here
        self.mlp = nn.Linear(hidden_size, num_classes)
    def forward(self, x):

        #apply convolutional layer to input image
        x = self.conv(x) #(batch_size, hidden_size, num_patches, num_patches)

        #flatten the spatial dimensions and interchange the axes
        x = x.flatten(2).transpose(1, 2) #(batch_size, num_patches, hidden_size)

        #class tokens for all the images in batch are repeated and concatenated with the feature maps attained
        cls_tokens = self.class_token.repeat(x.shape[0], 1, 1) #(batch_size, 1, hidden_size)
        x = torch.cat((cls_tokens, x), dim=1) #(batch_size, num_patches+1, hidden_size)

        #add the postional embeddings to each patch
        x = x + self.position_embeddings

        #pass the sequence input into the series of encoders
        x = self.encoder(x)

        #attain the first vector that represents the whole image
        x = x[:, 0]

        #apply linear layer to attain the class probabilities
        x = self.mlp(x)

        return x
```

Figure 1: ViT implementation code block

3.2 Image Classification with Your ViT

The ViT model is trained for 10 epochs using Adam optimizer and Cross Entropy Loss function with learning rate 1e-4. The training loss to iterations plot shown in Figure 2 shows that the loss decreases almost consistently with each epoch and the accuracy to iterations plot in Figure 3 shows the accuracy to increase to almost 95%.

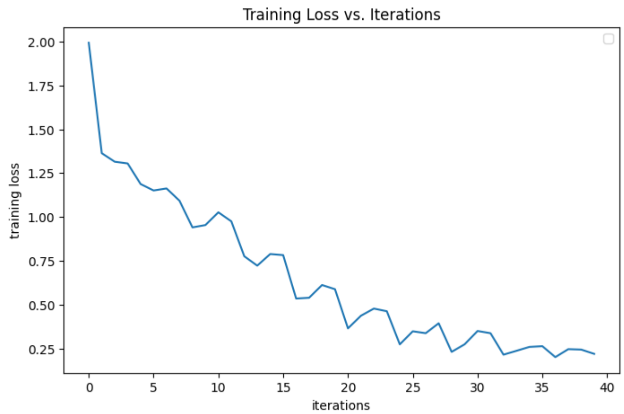

Figure 2: Loss vs iterations plot

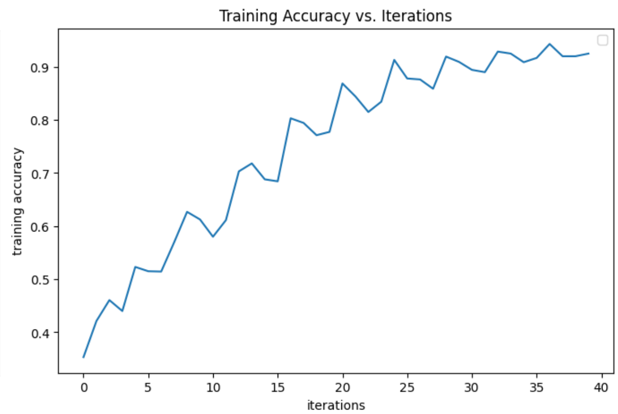

Figure 3: Accuracy vs iterations plot

The trained ViT model is evaluated on the 2.5k unseen images of COCO dataset. The confusion matrix and accuracy achieved of ~52% can be seen in Figure 4. However, we can observe that the accuracy for evaluation is very low compared to training which clearly indicates that the model overfits. But this is expected due to the low amount of data used for training such a powerful Transformers model.

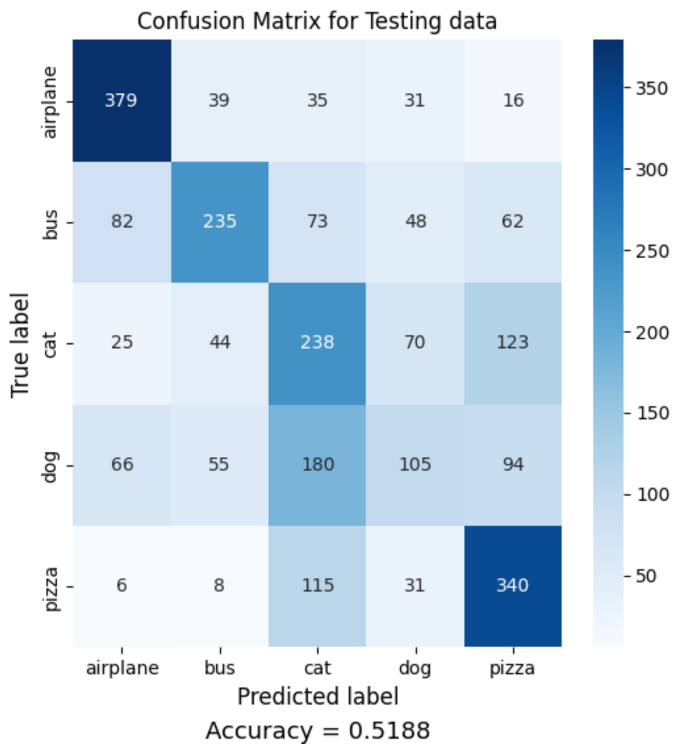

Figure 4: Confusion matrix and accuracy attained by ViT for testing data

Comparing CNN and ViT based Image classification:

The accuracy achieved by the CNN model in HW4 was about ~58% which is higher than the ~52% accuracy achieved by ViT model for the same data. Though it is expected that the ViT might perform better than the CNN model, this behavior is due to the less amount of data available to train ViT model that have large capacity. By increasing the amount of data, we can expect the accuracy of ViT model to increase than the CNN model, but for the data used here it is the inverse.

3.3 Extra Credit

For the extra credit task, the multi-headed self-attention mechanism is implemented using *torch.einsum* within 10 lines of code. The implementation code block can be seen in Figure 5.

```
#Multi-headed Self attention using torch.einsum
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

        #initialize three linear layers for query, key, values and one final output linear layer
        self.WQ = nn.Linear(embedding_size, embedding_size)
        self.WK = nn.Linear(embedding_size, embedding_size)
        self.WV = nn.Linear(embedding_size, embedding_size)
        self.fc = nn.Linear(embedding_size, embedding_size)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        #applying linear tranformations to the input
        #attain query, keys and values tensors of shape (batch_size, max_seq_length, embedding_size)
        Q = self.WQ(x)
        K = self.WK(x)
        V = self.WV(x)

        #compute attention scores between for the Q,K,V tensors to get output of shape (batch_size, max_seq_length, max_seq_length)
        #einsum does that easily for us by projecting the operation that needs to be done on the right hand side of '->'
        Q_ = torch.einsum('b i d, b j d -> b i j', Q, Q)
        K_ = torch.einsum('b i d, b j d -> b i j', K, K)
        V_ = torch.einsum('b i d, b j d -> b i j', V, V)

        #dot product of Q and K vectors is computed and scaled by the square root of the size of the vector
        #the product is sent to a softmax function to get the attention weights
        #this weights are then used to weight the value vectors to calculate the values of output at each position using einsum
        Z = torch.einsum('b i j, b j d -> b i d', self.softmax(
            torch.einsum('b i d, b j d -> b i j', Q, K)/torch.sqrt(torch.tensor([self.qkv_size], dtype=torch.float32, device=Q.device))), V)

        #apply linear tranformation to attain the final output tensor of shape (batch_size, max_seq_length, embedding_size)
        return self.fc(Z)
```

Figure 5: torch.einsum based multi-headed self-attention implementation

The model is again trained similarly and the loss to iterations, accuracy to iterations plots can be seen in Figures 6,7 respectively. The accuracy and the confusion matrix for testing data can also be seen in Figure 8 where the accuracy is about ~52% similarly.

Figure 6: Loss vs iterations plot

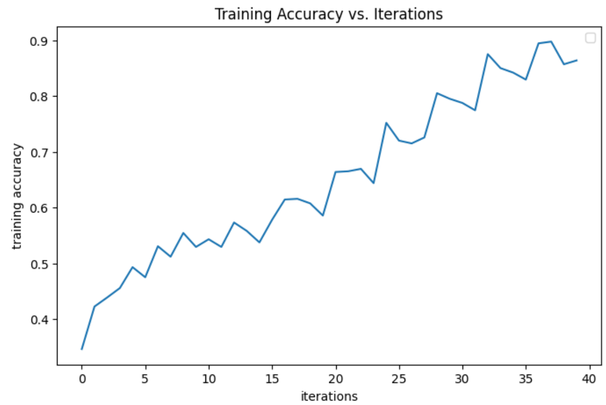

Figure 7: Accuracy vs iterations plot

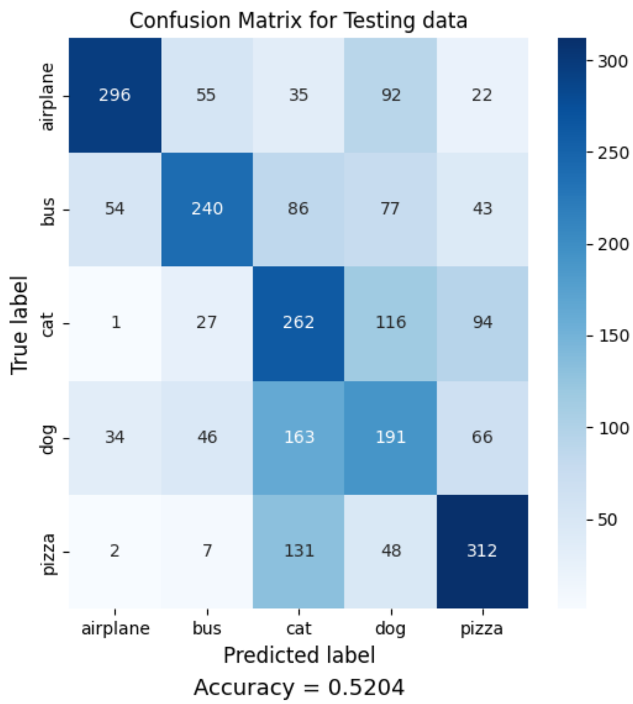

Figure 8: Confusion matrix and accuracy attained by ViT using torch.einsum for testing data

SOURCE CODE:

```
# -*- coding: utf-8 -*-
"""HW9.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1DyUAx_SOXCqsdMBroPq8dMdW8cezQweo
```

```
"""

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

# Commented out IPython magic to ensure Python compatibility.
#import libraries required
# %matplotlib inline
from pycocotools.coco import COCO
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
import random
import os
from shutil import copyfile
#import libraries
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import pandas as pd
import torchvision.transforms as tvt
import torch.nn as nn
import torch.nn.functional as F

#check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device

#create dataset and dataloader
class MyDataset(Dataset):

    #intializations
    def __init__(self, root_dir, transform = None):
        super().__init__()
        self.root_dir = root_dir
        self.transform = transform
        self.classes = []
        self.image_paths = []
        self.labels = []

        for cls_folder in os.listdir(self.root_dir):
          cls_dir = os.path.join(self.root_dir, cls_folder)

          #add class name
```

```
          if os.path.isdir(cls_dir):
            self.classes.append(cls_folder)

          #add image paths and corresponding class as a label
          for img_name in os.listdir(cls_dir):
            if img_name.endswith('.jpg'):
              self.image_paths.append(os.path.join(cls_dir,img_name))
              self.labels.append(len(self.classes)-1)

    #compute length of dataset
    def __len__(self):
        return len(self.image_paths)

    #apply transformations for the image chosen by index
    def __getitem__(self, index):
        img = Image.open(self.image_paths[index]).convert('RGB')
        label = self.labels[index]

        if self.transform:
            img = self.transform(img)

        return img, label

#initialize the dataset and dataloader and apply transformations as required

transform = tvt.Compose([tvt.ToTensor()])

train_dataset =
MyDataset('/content/drive/MyDrive/Purdue/HW4/custom_data/train_data', transform =
transform)
val_dataset =
MyDataset('/content/drive/MyDrive/Purdue/HW4/custom_data/valid_data', transform =
transform)

#check for the data length
print(len(train_dataset))
print(len(val_dataset))

#initialize batch and num workers
batch_size = 8
num_workers = 2

#create dataloader
train_data_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle =
True, num_workers=num_workers)
```

```
val_data_loader = DataLoader(val_dataset, batch_size = batch_size, shuffle =
True, num_workers=num_workers)

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

##           https://engineering.purdue.edu/kak/distDLS/

class MasterEncoder(nn.Module):
    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):
    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):
    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

#Multi-headed Self attention using torch.einsum
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

        #initialize three linear layers for query, key, values and one final
output linear layer
        self.WQ = nn.Linear(embedding_size, embedding_size)
        self.WK = nn.Linear(embedding_size, embedding_size)
        self.WV = nn.Linear(embedding_size, embedding_size)
        self.fc = nn.Linear(embedding_size, embedding_size)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        #applying linear tranformations to the input
        #attain query, keys and values tensors of shape (batch_size,
max_seq_length, embedding_size)
```

```
        Q = self.WQ(x)
        K = self.WK(x)
        V = self.WV(x)

        #compute attention scores between for the Q,K,V tensors to get output of
shape (batch_size, max_seq_length, max_seq_length)
        #einsum does that easily for us by projecting the operation that needs to
be done on the right hand side of '->'
        Q_ = torch.einsum('b i d, b j d -> b i j', Q, Q)
        K_ = torch.einsum('b i d, b j d -> b i j', K, K)
        V_ = torch.einsum('b i d, b j d -> b i j', V, V)

        #dot product of Q and K vectors is computed and scaled by the square root
of the size of the vector
        #the product is sent to a softmax function to get the attention weights
        #this weights are then used to weight the value vectors to calculate the
values of output at each position using einsum
        Z = torch.einsum('b i j, b j d -> b i d', self.softmax(
            torch.einsum('b i d, b j d -> b i j', Q,
K)/torch.sqrt(torch.tensor([self.qkv_size], dtype=torch.float32,
device=Q.device))), V)

        #apply linear tranformation to attain the final output tensor of shape
(batch_size, max_seq_length, embedding_size)
        return self.fc(Z)

#initilize the parameters
img_size = 64
patch_size = 16
num_classes = 5
hidden_size = 256
num_heads = 8
num_layers = 6
num_patches = (img_size // patch_size) ** 2

#define the ViT net

class ViT(nn.Module):
    def __init__(self):
        super().__init__()

        #initialize class token and postion embeddings as learnable parameters
        self.class_token = nn.Parameter(torch.zeros(1, 1, hidden_size))
        self.position_embeddings = nn.Parameter(torch.zeros(1, num_patches + 1,
hidden_size))
```

```
        #the parameters adjusted to normal distribution and a standard deviation
0.02
        nn.init.trunc_normal_(self.class_token, std=0.02)
        nn.init.trunc_normal_(self.position_embeddings, std=0.02)

        #convolutional layer to extract patches from input image and apply linear
transformation
        self.conv = nn.Conv2d(3, hidden_size, kernel_size=patch_size,
stride=patch_size)

        #the masterencoder that creates a stack of basic encoder layers using
Multi-head self attention
        #max_seq_length  = num_patches+1
        self.encoder = MasterEncoder(max_seq_length = 17, embedding_size = 256,
                                     how_many_basic_encoders = 6, num_atten_heads
= 8 )

        #final linear layer to map the hidden states to the number of classes i.e
5 here
        self.mlp = nn.Linear(hidden_size, num_classes)

    def forward(self, x):

        #apply convolutional layer to input image
        x = self.conv(x) #(batch_size, hidden_size, num_patches, num_patches)

        #flatten the spatial dimensions and interchange the axes
        x = x.flatten(2).transpose(1, 2) #(batch_size, num_patches, hidden_size)

        #class tokens for all the images in batch are repeated and concatenated
with the feature maps attained
        cls_tokens = self.class_token.repeat(x.shape[0], 1, 1) #(batch_size, 1,
hidden_size)
        x = torch.cat((cls_tokens, x), dim=1) #(batch_size, num_patches+1,
hidden_size)

        #add the postional embeddings to each patch
        x = x + self.position_embeddings

        #pass the sequence input into the series of encoders
        x = self.encoder(x)

        #attain the first vector that represents the whole image
        x = x[:, 0]
```

```
        #apply linear layer to attain the class probabilities
        x = self.mlp(x)

        return x

#create model instance
model = ViT()
model.to(device)

#check model summary
from torchsummary import summary

input_tensor = torch.randn(3, 64, 64).to(device)
summary(model, input_tensor.shape)

##Training begins...
#give loss and optimizer functions
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

#define epochs
num_epochs = 10

#store loss and accuracy
losses = []
accuracies = []

for epoch in range(num_epochs):
    running_loss = 0.0
    running_accuracy = 0.0

    for i, (images, labels) in enumerate(train_data_loader):
        images = images.to(device)
        labels = labels.to(device)

        #compute loss
        optimizer.zero_grad()

        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()

        optimizer.step()
```

```
        running_loss += loss.item()

        #compute accuracy
        preds = torch.argmax(outputs, dim=1)
        accuracy = torch.sum(preds == labels).item()/labels.size(0)
        running_accuracy += accuracy

        #store the running loss and accuracy
        if i % 200 == 199:
            avg_loss = running_loss / float(200)
            avg_accuracy = running_accuracy/float(200)
            losses.append(avg_loss)
            accuracies.append(avg_accuracy)

            print("[epoch:%d  iter:%4d]     loss: %.5f    accuracy: %.5f" %
(epoch+1, i+1, avg_loss, avg_accuracy))
            running_loss = 0.0
            running_accuracy = 0.0

    print(f"Epoch {epoch+1}/{num_epochs}, Training Loss:
{running_loss/len(train_data_loader):.4f}, Training Accuracy:
{running_accuracy/len(train_data_loader):.4f}")

#plotting loss vs iterations
plt.figure(figsize=(8,5))
plt.title("Training Loss vs. Iterations")
plt.plot(losses)

plt.xlabel("iterations")
plt.ylabel("training loss")
plt.legend()
plt.savefig("training_loss.png")
plt.show()

#plotting accuracy vs iterations
plt.figure(figsize=(8,5))
plt.title("Training Accuracy vs. Iterations")
plt.plot(accuracies)

plt.xlabel("iterations")
plt.ylabel("training accuracy")
plt.legend()
plt.savefig("training_accuracy.png")
plt.show()
```

```
#import libraries
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns

# make predictions on the test

model.eval()

y_true = []
y_pred = []

classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']

with torch.no_grad():
    for inputs, labels in val_data_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)

        outputs = model(inputs)

        _, predicted = torch.max(outputs.data, 1)
        y_true += labels.cpu().numpy().tolist()
        y_pred += predicted.cpu().numpy().tolist()

# construct the confusion matrix
cm = confusion_matrix(y_true, y_pred)
acc = accuracy_score(y_true, y_pred)

plt.figure(figsize=(6, 6))
sns.heatmap(cm, annot = True, cmap = 'Blues', xticklabels = classes, yticklabels
= classes, fmt = 'g')
plt.title('Confusion Matrix for Testing data')
plt.xlabel('Predicted label', fontsize = 12)
plt.ylabel('True label', fontsize = 12)
plt.text(2.5, 5.7, 'Accuracy = ' + str(acc), fontsize = 13, ha='center',
va='center')
plt.show()
```

****