

# hw9\_ChengjunGuo

Chengjun Guo

April 2023

## 1 Multihead attention block

```
1 class MultiHeadAttention(nn.Module):
2     def __init__(self, emb_size = 768, num_heads = 8, dropout = 0):
3         super().__init__()
4         self.emb_size = emb_size
5         self.num_heads = num_heads
6         # separate into q k v
7         self.qkv = nn.Linear(emb_size, emb_size * 3)
8
9
10    def forward(self, x):
11        # batch N_W M*3 -> 3 batch N_H N_W s_qkv
12        queries, keys, values = rearrange(self.qkv(x), "b n (h d
13        qkv) -> (qkv) b h n d", h=self.num_heads, qkv=3)
14        # calculate filter Matrix multiplication Q @ K ,batch N_H
15        N_W s_qkv -> batch N_H N_W N_W
16        filter = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
17        #softmax 1 to -1
18        x = F.softmax(filter, dim=-1)
19        # Matrix multiplication X @ V batch N_H N_W N_W @ batch N_H
20        N_W s_qkv -> batch N_H N_W s_qkv
21        out = torch.einsum('bhal, bhlv -> bhav ', x, values)
22        #batch N_H N_W s_qkv -> batch N_H M, normalize
23        out = rearrange(out, "b h n d -> b n (h d)") / (self.
24        emb_size ** (1 / 2))
25        return out
```

Listing 1: Code block

## 2 ViT implementation

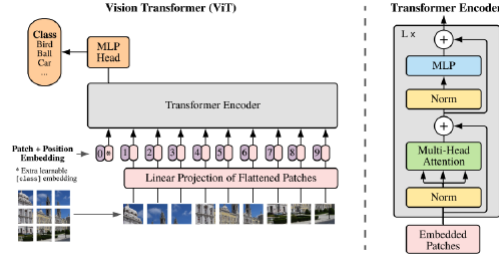


Figure 1: ViT structure

```

1 class PatchEmbedding(nn.Module):
2     def __init__(self, in_channels = 3, patch_size = 16, emb_size =
768, img_size = 224):
3         super().__init__()
4         self.patch_size = patch_size
5         self.proj = nn.Sequential(
6             # takes in batch * 3 channel * 64 * 64 images, generate
batch * (16*16*3) * 4*4
7             nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
, stride=patch_size),
8             # takes in batch * (16*16*3) * 4*4, convert to batch *
(4*4) * (16*16*3)
9             Rearrange('b c (h) (w) -> b (h w) c'),
10        )
11        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size))
12        self.positions = nn.Parameter(torch.randn((img_size //
patch_size) ** 2 + 1, emb_size))
13
14    def forward(self, x):
15        batch_size = x.shape[0]
16        x = self.proj(x)
17        # convert cls token to batch, 1, emb_size
18        cls_tokens = self.cls_token.repeat(batch_size, 1, 1)
19        x = torch.cat([cls_tokens, x], dim=1)
20        #position embedding
21        x += self.positions
22        return x
23
24
25 class ClassificationHead(nn.Sequential):
26     def __init__(self, emb_size = 768, n_classes = 5):
27         super().__init__(
28             Reduce('b n e -> b e', reduction='mean'),
29             nn.LayerNorm(emb_size),
30             nn.Linear(emb_size, n_classes))
31
32
33 class ViT(nn.Sequential):
34     def __init__(self, in_channels = 3, patch_size = 16, emb_size =
768, img_size = 64, depth = 2, n_classes = 5):

```

35

```
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_attn_heads = depth), ClassificationHead(emb_size,
    n_classes))
```

Listing 2: Code block

The encoding block is as provided with 6ViT helper. 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  $64 \times 64$  image is divided into 16  $16 \times 16$  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

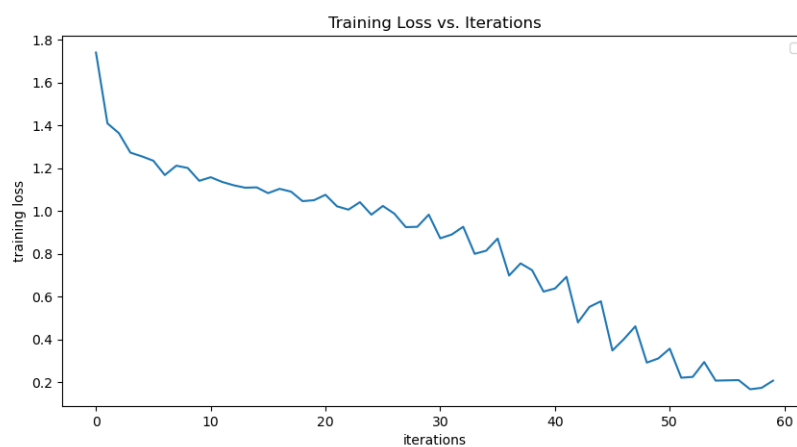


Figure 2: ViT training loss 20 epochs

### 4 Confusion matrix of ViT

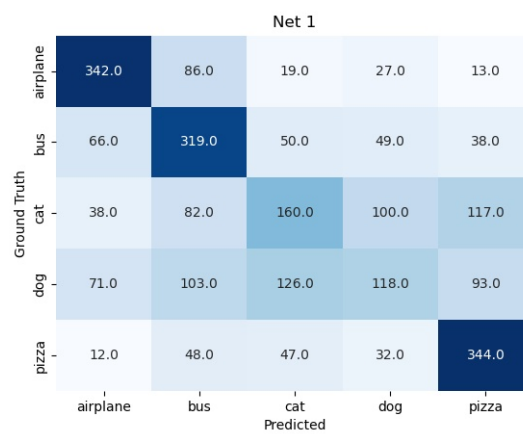


Figure 3: Confusion Matrix of ViT

### 5 Comparison with hw4 CNN performance

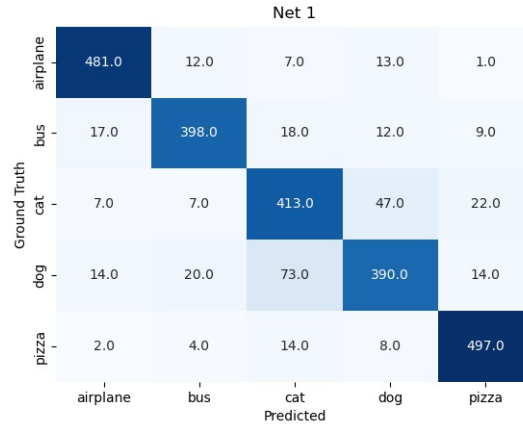


Figure 4: Confusion Matrix of CNN

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:

---

```

1 ## This code is from the Transformers co-class of DLStudio:
2
3 ## https://engineering.purdue.edu/kak/distDLS/
4
5 import torch
6 import torch.nn as nn
7 import torch.nn.functional as F
8 from einops import rearrange, reduce, repeat
9
10 device = torch.device("cuda:0" if torch.cuda.is_available() else "
    cpu")
11 # device = torch.device("cpu")
12 class MasterEncoder(nn.Module):
13     def __init__(self, max_seq_length, embedding_size,
14         how_many_basic_encoders, num_attn_heads):
15         super().__init__()
16         self.max_seq_length = max_seq_length
17         self.basic_encoder_arr = nn.ModuleList([BasicEncoder(
18             max_seq_length, embedding_size, num_attn_heads) for _
19             in range(how_many_basic_encoders)]) # (A)
20
21     def forward(self, sentence_tensor):

```

```

20     out_tensor = sentence_tensor
21     for i in range(len(self.basic_encoder_arr)): # (B)
22         out_tensor = self.basic_encoder_arr[i](out_tensor)
23     return out_tensor
24
25
26 class BasicEncoder(nn.Module):
27     def __init__(self, max_seq_length, embedding_size,
28                 num_atten_heads):
29         super().__init__()
30         self.max_seq_length = max_seq_length
31         self.embedding_size = embedding_size
32         self.qkv_size = self.embedding_size // num_atten_heads
33         self.num_atten_heads = num_atten_heads
34         # self.self_attention_layer = SelfAttention(max_seq_length,
35         embedding_size, num_atten_heads) # (A)
36         self.self_attention_layer = MultiHeadAttention(emb_size =
37         embedding_size, num_heads = num_atten_heads)
38         self.norm1 = nn.LayerNorm(self.embedding_size) # (C)
39         self.W1 = nn.Linear(self.max_seq_length * self.
40         embedding_size,
41                             self.max_seq_length * 2 * self.
42         embedding_size)
43         self.W2 = nn.Linear(self.max_seq_length * 2 * self.
44         embedding_size,
45                             self.max_seq_length * self.
46         embedding_size)
47         self.norm2 = nn.LayerNorm(self.embedding_size) # (E)
48
49     def forward(self, sentence_tensor):
50         input_for_self_attn = sentence_tensor.float()
51         normed_input_self_attn = self.norm1(input_for_self_attn)
52         output_self_attn = self.self_attention_layer(
53         normed_input_self_attn).to(device) # (F)
54         input_for_FFN = output_self_attn + input_for_self_attn
55         normed_input_FFN = self.norm2(input_for_FFN) # (I)
56         basic_encoder_out = nn.ReLU()(
57         self.W1(normed_input_FFN.view(sentence_tensor.shape[0],
58         -1))) # (K)
59         basic_encoder_out = self.W2(basic_encoder_out) # (L)
60         basic_encoder_out = basic_encoder_out.view(sentence_tensor.
61         shape[0], self.max_seq_length, self.embedding_size)
62         basic_encoder_out = basic_encoder_out + input_for_FFN
63         return basic_encoder_out
64
65 ##### Self Attention Code
66 TransformerPreLN #####
67
68 class SelfAttention(nn.Module):
69     def __init__(self, max_seq_length, embedding_size,
70                 num_atten_heads):
71         super().__init__()
72         self.max_seq_length = max_seq_length
73         self.embedding_size = embedding_size
74         self.num_atten_heads = num_atten_heads
75         self.qkv_size = self.embedding_size // num_atten_heads

```

```

64     self.attention_heads_arr = nn.ModuleList([AttentionHead(
        self.max_seq_length,
65
        self.qkv_size) for _ in range(num_attn_heads)]) # (A)
66
67     def forward(self, sentence_tensor): # (B)
68         concat_out_from_attn_heads = torch.zeros(sentence_tensor.
        shape[0], self.max_seq_length,
69
        self.
        num_attn_heads * self.qkv_size).float()
70         for i in range(self.num_attn_heads): # (C)
71             sentence_tensor_portion = sentence_tensor[:,
72
        :, i * self.
        qkv_size: (i+1) * self.qkv_size]
73             concat_out_from_attn_heads[:, :, i * self.qkv_size: (i
        +1) * self.qkv_size] = \
74                 self.attention_heads_arr[i](sentence_tensor_portion
        ) # (D)
75         return concat_out_from_attn_heads
76
77
78 class AttentionHead(nn.Module):
79     def __init__(self, max_seq_length, qkv_size):
80         super().__init__()
81         self.qkv_size = qkv_size
82         self.max_seq_length = max_seq_length
83         self.WQ = nn.Linear(max_seq_length * self.qkv_size,
84
        max_seq_length * self.qkv_size) # (B)
85         self.WK = nn.Linear(max_seq_length * self.qkv_size,
86
        max_seq_length * self.qkv_size) # (C)
87         self.WV = nn.Linear(max_seq_length * self.qkv_size,
88
        max_seq_length * self.qkv_size) # (D)
89         self.softmax = nn.Softmax(dim=1) # (E)
90
91     def forward(self, sentence_portion): # (F)
92         Q = self.WQ(sentence_portion.reshape(
93
        sentence_portion.shape[0], -1).float()).to(device) # (
        G)
94         K = self.WK(sentence_portion.reshape(
95
        sentence_portion.shape[0], -1).float()).to(device) # (
        H)
96         V = self.WV(sentence_portion.reshape(
97
        sentence_portion.shape[0], -1).float()).to(device) # (
        I)
98         Q = Q.view(sentence_portion.shape[0],
99
        self.max_seq_length, self.qkv_size) # (J)
100        K = K.view(sentence_portion.shape[0],
101
        self.max_seq_length, self.qkv_size) # (K)
102        V = V.view(sentence_portion.shape[0],
103
        self.max_seq_length, self.qkv_size) # (L)
104        A = K.transpose(2, 1) # (M)
105        QK_dot_prod = Q @ A # (N)
106        rowwise_softmax_normalizations = self.softmax(QK_dot_prod)
        # (O)
107        Z = rowwise_softmax_normalizations @ V
108        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float
        ().to(device)) # (S)

```

```

109         Z = coeff * Z # (T)
110         return Z
111
112
113 class MultiHeadAttention(nn.Module):
114     def __init__(self, emb_size = 768, num_heads = 8):
115         super().__init__()
116         self.emb_size = emb_size
117         self.num_heads = num_heads
118         # separate into q k v
119         self.qkv = nn.Linear(emb_size, emb_size * 3)
120
121
122     def forward(self, x):
123         # batch N_W M*3 -> 3 batch N_H N_W s_qkv
124         queries, keys, values = rearrange(self.qkv(x), "b n (h d
125         qkv) -> (qkv) b h n d", h=self.num_heads, qkv=3)
126         # calculate filter Matrix multiplication Q @ K ,batch N_H
127         N_W s_qkv -> batch N_H N_W N_W
128         filter = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
129         #softmax 1 to -1
130         x = F.softmax(filter, dim=-1)
131         # Matrix multiplication X @ V batch N_H N_W N_W @ batch N_H
132         N_W s_qkv -> batch N_H N_W s_qkv
133         out = torch.einsum('bhal, bhlv -> bhav ', x, values)
134         #batch N_H N_W s_qkv -> batch N_H M, normalize
135         out = rearrange(out, "b h n d -> b n (h d)") / (self.
136         emb_size ** (1 / 2))
137         return out

```

---

hw9:

---

```

1  ##          https://engineering.purdue.edu/kak/distDLS/
2  import os
3  os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
4  import torch
5  from PIL import Image
6  from torch.utils.data import DataLoader, Dataset
7  import copy
8  import matplotlib.pyplot as plt
9  import seaborn as sns
10 from sklearn.metrics import confusion_matrix
11 import numpy as np
12 import sys
13 import random
14 import pickle
15 import time
16 import torch.nn as nn
17 import torchvision
18 import torchvision.transforms as tvf
19 import torch.nn.functional as F
20 from ViTHelper import *

```



```

21 from einops.layers.torch import Rearrange, Reduce
22 from torchsummary import summary
23 from pycocotools.coco import COCO
24
25
26 class PatchEmbedding(nn.Module):
27     def __init__(self, in_channels = 3, patch_size = 16, emb_size =
28         768, img_size = 224):
29         super().__init__()
30         self.patch_size = patch_size
31         self.proj = nn.Sequential(
32             # takes in batch * 3 channel * 64 * 64 images, generate
33             batch * (16*16*3) * 4*4
34             nn.Conv2d(in_channels, emb_size, kernel_size=patch_size
35             , stride=patch_size),
36             # takes in batch * (16*16*3) * 4*4, convert to batch *
37             (4*4) * (16*16*3)
38             Rearrange('b c (h) (w) -> b (h w) c'),
39         )
40         self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size))
41         self.positions = nn.Parameter(torch.randn((img_size //
42             patch_size) ** 2 + 1, emb_size))
43
44     def forward(self, x):
45         batch_size = x.shape[0]
46         x = self.proj(x)
47         # convert cls token to batch, 1, emb_size
48         cls_tokens = self.cls_token.repeat(batch_size, 1, 1)
49         x = torch.cat([cls_tokens, x], dim=1)
50         #position embedding
51         x += self.positions
52         return x
53
54 class ClassificationHead(nn.Sequential):
55     def __init__(self, emb_size = 768, n_classes = 5):
56         super().__init__(
57             Reduce('b n e -> b e', reduction='mean'),
58             nn.LayerNorm(emb_size),
59             nn.Linear(emb_size, n_classes))
60
61 class ViT(nn.Sequential):
62     def __init__(self, in_channels = 3, patch_size = 16, emb_size =
63         768, img_size = 64, depth = 2, n_classes = 5):
64         super(ViT, self).__init__(PatchEmbedding(in_channels,
65             patch_size, emb_size, img_size), MasterEncoder(max_seq_length =
66             17, embedding_size = emb_size, how_many_basic_encoders = depth,
67             num_attn_heads = depth), ClassificationHead(emb_size,
68             n_classes))
69
70 def run_training(net, train_data_loader, net_save_path):
71     net = net.to(device)

```

```

68 criterion = torch.nn.CrossEntropyLoss()
69 optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas
=(0.9, 0.99), eps = 1e-4)
70 epochs = 20
71 Loss_runtime = []
72 for epoch in range(epochs):
73     running_loss = 0.0
74     for i, data in enumerate(train_data_loader):
75         inputs, labels = data
76         inputs = inputs.to(device)
77         labels = labels.to(device)
78         optimizer.zero_grad()
79         outputs = net(inputs)
80         loss = criterion(outputs, labels)
81         loss.backward()
82         optimizer.step()
83         running_loss += loss.item()
84         if (i+1) % 500 == 0:
85             print("[epoch: %d, batch: %5d] loss: %.3f" % (epoch
+1, i+1, running_loss/500))
86             Loss_runtime.append(running_loss/500)
87             running_loss = 0.0
88 torch.save(net, net_save_path)
89 return Loss_runtime
90
91
92 def run_testing(net, validation_data_loader):
93     net = copy.deepcopy(net)
94     net = net.to(device)
95     Confusion_Matrix = torch.zeros(5, 5)
96     for i, data in enumerate(validation_data_loader):
97         inputs, labels = data
98         inputs = inputs.to(device)
99         labels = labels.to(device)
100         outputs = net(inputs)
101         _, predicted = torch.max(outputs.data, 1)
102         predicted = predicted.tolist()
103         for label, prediction in zip(labels, predicted):
104             Confusion_Matrix[label][prediction] += 1
105     return Confusion_Matrix
106
107
108 class MyDataset(torch.utils.data.Dataset):
109     def __init__(self):
110         super().__init__()
111         self.root_path = 'E:\ECE60146DL\hw4_new\data/'
112         self.coco_json_path = 'E:\ECE60146DL\hw4_new/
annotations_trainval2014/annotations/instances_train2014.json'
113         self.class_list = ['airplane', 'bus', 'cat', 'dog', 'pizza']
114         self.images_per_class = 2000 # 1500 for train, 500 for
validation
115         self.coco = COCO(self.coco_json_path)
116         self.img_labels = []
117         self.transform = tvn.Compose([tvn.ToTensor(), tvn.Normalize
([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])
118         for cat in self.class_list:
119             path, dirs, files = next(os.walk(self.root_path + cat))

```

```

120         for file in files:
121             # first image path, second label
122             self.img_labels.append([cat + '/' + file, self.
class_list.index(cat)])
123
124
125     def __len__(self):
126         return int(self.images_per_class * len(self.class_list))
127
128     def __getitem__(self, index):
129         img_path = os.path.join(self.root_path, self.img_labels[
index][0])
130         image = Image.open(img_path).convert("RGB")
131         label = torch.tensor(self.img_labels[index][1])
132
133         image = self.transform(image)
134         return image, label
135
136
137 if __name__ == '__main__':
138     if torch.cuda.is_available() == True:
139         device = torch.device("cuda:0")
140     else:
141         device = torch.device("cpu")
142     # device = torch.device("cpu")
143     vit = ViT()
144     # summary(vit.to("cuda"), (3,64,64), device="cuda")
145
146     my_dataset = MyDataset()
147     train_dataset, test_dataset = torch.utils.data.random_split(
my_dataset, [7500, 2500])
148     train_data_loader = torch.utils.data.DataLoader(dataset=
train_dataset, batch_size=4, shuffle=True, num_workers=4)
149     loss1 = run_training(vit, train_data_loader, "net1.pth")
150     plt.figure(figsize=(10, 5))
151     plt.title("Training Loss vs. Iterations")
152     plt.plot(loss1)
153     plt.xlabel("iterations")
154     plt.ylabel("training loss")
155     plt.legend()
156     plt.savefig("training_loss.png")
157     plt.show()
158
159     test_data_loader = torch.utils.data.DataLoader(dataset=
test_dataset, batch_size=4, shuffle=True, num_workers=4)
160     model1 = torch.load('net1.pth').eval()
161     cm1 = run_testing(model1, test_data_loader)
162     plt.figure()
163     sns.heatmap(cm1, annot = True, fmt = "", cmap = "Blues", cbar =
False, xticklabels = ['airplane', 'bus', 'cat', 'dog', 'pizza'],
yticklabels = ['airplane', 'bus', 'cat', 'dog', 'pizza'])
164     plt.title("Net 1")
165     plt.xlabel("Predicted")
166     plt.ylabel("Ground Truth")
167     plt.savefig("cm1.jpg")

```

---