ECE 60146 Deep Learning **Name:** *Nikita Ravi (ravi30)*
Homework 9
04/30/2023

1 Introduction

The objective of this homework assignment is to gain an understanding into the multi-headed self-attention mechanism and the transformer architecture. This is achieved by implementing our own Vision Transformer (ViT) for image classification.

2 Theoretical Background

2.1 Scaled Dot Product Attention

The attention mechanism looks at an input sequence and decides at each step which other parts of the sequence are important. The input sequence consists of *queries* and *keys*, each with a dimension d_k, and *values* of dimension d_v. The query Q is a vector representation of one element in the sequence, the keys K are vector representations of all the elements in the sequence, and finally the values V are also the vector representations of all the elements in the sequence [1].

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

The dot product between the each query with all the keys are computed, then scaled by $\frac{1}{\sqrt{d_k}}$ and apply a softmax function to obtain the weights on the values [2] as shown in Eq 1.

2.2 Multi-Head Attention

We partition the input tensor X along its embedded axis into N_H slices and apply single-headed attention to each slice [3]. For instance, suppose the sequence length is 9 and if the embedded size is 512 as recommended by the *Attention Is All You Need* paper and the number of heads N_H is 8, then the dimension of the input tensor X is 9×512 and each slice of all the 8 slices of X will have a dimension of 9×64 [4]. As illustrated in Figure 1, all eight slices will pass through the scaled dot product attention layer at the same time enabling the parallelization of this mechanism. We then concatenate all the outputs of the attention layer and push it through a fully connected linear layer.

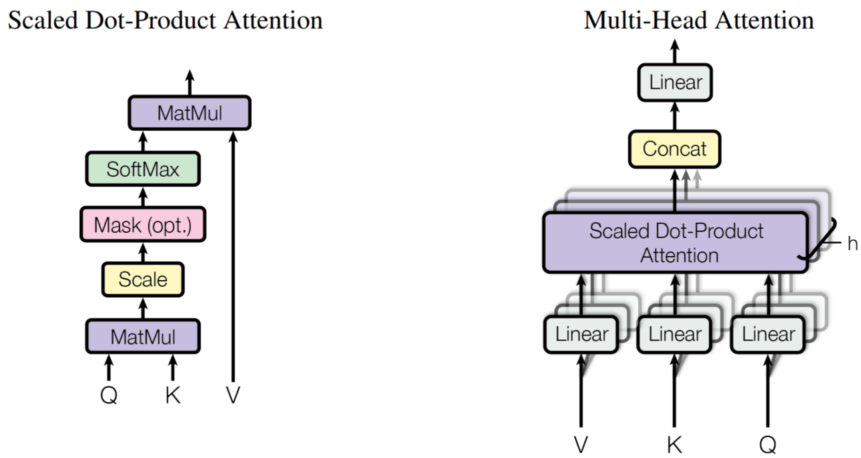


Figure 1: Multi-Head Attention [2]

2.3 Transformer

As shown in Figure 2, a transformer is composed of an encoder and a decoder. The encoder consists of six identical layers where each layer has two sub-layers. The first sub-layer is the multi-head attention layer described in the previous section and the second sub-layer is a position-wise fully connected feed-forward network. A residual connection around each of the two sub-layers is used followed by a layer normalization [2]

The decoder, whose job is to generate sequences, also consists of six identical layers where each layer has three sub-layers, two of which are the same as the ones found in the encoder. The third sub-layer performs multi-head attention over the output of the encoder stack. Similar to the encoder, the decoder also has residual connections around each of the sub-layers followed by layer normalizations [2]. The first self-attention sub-layer in the decoder is masked for the reason of preventing the computation of attention scores for future words [5]. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i [2].

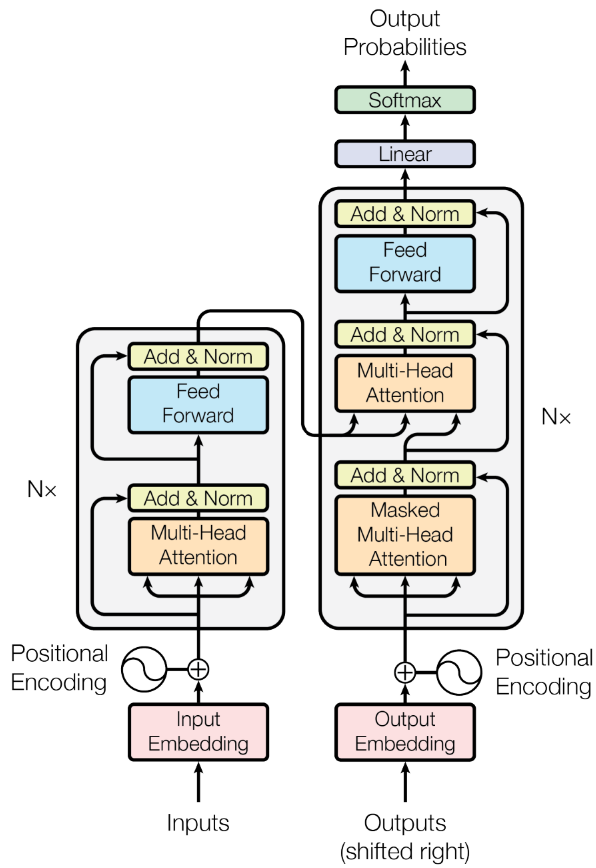


Figure 2: Transformers [2]

3 Methodology

3.1 Patch Embedding

The transformer encoder was developed with sequence data in mind but images are not sequences. So to "sequencify" an image we break it into multiple sub-images and map each sub=image to a vector. This is achieved by reshaping the batch input from (batch size, input channel, height, width) to (batch size, patch height × patch width, embedded size). A randomly initialized class token is pre-pended to the beginning of the input sequence to accumulate information from the other tokens in the sequence the deeper and more layers the transformer is [6]. We also add positional encoding to each patch in the image to identify the order of the patch sequence.

3.2 Classification Head

The Classification Head class is used predominantly to change the shape of the tensor from (batch size, sequence length, embedded size) to a shape of (batch size, number of classes) and pass this reshaped tensor through a linear layer to predict the class.

4 Implementation and Results

4.1 Attention Head using torch.einsum

```python
class AttentionHead2(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):
        batch_size = sentence_portion.shape[0]
        Q = self.WQ(sentence_portion.reshape(batch_size, -1).float()).to(device)
        K = self.WK(sentence_portion.reshape(batch_size, -1).float()).to(device)
        V = self.WV(sentence_portion.reshape(batch_size, -1).float()).to(device)

        Q = Q.view(batch_size, self.max_seq_length, self.qkv_size)
        K = K.view(batch_size, self.max_seq_length, self.qkv_size)
        V = V.view(batch_size, self.max_seq_length, self.qkv_size)

        QK_dot_prod = torch.einsum("bqn, bnk -> bqk", Q, rearrange(K, "b n e -> b e n")) # shape change from [16, 17, emb_size / heads] -> [16, emb_size / heads, 17]
        scaling_coeff = 1.0 / torch.sqrt(torch.tensor([self.qkv_size]).float()).to(device)
        return scaling_coeff * torch.einsum("ban, bne -> bae", self.softmax(QK_dot_prod), V)
```

Listing 1: Attention Head using torch.einsum

4.2 ViT Implementation

The implementation and the results are shown in the following pages:

ViT

April 19, 2023

[1]:
```python
# Import Libraries
import numpy as np
import torch
import torchvision.transforms as tvt
import torch.utils.data
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from torchinfo import summary
from PIL import Image
import os
from pprint import pprint
import seaborn as sns
import cv2
from ViTHelper import MasterEncoder
from einops import repeat
from einops.layers.torch import Rearrange, Reduce
import time
import datetime
```

```
/home/dfarache/.conda/envs/cent7/2020.11-py38/eceDL2/lib/python3.8/site-
packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[2]:
```python
# GLOBAL VARIABLES
train_dir = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw09/Train"
test_dir = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw09/Val"
path_to_model = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw09/model"
path_to_results = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw09/results"
batch_size = 16

num_classes = 5
class_list = ("airplane", "bus", "cat", "dog", "pizza")
class_to_integer = {"airplane": 0, "bus": 1, "cat": 2, "dog": 3, "pizza": 4}
integer_to_class = {0: "airplane", 1: "bus", 2: "cat", 3: "dog", 4: "pizza"}
```

```
patch_size = 16 # 16 pixels
C_in = 3 # Input channels
embedded_size = 128
max_seq_length = patch_size + 1 # account for class token
image_size = 64 # 64x64, h x w

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Torch is on {device}")
device = torch.device(device)
```

```
Torch is on cuda
```

0.1 Generate Dataset

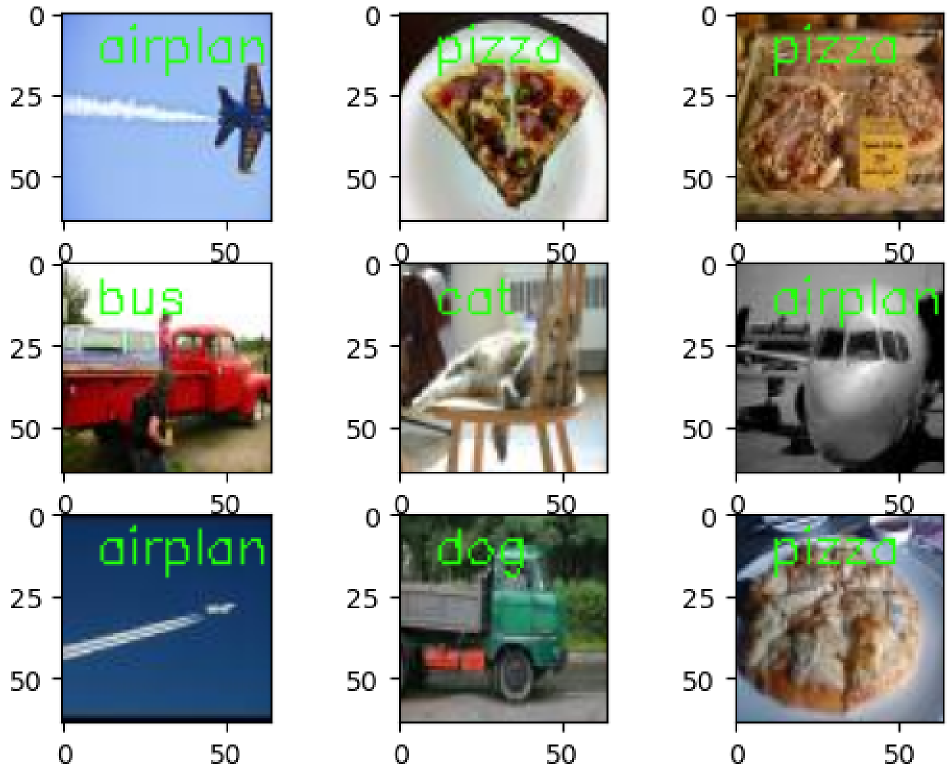
[3]:
```
def get_images(root, category):
    category_path = os.path.join(root, category)
    image_files = [image for image in os.listdir(category_path) if image != ".
↪DS_Store"]

    images_pil = [Image.open(os.path.join(category_path, image)).convert("RGB")␣
↪for image in image_files]
    return images_pil
```

[4]:
```
class GenerateDataset(torch.utils.data.Dataset):
    def __init__(self, root, class_list, transform=None):
        super().__init__()
        self.root = root
        self.class_list = class_list
        self.transform = transform
        self.data = []

        for idx, category in enumerate(self.class_list):
            images = get_images(self.root, category)
            for image in images:
                self.data.append([image, idx])

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        image = self.transform(self.data[idx][0]) if self.transform else self.
↪data[idx][0]
        label = torch.tensor(self.data[idx][1])

        return image, label
```

[5]:
```
def generate_dataloader(root_path, class_list, debug=False):
    transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.
↪5, 0.5, 0.5))])
    dataset = GenerateDataset(root_path, class_list, transform=transform)
    if(debug):
        image_size, label = dataset[0][0].shape, dataset[0][1]
        label = integer_to_class[int(label)]
        print(f"Image Shape: {image_size} and Label: {label}")

    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,␣
↪shuffle=True, num_workers=2, drop_last=True)
    return dataloader

trainloader = generate_dataloader(train_dir, class_list, debug=True)
testloader = generate_dataloader(test_dir, class_list)
```

```
Image Shape: torch.Size([3, 64, 64]) and Label: airplane
```

[6]:
```
def display_input_images(dataloader):
    fig, ax = plt.subplots(3, 3)
    row, col = 0, 0
    for batch_idx, (images_in_batch, labels_in_batch) in enumerate(dataloader):
        for idx in range(len(labels_in_batch)):
            if(col == 3):
                row += 1
                col = 0
            if(row == 3):
                break
            label = integer_to_class[int(labels_in_batch[idx])]
            image = np.asarray(tvt.ToPILImage()(images_in_batch[idx].
↪squeeze(dim=0) / 2  + 0.5))
            image = cv2.putText(image, label, (10, 15), fontFace=cv2.
↪FONT_HERSHEY_SIMPLEX, fontScale=0.5, color=(36, 255, 12), thickness=1)
            ax[row, col].imshow(image)
            col += 1
        if(row == 3):
            break

display_input_images(trainloader)
```



0.2 Networks

[7]:
```
# Inspired by: https://towardsdatascience.com/
↪implementing-visualttransformer-in-pytorch-184f9f16f632
# Inspired by: https://medium.com/mlearning-ai/
↪vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c

class PatchEmbedding(nn.Module):
    def __init__(self, patch_size, embedded_size, image_size, in_channels=3):
        super(PatchEmbedding, self).__init__()

        self.in_channels = in_channels
        self.patch_size = patch_size
        self.embedded_size = embedded_size
        self.image_size = image_size

        """
        The transformer encoder was developed with sequence data in mind but␣
↪images are not sequences. So to 'sequencify' an image we break it into
```

```
        multiple sub-images and map each sub-image to a vector. This is␣
↪achieved by reshaping our input from (batch_size, input_channel, height,␣
↪width)
        to shape (batch_size, patch_height * patch_width, embedded_size) where␣
↪embedded_size = input_channel * patch_size ** 2
        """

        self.sequential = nn.Sequential(
            # Convolutional Network is used for better performance
            nn.Conv2d(in_channels, embedded_size, kernel_size=patch_size,␣
↪stride=patch_size), # Returns a shape of (batch_size, embedded_size,␣
↪patch_height, patch_width)
            Rearrange('b e (h) (w) -> b (h w) e') # chance shape from␣
↪(batch_size, embedded_size, patch_height, patch_width) to (batch_size,␣
↪patch_height * patch_width, embedded_size)
        )

        self.class_token = nn.Parameter(torch.rand(1, 1, self.embedded_size)) #␣
↪Number placed in front of each sequence
        self.positions = nn.Parameter(torch.randn((self.image_size // self.
↪patch_size)**2 + 1, self.embedded_size)) # The positional embedding allows␣
↪the network to know where each sub-image is positioned originally in the␣
↪image

    def forward(self, x):
        x = self.sequential(x)
        class_tokens = repeat(self.class_token, '() n e -> b n e',␣
↪b=batch_size) # repeating in each batch
        x = torch.cat([class_tokens, x], dim=1) # prepend the class token to␣
↪the input make it sequence length of 17
        x = x + self.positions
        return x
```

[8]:
```
class ClassificationHead(nn.Sequential):
  # Inspired by: https://towardsdatascience.com/
↪implementing-visualttransformer-in-pytorch-184f9f16f632
    def __init__(self, embedded_size, num_classes):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.Linear(embedded_size, num_classes))
```

0.2.1 Homework

[9]:
```
class ViT(nn.Sequential):
    # Inspired by: https://towardsdatascience.com/
 ↪implementing-visualttransformer-in-pytorch-184f9f16f632
    def __init__(self, how_many_basic_encoders=2, num_attention_heads=2,␣
 ↪in_channels=3):
        super(ViT, self).__init__(
            PatchEmbedding(patch_size, embedded_size, image_size, in_channels),␣
 ↪# Returns shape [batch size, max_seq_length, embedded_size]
            MasterEncoder(max_seq_length, embedded_size,␣
 ↪how_many_basic_encoders, num_attention_heads), # Returns shape [batch size,␣
 ↪max_seq_length, embedded_size]
            ClassificationHead(embedded_size, num_classes) # Returns shape␣
 ↪[batch size, num_classes]
        )
```

[10]:
```
# Number of layers and learnable parameters in the Generator
net1 = ViT()
num_layers = len(list(net1.parameters()))
num_learnable_parameters = sum(p.numel() for p in net1.parameters() if p.
 ↪requires_grad)

print(f"Number of layers in the network: {num_layers}")
print(f"Number of learnable parameters in the network:␣
 ↪{num_learnable_parameters}")
summary(net1, input_size=(batch_size, 3, image_size, image_size))
```

```
Number of layers in the network: 46
Number of learnable parameters in the network: 52213253
```

[10]:
```
================================================================================
====================
Layer (type:depth-idx)                                 Output Shape
Param #
================================================================================
====================
ViT                                                     [16, 5]                   --
 PatchEmbedding: 1-1                                   [16, 17, 128]
2,304
     Sequential: 2-1                                   [16, 16, 128]            --
          Conv2d: 3-1                                  [16, 128, 4, 4]
98,432
          Rearrange: 3-2                               [16, 16, 128]            --
 MasterEncoder: 1-2                                    [16, 17, 128]            --
     ModuleList: 2-2                                   --                       --
          BasicEncoder: 3-3                            [16, 17, 128]
```

```
26,055,936
          BasicEncoder: 3-4                      [16, 17, 128]
26,055,936
 ClassificationHead: 1-3                          [16, 5]                  --
     Reduce: 2-3                                 [16, 128]                --
     Linear: 2-4                                 [16, 5]                  645
================================================================================
====================
Total params: 52,213,253
Trainable params: 52,213,253
Non-trainable params: 0
Total mult-adds (M): 859.00
================================================================================
====================
Input size (MB): 0.79
Forward/backward pass size (MB): 4.72
Params size (MB): 208.84
Estimated Total Size (MB): 214.35
================================================================================
====================
```

0.2.2 Extra Credit

[11]:
```python
class ViT2(nn.Sequential):
    # Inspired by: https://towardsdatascience.com/
 ↪implementing-visualttransformer-in-pytorch-184f9f16f632
    def __init__(self, how_many_basic_encoders=2, num_attention_heads=2,␣
 ↪in_channels=3):
        super(ViT2, self).__init__(
            PatchEmbedding(patch_size, embedded_size, image_size, in_channels),␣
 ↪# Returns shape [batch size, max_seq_length, embedded_size]
            MasterEncoder(max_seq_length, embedded_size,␣
 ↪how_many_basic_encoders, num_attention_heads, task=2), # Returns shape␣
 ↪[batch size, max_seq_length, embedded_size]
            ClassificationHead(embedded_size, num_classes) # Returns shape␣
 ↪[batch size, num_classes]
        )
```

[12]:
```python
# Number of layers and learnable parameters in the Generator
net2 = ViT2()
num_layers = len(list(net2.parameters()))
num_learnable_parameters = sum(p.numel() for p in net2.parameters() if p.
 ↪requires_grad)

print(f"Number of layers in the network: {num_layers}")
```

```python
print(f"Number of learnable parameters in the network:␣
↪{num_learnable_parameters}")
summary(net2, input_size=(batch_size, 3, image_size, image_size))
```

```
Number of layers in the network: 46
Number of learnable parameters in the network: 52213253
```

[12]:
```
================================================================================
====================
Layer (type:depth-idx)                                  Output Shape
Param #
================================================================================
====================
ViT2                                                     [16, 5]                  --
 PatchEmbedding: 1-1                                    [16, 17, 128]
2,304
     Sequential: 2-1                                   [16, 16, 128]            --
         Conv2d: 3-1                                   [16, 128, 4, 4]
98,432
         Rearrange: 3-2                                [16, 16, 128]            --
 MasterEncoder: 1-2                                     [16, 17, 128]            --
     ModuleList: 2-2                                   --                       --
         BasicEncoder: 3-3                             [16, 17, 128]
26,055,936
         BasicEncoder: 3-4                             [16, 17, 128]
26,055,936
 ClassificationHead: 1-3                                [16, 5]                  --
     Reduce: 2-3                                       [16, 128]                --
     Linear: 2-4                                       [16, 5]                  645
================================================================================
====================
Total params: 52,213,253
Trainable params: 52,213,253
Non-trainable params: 0
Total mult-adds (M): 859.00
================================================================================
====================
Input size (MB): 0.79
Forward/backward pass size (MB): 4.72
Params size (MB): 208.84
Estimated Total Size (MB): 214.35
================================================================================
====================
```

0.3 Training

[13]:
```
def plot_losses(loss, epochs):
    plt.plot(range(len(loss)), loss)

    plt.title(f"Loss per Iteration")
    plt.xlabel(f"Iterations over {epochs} epochs")
    plt.ylabel("Loss")
    plt.legend(loc="lower right")

    filename = "train_loss.jpg"
    plt.savefig(os.path.join(path_to_results, filename))
    plt.show()
```

[14]:
```
def train(net, epochs, lr, betas, dataloader, log=100, mode=True):
    net = net.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas)
    loss_per_iteration = []

    print(f"Training started at time {datetime.datetime.now().time()}")
    start_time = time.time()
    check_loss = float("inf")

    for epoch in range(1, epochs + 1):
        running_loss = 0.0
        for batch_idx, (inputs, labels) in enumerate(dataloader):
            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((batch_idx + 1) % log == log-1):
                print("[epoch: %d, batch: %5d] loss: %.3f" % (epoch, batch_idx␣
↪+ 1, running_loss / log))
                loss_per_iteration.append(running_loss / log)

                if(running_loss < check_loss):
                    model_name = "model1.pth" if mode else "model2.pth"
                    torch.save(net.state_dict(), os.path.join(path_to_model,␣
↪model_name))
                    check_loss = running_loss
                running_loss = 0.0
```

```
    print("Lowest loss achieved by network: %.4f" % (check_loss / float(log)))
    print("Training finished in %4d secs" % (time.time() - start_time))
    return loss_per_iteration
```

[15]:
```
# Parameters for training
lr = 1e-4 # Learning Rate
betas = (0.9, 0.999) # Betas factor
epochs = 20 # Number of epochs to train
```

0.3.1 Homework

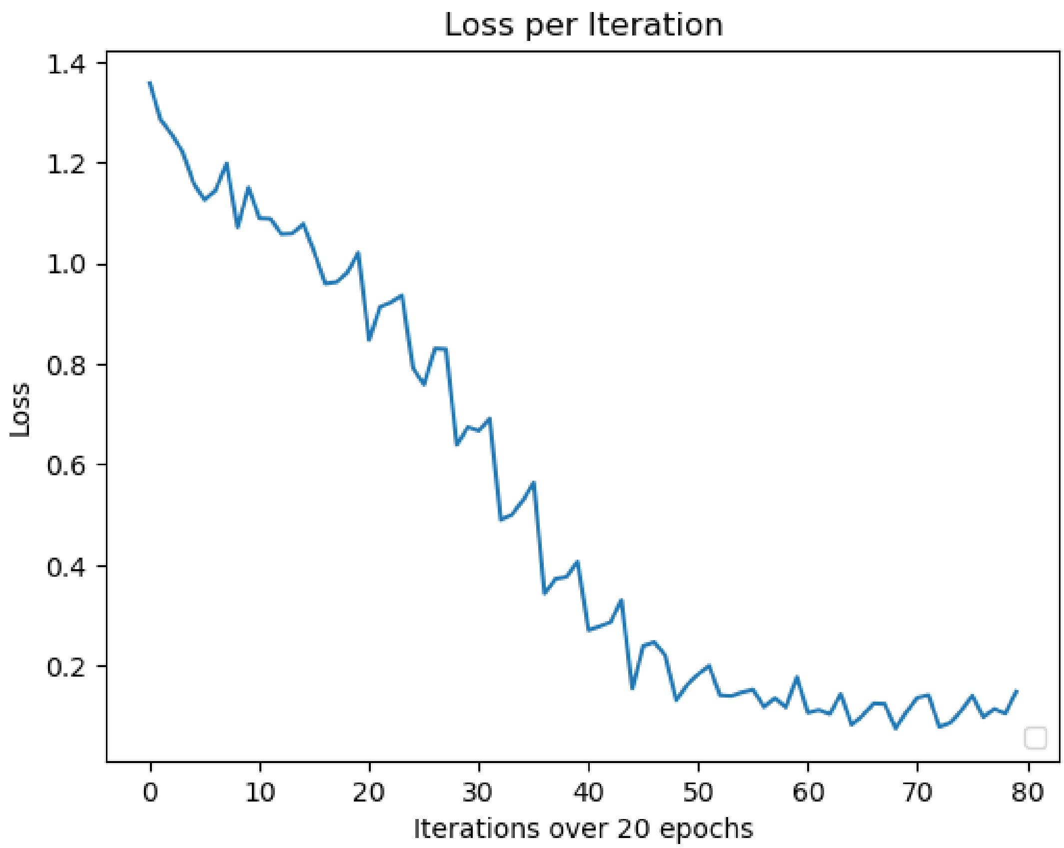
[16]:
```
training_loss = train(net1, epochs, lr, betas, trainloader)
plot_losses(training_loss, epochs)
```

```
Training started at time 01:24:23.670707
[epoch: 1, batch:    99] loss: 1.358
[epoch: 1, batch:   199] loss: 1.285
[epoch: 1, batch:   299] loss: 1.257
[epoch: 1, batch:   399] loss: 1.221
[epoch: 2, batch:    99] loss: 1.159
[epoch: 2, batch:   199] loss: 1.126
[epoch: 2, batch:   299] loss: 1.146
[epoch: 2, batch:   399] loss: 1.198
[epoch: 3, batch:    99] loss: 1.072
[epoch: 3, batch:   199] loss: 1.151
[epoch: 3, batch:   299] loss: 1.090
[epoch: 3, batch:   399] loss: 1.089
[epoch: 4, batch:    99] loss: 1.058
[epoch: 4, batch:   199] loss: 1.059
[epoch: 4, batch:   299] loss: 1.078
[epoch: 4, batch:   399] loss: 1.022
[epoch: 5, batch:    99] loss: 0.959
[epoch: 5, batch:   199] loss: 0.962
[epoch: 5, batch:   299] loss: 0.981
[epoch: 5, batch:   399] loss: 1.021
[epoch: 6, batch:    99] loss: 0.848
[epoch: 6, batch:   199] loss: 0.913
[epoch: 6, batch:   299] loss: 0.923
[epoch: 6, batch:   399] loss: 0.936
[epoch: 7, batch:    99] loss: 0.791
[epoch: 7, batch:   199] loss: 0.759
[epoch: 7, batch:   299] loss: 0.830
[epoch: 7, batch:   399] loss: 0.829
[epoch: 8, batch:    99] loss: 0.639
[epoch: 8, batch:   199] loss: 0.674
[epoch: 8, batch:   299] loss: 0.667
[epoch: 8, batch:   399] loss: 0.691
```

```
[epoch: 9, batch:    99] loss: 0.490
[epoch: 9, batch:   199] loss: 0.500
[epoch: 9, batch:   299] loss: 0.528
[epoch: 9, batch:   399] loss: 0.564
[epoch: 10, batch:    99] loss: 0.343
[epoch: 10, batch:   199] loss: 0.372
[epoch: 10, batch:   299] loss: 0.376
[epoch: 10, batch:   399] loss: 0.407
[epoch: 11, batch:    99] loss: 0.270
[epoch: 11, batch:   199] loss: 0.278
[epoch: 11, batch:   299] loss: 0.286
[epoch: 11, batch:   399] loss: 0.330
[epoch: 12, batch:    99] loss: 0.154
[epoch: 12, batch:   199] loss: 0.239
[epoch: 12, batch:   299] loss: 0.246
[epoch: 12, batch:   399] loss: 0.220
[epoch: 13, batch:    99] loss: 0.130
[epoch: 13, batch:   199] loss: 0.162
[epoch: 13, batch:   299] loss: 0.183
[epoch: 13, batch:   399] loss: 0.199
[epoch: 14, batch:    99] loss: 0.140
[epoch: 14, batch:   199] loss: 0.139
[epoch: 14, batch:   299] loss: 0.147
[epoch: 14, batch:   399] loss: 0.152
[epoch: 15, batch:    99] loss: 0.118
[epoch: 15, batch:   199] loss: 0.135
[epoch: 15, batch:   299] loss: 0.117
[epoch: 15, batch:   399] loss: 0.177
[epoch: 16, batch:    99] loss: 0.106
[epoch: 16, batch:   199] loss: 0.112
[epoch: 16, batch:   299] loss: 0.103
[epoch: 16, batch:   399] loss: 0.143
[epoch: 17, batch:    99] loss: 0.082
[epoch: 17, batch:   199] loss: 0.101
[epoch: 17, batch:   299] loss: 0.125
[epoch: 17, batch:   399] loss: 0.123
[epoch: 18, batch:    99] loss: 0.075
[epoch: 18, batch:   199] loss: 0.108
[epoch: 18, batch:   299] loss: 0.136
[epoch: 18, batch:   399] loss: 0.141
[epoch: 19, batch:    99] loss: 0.078
[epoch: 19, batch:   199] loss: 0.086
[epoch: 19, batch:   299] loss: 0.110
[epoch: 19, batch:   399] loss: 0.140
[epoch: 20, batch:    99] loss: 0.097
[epoch: 20, batch:   199] loss: 0.114
[epoch: 20, batch:   299] loss: 0.105
[epoch: 20, batch:   399] loss: 0.148
```

```
No handles with labels found to put in legend.
```

```
Lowest loss achieved by network: 0.0748
Training finished in  268 secs
```



0.3.2 Extra Credit

[17]:
```python
training_loss = train(net2, epochs, lr, betas, trainloader, mode=False)
plot_losses(training_loss, epochs)
```

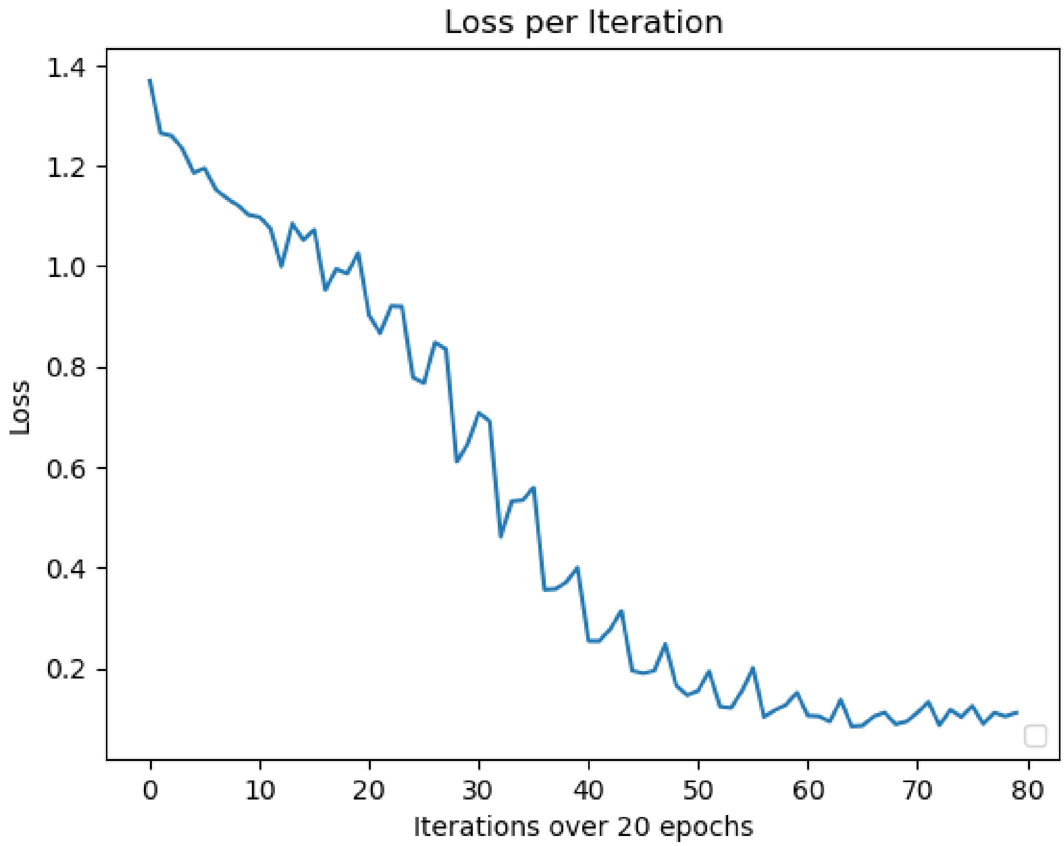
```
Training started at time 01:28:52.955506
[epoch: 1, batch:    99] loss: 1.369
[epoch: 1, batch:   199] loss: 1.265
[epoch: 1, batch:   299] loss: 1.259
[epoch: 1, batch:   399] loss: 1.233
[epoch: 2, batch:    99] loss: 1.185
[epoch: 2, batch:   199] loss: 1.195
[epoch: 2, batch:   299] loss: 1.153
[epoch: 2, batch:   399] loss: 1.135
[epoch: 3, batch:    99] loss: 1.122
[epoch: 3, batch:   199] loss: 1.102
```

```
[epoch: 3, batch:   299] loss: 1.098
[epoch: 3, batch:   399] loss: 1.076
[epoch: 4, batch:    99] loss: 0.999
[epoch: 4, batch:   199] loss: 1.085
[epoch: 4, batch:   299] loss: 1.052
[epoch: 4, batch:   399] loss: 1.073
[epoch: 5, batch:    99] loss: 0.953
[epoch: 5, batch:   199] loss: 0.994
[epoch: 5, batch:   299] loss: 0.985
[epoch: 5, batch:   399] loss: 1.026
[epoch: 6, batch:    99] loss: 0.902
[epoch: 6, batch:   199] loss: 0.866
[epoch: 6, batch:   299] loss: 0.920
[epoch: 6, batch:   399] loss: 0.919
[epoch: 7, batch:    99] loss: 0.778
[epoch: 7, batch:   199] loss: 0.767
[epoch: 7, batch:   299] loss: 0.848
[epoch: 7, batch:   399] loss: 0.835
[epoch: 8, batch:    99] loss: 0.610
[epoch: 8, batch:   199] loss: 0.647
[epoch: 8, batch:   299] loss: 0.708
[epoch: 8, batch:   399] loss: 0.691
[epoch: 9, batch:    99] loss: 0.461
[epoch: 9, batch:   199] loss: 0.532
[epoch: 9, batch:   299] loss: 0.534
[epoch: 9, batch:   399] loss: 0.559
[epoch: 10, batch:    99] loss: 0.356
[epoch: 10, batch:   199] loss: 0.357
[epoch: 10, batch:   299] loss: 0.371
[epoch: 10, batch:   399] loss: 0.400
[epoch: 11, batch:    99] loss: 0.254
[epoch: 11, batch:   199] loss: 0.254
[epoch: 11, batch:   299] loss: 0.278
[epoch: 11, batch:   399] loss: 0.314
[epoch: 12, batch:    99] loss: 0.194
[epoch: 12, batch:   199] loss: 0.190
[epoch: 12, batch:   299] loss: 0.194
[epoch: 12, batch:   399] loss: 0.248
[epoch: 13, batch:    99] loss: 0.165
[epoch: 13, batch:   199] loss: 0.146
[epoch: 13, batch:   299] loss: 0.155
[epoch: 13, batch:   399] loss: 0.194
[epoch: 14, batch:    99] loss: 0.123
[epoch: 14, batch:   199] loss: 0.121
[epoch: 14, batch:   299] loss: 0.154
[epoch: 14, batch:   399] loss: 0.200
[epoch: 15, batch:    99] loss: 0.103
[epoch: 15, batch:   199] loss: 0.116
```

```
[epoch: 15, batch:   299] loss: 0.127
[epoch: 15, batch:   399] loss: 0.151
[epoch: 16, batch:    99] loss: 0.105
[epoch: 16, batch:   199] loss: 0.104
[epoch: 16, batch:   299] loss: 0.093
[epoch: 16, batch:   399] loss: 0.137
[epoch: 17, batch:    99] loss: 0.084
[epoch: 17, batch:   199] loss: 0.085
[epoch: 17, batch:   299] loss: 0.103
[epoch: 17, batch:   399] loss: 0.112
[epoch: 18, batch:    99] loss: 0.088
[epoch: 18, batch:   199] loss: 0.093
[epoch: 18, batch:   299] loss: 0.111
[epoch: 18, batch:   399] loss: 0.132
[epoch: 19, batch:    99] loss: 0.086
[epoch: 19, batch:   199] loss: 0.117
[epoch: 19, batch:   299] loss: 0.102
[epoch: 19, batch:   399] loss: 0.125
[epoch: 20, batch:    99] loss: 0.088
[epoch: 20, batch:   199] loss: 0.111
[epoch: 20, batch:   299] loss: 0.104
[epoch: 20, batch:   399] loss: 0.111
```

```
No handles with labels found to put in legend.
```

```
Lowest loss achieved by network: 0.0839
Training finished in  296 secs
```



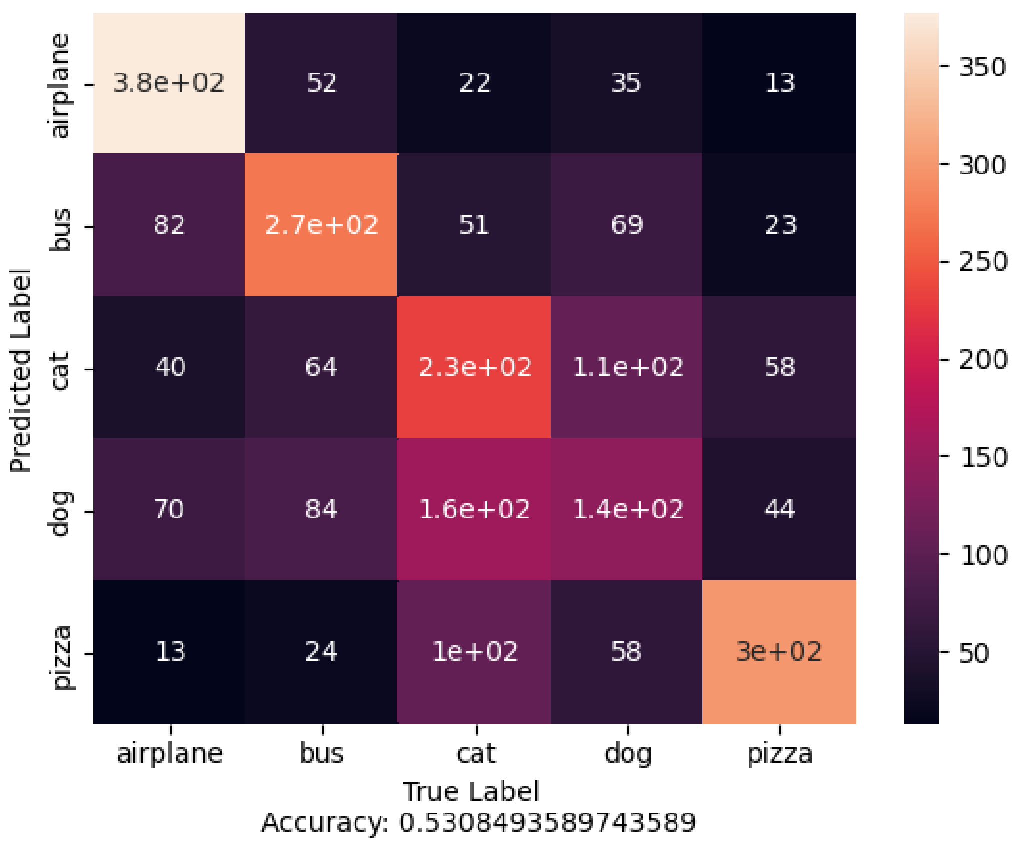
0.4 Testing

[18]:
```
def test(net, path_to_network, dataloader, num_classes, mode=True):
    model_name = "model1.pth" if mode else "model2.pth"
    net.load_state_dict(torch.load(os.path.join(path_to_network, model_name)))
    net = net.to(device)
    confusion_matrix = np.zeros((num_classes, num_classes))

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = net(inputs)
            _, predicted = torch.max(outputs, dim=1)
            for label, prediction in zip(labels, predicted):
                confusion_matrix[label][prediction] += 1

    accuracy = np.trace(confusion_matrix) / np.sum(confusion_matrix)
    return confusion_matrix, accuracy
```

[19]:
```
def display_confusion_matrix(conf, class_list, accuracy):
    sns.heatmap(conf, xticklabels=class_list, yticklabels=class_list,␣
 ↪annot=True)
    plt.xlabel(f"True Label \n Accuracy: {accuracy}")
    plt.ylabel("Predicted Label")

    filename = "conf.jpg"
    plt.savefig(os.path.join(path_to_results, filename))
    plt.show()
```

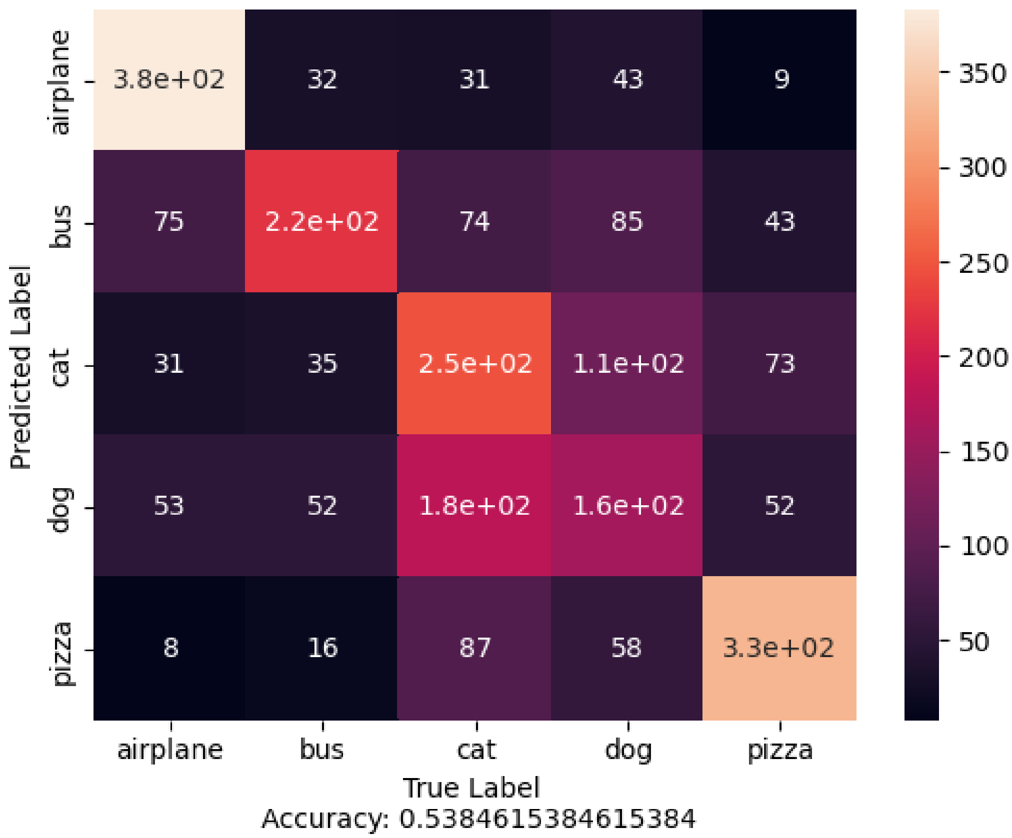
0.4.1 Homework

[20]:
```
confusion_matrix, accuracy = test(net1, path_to_model, testloader, num_classes)
display_confusion_matrix(confusion_matrix, class_list, accuracy)
```



0.4.2 Extra Credit

[21]:
```
confusion_matrix, accuracy = test(net2, path_to_model, testloader, num_classes,␣
↪mode=False)
display_confusion_matrix(confusion_matrix, class_list, accuracy)
```



5 Evaluation

After implementing the Vision Transformer (ViT) and trained the network for 20 epochs, I achieved an accuracy of at least 53% whereas with the CNN network, the accuracy I achieved was at least 95%. This is probably a result of using a lower embedding size than the recommended size in the *Attention Is All You Need* paper or the hyper-parameters must be tweaked to obtain better results. But as of right now the ViT did underperform in image classification in comparison to DCNN.

References

[1] Maxime, "What is a Transformer", *in Medium*, 2019, URL: `https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04`.

[2] V. et al., "Attention Is All You Need", *in Association for Computing Machinery*, 2017, URL: `https://arxiv.org/pdf/1706.03762.pdf`.

[3] A. Kak, "Transformers", *in Purdue ECE*, 2023, URL: `https://engineering.purdue.edu/DeepLearn/pdf-kak/Transformers.pdf`.

[4] Y. Tamura, "Multi-head attention mechanism: "queries", "keys", and "values," over and over again", *in Data Science Blog*, 2021, URL: `https://data-science-blog.com/blog/2021/04/07/multi-head-attention-mechanism/`.

[5] M. Phi, "Illustrated Guide to Transformers - Step by Step Explanation", *in Towards Datascience*, 2020, URL: `https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explana`

[6] W. et al., "Vision Transformers for Computer Vision", *in Deep GAN Team*, 2021, URL: `https://deepganteam.medium.com/vision-transformers-for-computer-vision-9f70418fe41a:`