HW 8: Gating mechanisms in RNN

David E. Farache, Email ID: dfarache@purdue.edu

April 30, 2023

1 Introduction

For this assignment, we focused on implementing a Vision Transformer (ViT) using a multi-headed self-attention mechanism and transformer architecture for image classification.

2 Theory/Methods

2.1 Scaled Dot Product Attention

Scaled dot product attention is the method utilized to transfer an input sequence into separate relevant vector values to determine importance. These tensors are Q, all query-based vectors into a single data object, K, word-based key vectors, and V, value vectors. The attention is calculated as tensor V being modified by tensors Q and K as seen below:

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

In this case, $\sqrt{d_k}$ normalizes the dot product of Q and K^T and softmax turns those normalized values into weights that are then multiplied to V, giving the attention value.

2.2 Multi-Headed Attention

In the case of multi-headed attention, the input tensor is first split along its embedded axis into N slices. Each section is then processed as if it were single-headed, going through scale product attention, with individual Q, K, and V tensors. Each partition, after returning values after applying equation 1, is then concatenated and passed through a fully connected linear layer.

2.3 Transformer

Transformers function by adopting a mechanism of self-attention, then weighting the significance of each part of the input. It does this via an architecture of stacked encoders and decoders. The encoders create an attention map for sentence sequence fed into it while the decoder translates, based on the prior attention map, to a designated sentence.

The encoder structure, in this case, is a 6-layer structure with 3 different sections of 2-layer sections. The first section of the three applies the multi-head attention layer described previously and the section after applies the position-wise fully connected feed-forward network. The final

section is connected with all layers via a residual connection and which passes the information through a normalization layer.

The decoder architecture is the same as the encoder, a 6-layer structure with 3 sub-layers that each have 2 layers. These layers mirror the structure of the encoder. With the first layer being a self-attention layer, the second being a position-wise fully connected layer, and the final layer being a multi-headed attention layer.

This logic, so far, for the transformer has been focused on the assumption that the data being fed into are sentences or sequences. For ViT the issue is that we need to process images which requires transferring image data into sequence data. This is achieved via Patch Embedding, which functions by slicing the image that is then condensed into data vectors. This dome via reshaping and positional information is included to enable patch identification. Furthermore, tensors have an input sequence pre-pended to store information from one layer to another and create deeper layers.

3 ViTHelper

For the assignment, the ViTHelper python library from DLStudio was utilized for the development of Transformers and creating the ViT. For the extra credit assignment, the function AttentionHead was copied and updated to add torch.einsum as seen in the code below:

```
class AttentionHead_ExtraCredit(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):
        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)

        QK_DotProduct = torch.einsum("bqn, bnk -> bqk", Q, rearrange(K, "b n e
    -> b e n"))
        ScaleCoeff = 1.0 / torch.sqrt(torch.tensor([self.qkv_size]).float()).to(
    device)
```

```
        return ScaleCoeff * torch.einsum("ban, bne -> bae", self.softmax(
    QK_DotProduct), V)
```

Listing 1: *ViT Extra Credit*

4 Setup Code

```
# 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 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

device = "cuda"
device = torch.device("cuda")

# GLOBAL VARIABLES
categories = ["airplane", "bus", "cat", "dog", "pizza"]
num_classes = len(categories)
class_encoding = {0: "airplane", 1: "bus", 2: "cat", 3: "dog", 4: "pizza"}

patch_size = 16 # pixels
C_in = 3
embedded_size = 64
max_seq_length = patch_size + 1 # class token
image_size = 64 # h x w

images_Train = '/scratch/gilbreth/dfarache/ece60146/David/HW9/trainingData'
images_Val = '/scratch/gilbreth/dfarache/ece60146/David/HW9/valData'

#DataLoader
def ImageProcessing(directory, cat):
    dir = os.path.join(directory, cat)

    imageFile = [image for image in os.listdir(dir)]
    imagesPIL = [Image.open(os.path.join(dir, image)).convert("RGB") for image
    in imageFile]

    toTensor = [tvt.ToTensor()(image) for image in imagesPIL]
    toNormalize = [tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))(tensor) for
    tensor in toTensor]
    return toNormalize
```

```

class MyDataset(torch.utils.data.Dataset):

    def __init__(self, root, categories):
        super().__init__()
        self.categories = categories
        self.root = root
        self.data = []

        for i, cat in enumerate(self.categories):
            images = ImageProcessing(self.root, cat)
            for image in images:
                self.data.append([image, i])

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

    def __getitem__(self, i):
        image = self.data[i][0]
        label = torch.tensor(self.data[i][1])
        return image, label
```

Listing 2: *Setup Code*

4.1 Task 1: Network

For this work, the batch size is 16, and the parameters of the network are a learning rate of 1e-4 and trained for 20 epochs. The difference between the given assignment and extra credit is using the prior code of AttentionHead_ExtraCredit vs AttentionHead_Assignment.

```
# Network
# Based on https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
# Based on 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

        self.sequential = nn.Sequential(
            # batch_size, embedded_size, patch_height, patch_width
            nn.Conv2d(in_channels, embedded_size, kernel_size=patch_size, stride=patch_size),

            # prior to batch_size, patch_height * patch_width, embedded_size
            Rearrange('b e (h) (w) -> b (h w) e')
        )

        self.class_token = nn.Parameter(torch.rand(1, 1, self.embedded_size))
```

```
        self.positions = nn.Parameter(torch.randn((self.image_size // self.
    patch_size)**2 + 1, self.embedded_size))

    def forward(self, x):
        x = self.sequential(x)
        # Repeat batch
        class_tokens = repeat(self.class_token, '() n e -> b n e', b=batch_size)
        x = torch.cat([class_tokens, x], dim=1) # prepend token to match
    sequence len
        x = x + self.positions
        return x

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

# Assignment Network
# Based on https://towardsdatascience.com/implementing-visualttransformer-in-
    pytorch-184f9f16f632
class ViT_Assignment(nn.Sequential):
    def __init__(self, how_many_basic_encoders=2, num_attention_heads=2,
    in_channels=3):
        super(ViT_Assignment, self).__init__(
            PatchEmbedding(patch_size, embedded_size, image_size, in_channels),
            MasterEncoder(max_seq_length, embedded_size, how_many_basic_encoders
    , num_attention_heads),
            ClassificationHead(embedded_size, num_classes)
        )

# Extra Credit Network
# Based on https://towardsdatascience.com/implementing-visualttransformer-in-
    pytorch-184f9f16f632
class ViT_ExtraCredit(nn.Sequential):
    def __init__(self, how_many_basic_encoders=2, num_attention_heads=2,
    in_channels=3):
        super(ViT_ExtraCredit, self).__init__(
            PatchEmbedding(patch_size, embedded_size, image_size, in_channels),
            MasterEncoder(max_seq_length, embedded_size, how_many_basic_encoders
    , num_attention_heads, method=2),
            ClassificationHead(embedded_size, num_classes)
        )
```

Listing 3: *Network*

4.2 Training, Testing, and Plotting

```
# Training
def train(net, epochs, lr, dataloader):
    net = net.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=(0.9, 0.999))
    lossRun = []
```

```
    for epoch in range(1, epochs+1):
        running_loss = 0.0
        for i, data in enumerate(dataloader):

            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) % 100 == 0):
                print("[epoch: %d, batch: %5d] loss: %.3f" % (epoch, i + 1,
   running_loss / 100))
                lossRun.append(running_loss / 100)
                running_loss = 0.0

    return lossRun
```

Listing 4: *Training*

```
# Testing
def test(net, dataloader, numCat):
    net = net.to(device)
    confusion_matrix = np.zeros((numCat, numCat))

    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
```

Listing 5: *Testing*

```
# Plotting
def trainingPlot(loss, epochs):
    plt.plot(range(len(loss)), loss)

    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.legend(loc="lower right")
    plt.show()

def confusionMatrix(conf, categories, acc):
    sns.heatmap(conf, xticklabels=categories, yticklabels=categories, annot=True
   )
    plt.xlabel(f"True Label \n Accuracy: {acc}")
    plt.ylabel("Predict Label")
```

```
    plt.show()
```

Listing 6: *Plotting*

5 Results

```
# Main
## DataLoaders

batch_size = 16

# Create DataLoaders
trainDataset = MyDataset(images_Train, categories)
trainDataloader = torch.utils.data.DataLoader(trainDataset, batch_size=
    batch_size, shuffle=True, num_workers=2, drop_last=True)

testDataset = MyDataset(images_Val, categories)
testDataloader = torch.utils.data.DataLoader(testDataset, batch_size=batch_size,
     shuffle=True, num_workers=2, drop_last=True)

## Parameters for Training
lr = 1e-4 # Learning Rate
epochs = 20 # Epochs

## Assignment
net1 = ViT_Assignment()
training_loss = train(net1, epochs, lr, trainDataloader)
trainingPlot(training_loss, epochs)

confusion_matrix, accuracy = test(net1, testDataloader, num_classes)
confusionMatrix(confusion_matrix, categories, accuracy)

## Extra Credit
net2 = ViT_ExtraCredit()
training_loss = train(net2, epochs, lr, trainDataloader)
trainingPlot(training_loss, epochs)

confusion_matrix, accuracy = test(net2, testDataloader, num_classes)
confusionMatrix(confusion_matrix, categories, accuracy)
```

Listing 7: *Main To Run Code*

As seen in Figures 1 and 2, the loss per iteration for both the original assignment and extra credit have the same loss value decrease and plateau. In terms of accuracy, the extra credit slightly outperformed the original assignment code with 54% beginning greater than 53%. Both performed less than the homework 4 assignment with that being 79% greater than 54% and 53%. Overall the DCNN performed better than the ViT model created in this case for both extra credit and assignment-based tasks. This could be due to low embedding size which had to be used due to lack of GPU space.

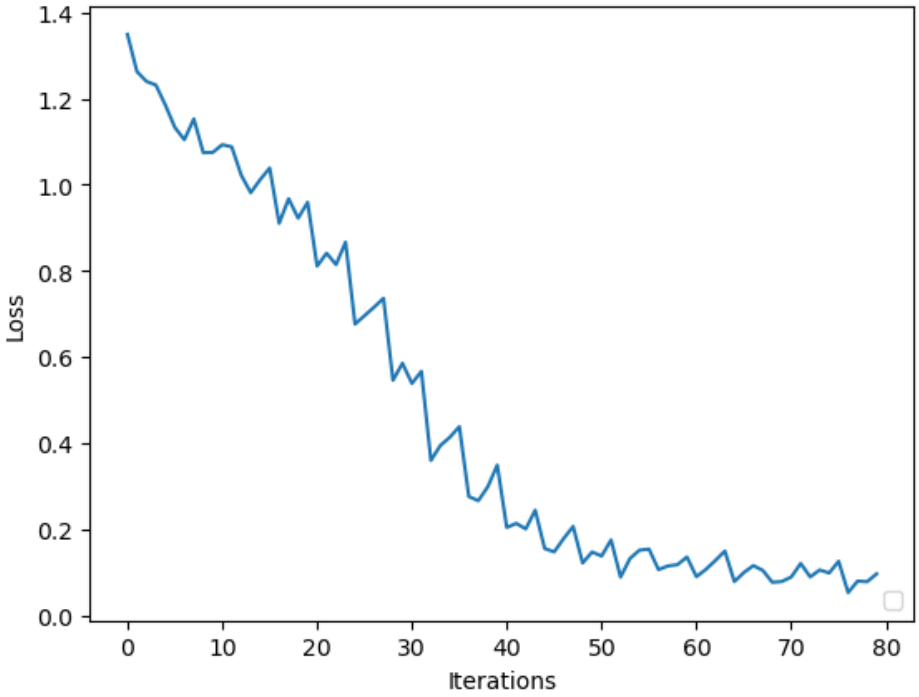

Figure 1: *Assignment Loss Plot*

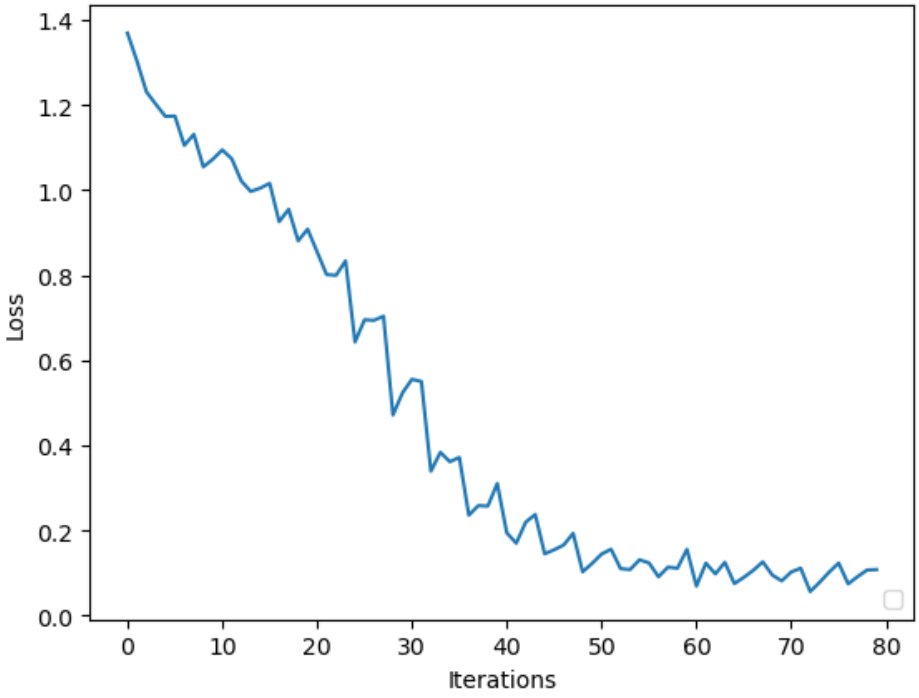

Figure 2: *Extra Credit Loss Plot*

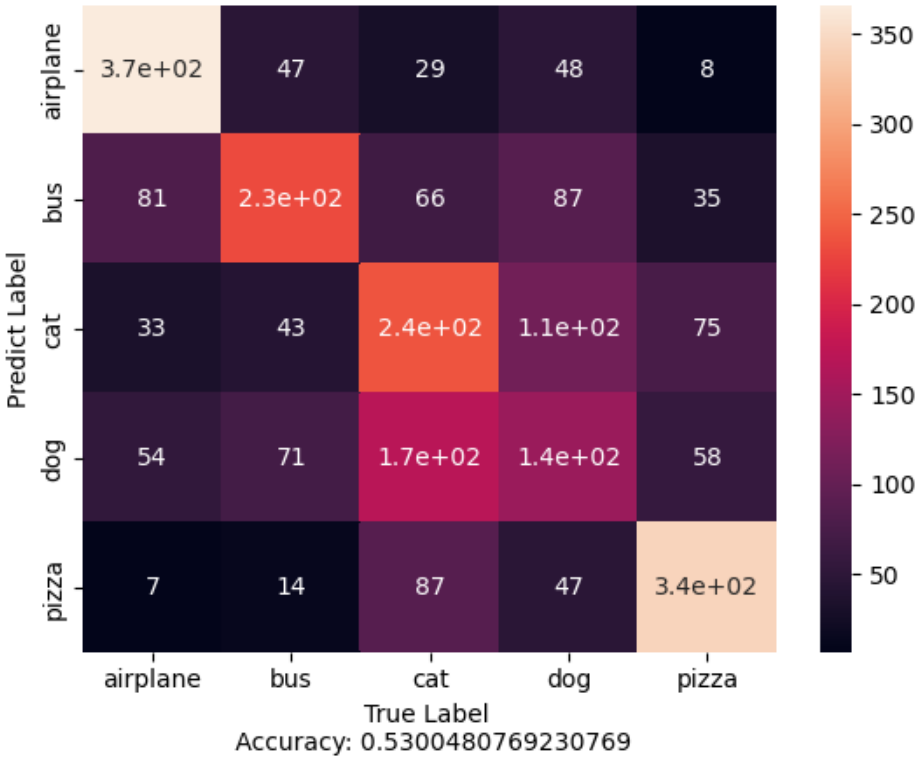

Figure 3: *Assignment Confusion Matrix*

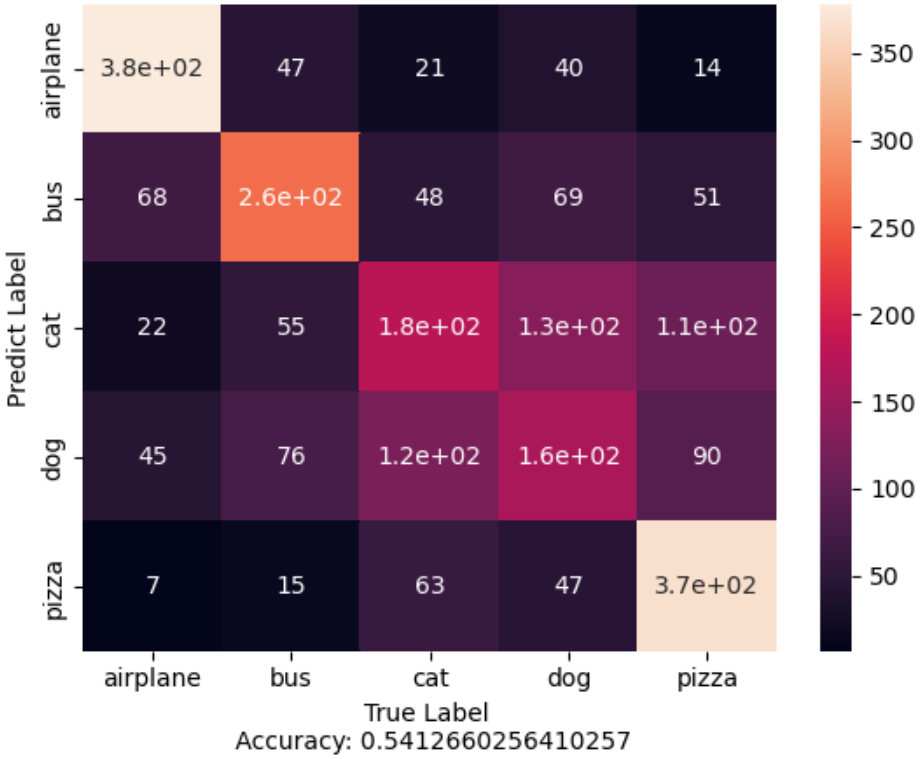

Figure 4: *Extra Credit Confusion Matrix*

6 Lessons learned

What has been learned is the utilization of a transformer and the functions of attention mechanisms.