hw9_ChengjunGuo

Chengjun Guo

April 2023

1 Multihead attention block

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size = 768, num_heads = 8, dropout = 0):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        # separate into q k v
        self.qkv = nn.Linear(emb_size, emb_size * 3)


    def forward(self, x):
        # batch N_W M*3 -> 3 batch N_H N_W s_qkv
        queries, keys, values = rearrange(self.qkv(x), "b n (h d qkv) -> (qkv) b h n d", h=self.num_heads, qkv=3)
        # calculate filter Matrix multiplication Q @ K ,batch N_H N_W s_qkv -> batch N_H N_W N_W
        filter = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
        #softmax 1 to -1
        x = F.softmax(filter, dim=-1)
        # Matrix multiplication X @ V batch N_H N_W N_W @ batch N_H N_W s_qkv -> batch N_H N_W s_qkv
        out = torch.einsum('bhal, bhlv -> bhav ', x, values)
        #batch N_H N_W s_qkv -> batch N_H M, normalize
        out = rearrange(out, "b h n d -> b n (h d)") / (self.emb_size ** (1 / 2))
        return out
```

Listing 1: Code block

2 ViT implementation

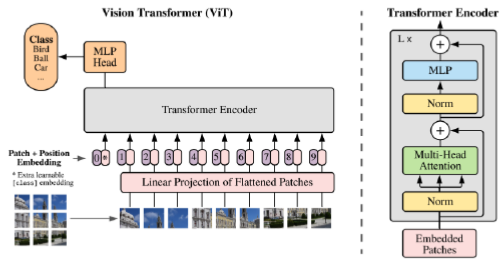

Figure 1: ViT structure

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels = 3, patch_size = 16, emb_size = 768, img_size = 224):
        super().__init__()
        self.patch_size = patch_size
        self.proj = nn.Sequential(
            # takes in batch * 3 channel * 64 * 64 images, generate batch * (16*16*3) * 4*4
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size, stride=patch_size),
            # takes in batch * (16*16*3) * 4*4, convert to batch * (4*4) * (16*16*3)
            Rearrange('b c (h) (w) -> b (h w) c'),
        )
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size))
        self.positions = nn.Parameter(torch.randn((img_size // patch_size) ** 2 + 1, emb_size))

    def forward(self, x):
        batch_size = x.shape[0]
        x = self.proj(x)
        # convert cls token to batch, 1, emb_size
        cls_tokens = self.cls_token.repeat(batch_size, 1, 1)
        x = torch.cat([cls_tokens, x], dim=1)
        #position embedding
        x += self.positions
        return x


class ClassificationHead(nn.Sequential):
    def __init__(self, emb_size = 768, n_classes = 5):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.LayerNorm(emb_size),
            nn.Linear(emb_size, n_classes))


class ViT(nn.Sequential):
    def __init__(self,in_channels = 3,patch_size = 16,emb_size = 768,img_size = 64,depth = 2,n_classes = 5):
```

```
        super(ViT, self).__init__(PatchEmbedding(in_channels,
    patch_size, emb_size, img_size),MasterEncoder(max_seq_length =
    17, embedding_size = emb_size, how_many_basic_encoders = depth,
     num_atten_heads = depth),ClassificationHead(emb_size,
    n_classes))
```

Listing 2: Code block

The encoding block is as provided with 6ViThelper. For my implementation, the structure is constructed with three blocks: PatchEmbedding, MasterEncoder and then the ClassificationHead. In my ViT class, it is basically concatenating the three blocks.

The image is initially embedded with PatchEmbedding block. The 64×64 image is divided into 16 16×16 patches. One more class token is appended to the patch sequence which results in 17 sequence length. According to the ViT paper, the position embedding is added to the sequence.

The tensor is passed into the transformer encoder and then into my last classification block. The last block is just simply a header that fit into the probability of 5 classes.

3 Training loss

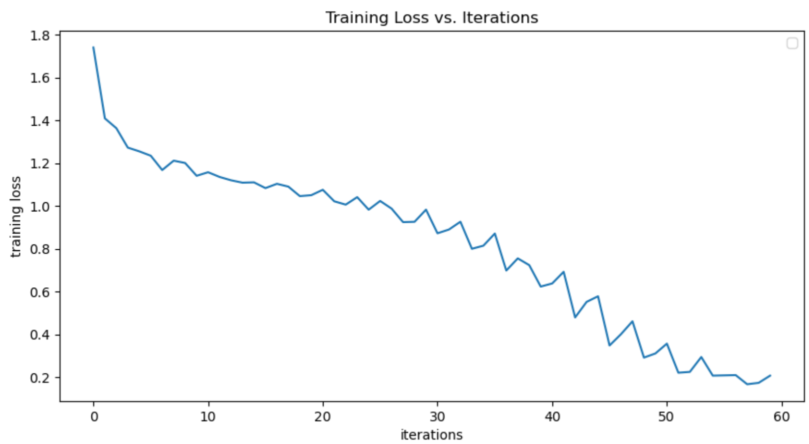

Figure 2: ViT training loss 20 epochs

4 Confusion matrix of ViT

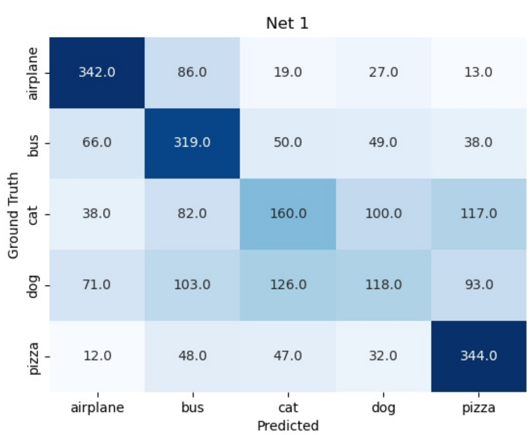

Figure 3: Confusion Matrix of ViT

5 Comparison with hw4 CNN performance

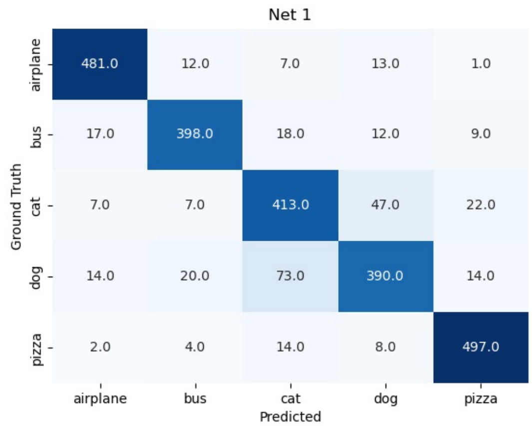

Figure 4: Confusion Matrix of CNN

Compared to CNN-based network, ViT does not have good performance. The possible reason is that the data is not enough. ViT normally need large amount of data.

6 Code

ViThelper:

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

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

import torch
import torch.nn as nn
import torch.nn.functional as F
from einops import rearrange, reduce, repeat

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# device = torch.device("cpu")
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):
```

```python
        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.self_attention_layer = MultiHeadAttention(emb_size = embedding_size, num_heads = num_atten_heads)
        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


class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size = 768, num_heads = 8):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        # separate into q k v
        self.qkv = nn.Linear(emb_size, emb_size * 3)


    def forward(self, x):
        # batch N_W M*3 -> 3 batch N_H N_W s_qkv
        queries, keys, values = rearrange(self.qkv(x), "b n (h d qkv) -> (qkv) b h n d", h=self.num_heads, qkv=3)
        # calculate filter Matrix multiplication Q @ K ,batch N_H N_W s_qkv -> batch N_H N_W N_W
        filter = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
        #softmax 1 to -1
        x = F.softmax(filter, dim=-1)
        # Matrix multiplication X @ V batch N_H N_W N_W @ batch N_H N_W s_qkv -> batch N_H N_W s_qkv
        out = torch.einsum('bhal, bhlv -> bhav ', x, values)
        #batch N_H N_W s_qkv -> batch N_H M, normalize
        out = rearrange(out, "b h n d -> b n (h d)") / (self.emb_size ** (1 / 2))
        return out
```

hw9:

```
##           https://engineering.purdue.edu/kak/distDLS/
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
import torch
from PIL import Image
from torch.utils.data import DataLoader, Dataset
import copy
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import numpy as np
import sys
import random
import pickle
import time
import torch.nn as nn
import torchvision
import torchvision.transforms as tvt
import torch.nn.functional as F
from ViTHelper import *
```

```
from einops.layers.torch import Rearrange, Reduce
from torchsummary import summary
from pycocotools.coco import COCO


class PatchEmbedding(nn.Module):
    def __init__(self, in_channels = 3, patch_size = 16, emb_size = 768, img_size = 224):
        super().__init__()
        self.patch_size = patch_size
        self.proj = nn.Sequential(
            # takes in batch * 3 channel * 64 * 64 images, generate batch * (16*16*3) * 4*4
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size, stride=patch_size),
            # takes in batch * (16*16*3) * 4*4, convert to batch * (4*4) * (16*16*3)
            Rearrange('b c (h) (w) -> b (h w) c'),
        )
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size))
        self.positions = nn.Parameter(torch.randn((img_size // patch_size) ** 2 + 1, emb_size))

    def forward(self, x):
        batch_size = x.shape[0]
        x = self.proj(x)
        # convert cls token to batch, 1, emb_size
        cls_tokens = self.cls_token.repeat(batch_size, 1, 1)
        x = torch.cat([cls_tokens, x], dim=1)
        #position embedding
        x += self.positions
        return x


class ClassificationHead(nn.Sequential):
    def __init__(self, emb_size = 768, n_classes = 5):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.LayerNorm(emb_size),
            nn.Linear(emb_size, n_classes))


class ViT(nn.Sequential):
    def __init__(self,in_channels = 3,patch_size = 16,emb_size = 768,img_size = 64,depth = 2,n_classes = 5):
        super(ViT, self).__init__(PatchEmbedding(in_channels, patch_size, emb_size, img_size),MasterEncoder(max_seq_length = 17, embedding_size = emb_size, how_many_basic_encoders = depth, num_atten_heads = depth),ClassificationHead(emb_size, n_classes))





def run_training(net,train_data_loader,net_save_path):
    net = net.to(device)
```

```
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3,betas=(0.9, 0.99), eps = 1e-4)
    epochs = 20
    Loss_runtime = []
    for epoch in range(epochs):
        running_loss = 0.0
        for i,data in enumerate(train_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) % 500 == 0:
                print("[epoch: %d, batch: %5d] loss: %.3f" % (epoch+1, i+1, running_loss/500))
                Loss_runtime.append(running_loss/500)
                running_loss = 0.0
    torch.save(net, net_save_path)
    return Loss_runtime


def run_testing(net, validation_data_loader):
    net = copy.deepcopy(net)
    net = net.to(device)
    Confusion_Matrix = torch.zeros(5, 5)
    for i, data in enumerate(validation_data_loader):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        predicted = predicted.tolist()
        for label, prediction in zip(labels, predicted):
            Confusion_Matrix[label][prediction] += 1
    return Confusion_Matrix


class MyDataset(torch.utils.data.Dataset):
    def __init__(self):
        super().__init__()
        self.root_path = 'E:\ECE60146DL\hw4_new\data/'
        self.coco_json_path = 'E:/ECE60146DL/hw4_new/annotations_trainval2014/annotations/instances_train2014.json'
        self.class_list = ['airplane','bus','cat','dog','pizza']
        self.images_per_class = 2000    # 1500 for train, 500 for validation
        self.coco = COCO(self.coco_json_path)
        self.img_labels = []
        self.transform = tvt.Compose([tvt.ToTensor(),tvt.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])
        for cat in self.class_list:
            path, dirs, files = next(os.walk(self.root_path + cat))
```

```
            for file in files:
                # first image path, second label
                self.img_labels.append([cat + '/' + file, self.class_list.index(cat)])


    def __len__(self):
        return int(self.images_per_class * len(self.class_list))

    def __getitem__(self, index):
        img_path = os.path.join(self.root_path, self.img_labels[index][0])
        image = Image.open(img_path).convert("RGB")
        label = torch.tensor(self.img_labels[index][1])

        image = self.transform(image)
        return image, label


if __name__ == '__main__':
    if torch.cuda.is_available() == True:
        device = torch.device("cuda:0")
    else:
        device = torch.device("cpu")
    # device = torch.device("cpu")
    vit = ViT()
    # summary(vit.to("cuda"),(3,64,64),device="cuda")

    my_dataset = MyDataset()
    train_dataset, test_dataset = torch.utils.data.random_split(my_dataset, [7500, 2500])
    train_data_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=4, shuffle=True, num_workers=4)
    loss1 = run_training(vit, train_data_loader, "net1.pth")
    plt.figure(figsize=(10, 5))
    plt.title("Training Loss vs. Iterations")
    plt.plot(loss1)
    plt.xlabel("iterations")
    plt.ylabel("training loss")
    plt.legend()
    plt.savefig("training_loss.png")
    plt.show()

    test_data_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=4, shuffle=True, num_workers=4)
    model1 = torch.load('net1.pth').eval()
    cm1 = run_testing(model1, test_data_loader)
    plt.figure()
    sns.heatmap(cm1,annot = True, fmt = "", cmap = "Blues", cbar = False, xticklabels = ['airplane','bus','cat','dog','pizza'], yticklabels = ['airplane','bus','cat','dog','pizza'])
    plt.title("Net 1")
    plt.xlabel("Predicted")
    plt.ylabel("Ground Truth")
    plt.savefig("cm1.jpg")
```