Lec9_CAM,Attention,Transformers

dataset:

`trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2)`

batch_size → samples per batch
shuffle → shuffle the order of samples
num_workers: loading parallelization

`for i, (images, labels) in enumerate(trainloader):`

i→ batch ID
images → images of the batch
labels: labels of the images

Design the Networks

img size - 28, batch size - 32

$28 * 28 * 1 \Rightarrow$(con1) $26 * 26 * 32$(convert to 3d) $\Rightarrow$(d1) $128 \Rightarrow$(d2) 10

```
def forward(self,x):
    # 32x1x28x28->32x32x26x26
    x = self.conv1(x)
    x = F.relu(x)
    # flatten to 32x(32*26*26)
    x=x.flatten(start_dim-1)
    # 32x(32*26*26) => 32x128
    x=self.d1(x)
    x=F.relu(x)
    # logits => 32x10
    logits = self.d2(x)
    out = F.softmax(logits,dim=1)
    return out
```

$$H(P^*|P) = -\sum_i \underbrace{P^*(i)}_{\text{TRUE CLASS DISTRIBUTION}} \log \underbrace{P(i)}_{\text{PREDICTED CLASS DISTRIBUTION}}$$

Advantage of DL: build complex model by simply stacking layers

Datasets for deep learning

ImageNet(classify): largest & most classic CV public dataset, quality low, image wrong

CIFAR(classify): more tidy, suitable for starting with img classification

COCO(detect, segment): largest recognition dataset with fine annotation

ADE20K(segment): rich indoor/outdoor scene

Class Activation Mapping (CAM)

Grad-CAM: gradient-weighted CAM

- use gradient of target concept flow into final convolutional layer to produce coarse粗略的 localization map
- **highlighting important regions** in the image for predicting the concept.
- applicable to wide variety of CNN model-families without architectural changes or re-training

Score-CAM:

- get rid of dependence of gradients by obtaining weight of each activation map through its forward passing score on target class
- result is obtained by linear combination of weights & activation maps.
- achieves better visual performance & fairness of interpreting decision making process

Conceptor-CAM

- model both inter-and inner-relation in one shot.
- compatible to other CAMs, use conceptors to "regulate" relations in those CAMs for better performance

Attentions

*__Attention:__ pays greater attention to certain factors when processing data.

CBAM(Convolutional Block Attention Module)

learns what and where to emphasize or suppress and refines intermediate features effectively.

Self Attentions, and Transformers

Self Attention

- calculates a weight according to input data to represent attention of each position in data sequence to other different

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

Transformer: encoder-decoder architecture based on self-attemtion, field of NLP→CV

ViT(Vision Transformer):

- image segmented into patches before entering model & patches are reshaped as 2d vector
- a standard transformer Encoder architecture, and output of Encoder gets probability of each classification after MLP layer.

```
# ResNet50 Model
norm_mean = [0.485, 0.456, 0.406]
norm_std = [0.229, 0.224, 0.225]

inference_transform = transforms.Compose([
```

```
    transforms.Resize(256),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])

def img_transform(img_rgb, transform=None):
    # img_rgb: PIL Image; transform: torchvision.transform; return: tensor
    if transform is None:
        raise ValueError("there is no transform")
    img_t = transform(Image.fromarray(img_rgb))
    return img_t

train_imgs = [img_transform(sample.img, inference_transform) for sample in
samples['train']]
train_imgs = torch.stack(train_imgs, dim=0)
test_imgs = [img_transform(sample.img, inference_transform) for sample in
samples['val']]
test_imgs = torch.stack(test_imgs, dim=0)

train_labels = [sample.label for sample in samples['train']]
train_labels = torch.tensor(train_labels)

# define a classifier following the network
class classification_head(nn.Module):
    def __init__(self,in_ch,num_classes):
        super(classification_head,self).__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(in_ch,num_classes)

    def forward(self, x):
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x

#define ResNet model
class Net(nn.Module):
    def __init__(self, input_ch, num_class,pretrained=True):
        super(Net,self).__init__()
        model = models.resnet50(pretrained=pretrained)
        self.backbone =  nn.Sequential(*list(model.children())[:-2]) #remove the
last Avgpool and Fully Connected Layer
        self.classification_head = classification_head(2048, num_class)

    def forward(self,x):
        x = self.backbone(x)
        output = self.classification_head(x)
        return output

# creat a model
model = Net(3, 2)
# fix the weights of ResNet
for p in model.backbone.parameters(): p.requires_grad = False
```

```
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad,
model.parameters()), lr=0.001)
model.train()

# training

# GramCAM model
target_layers = [model.backbone[-1][-1]]
inputs = []
imgs = []
for sample in samples['train']:
    rgb_img = sample.img
    rgb_img = cv2.resize(rgb_img, (256, 341))
    rgb_img = np.float32(rgb_img) / 255
    imgs.append(rgb_img)
    inputs.append(preprocess_image(rgb_img, mean=norm_mean, std=norm_std))

input_tensor = torch.cat(inputs, dim=0)

# Construct the CAM object once, and then re-use it on many images:
cam = GradCAM(model=model, target_layers=target_layers, use_cuda=False)

# You can also pass aug_smooth=True and eigen_smooth=True, to apply smoothing.
grayscale_cams = cam(input_tensor=input_tensor, targets=None)
```

ViT

```
class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head *  heads
        project_out = not (heads == 1 and dim_head == dim)

        self.heads = heads
        self.scale = dim_head ** -0.5

        self.attend = nn.Softmax(dim = -1)
        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)

        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
        ) if project_out else nn.Identity()

    def forward(self, x):
        qkv = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h =
self.heads), qkv)

        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale
```

```
        attn = self.attend(dots)

        out = torch.matmul(attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)

from einops import rearrange, repeat
from einops.layers.torch import Rearrange

# helpers
def pair(t):
    return t if isinstance(t, tuple) else (t, t)

# classes
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)

class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.net(x)

class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout = 0.):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Attention(dim, heads = heads, dim_head = dim_head,
dropout = dropout)),
                PreNorm(dim, FeedForward(dim, mlp_dim, dropout = dropout))
            ]))
    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
        return x

class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads,
```

```
mlp_dim, pool = 'cls', channels = 3, dim_head = 64, dropout = 0., emb_dropout =
0.):
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)

        assert image_height % patch_height == 0 and image_width % patch_width ==
0, 'Image dimensions must be divisible by the patch size.'

        num_patches = (image_height // patch_height) * (image_width //
patch_width)
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls token)
or mean (mean pooling)'

        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height,
p2 = patch_width),
            nn.Linear(patch_dim, dim),
        )

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

        self.transformer = Transformer(dim, depth, heads, dim_head, mlp_dim,
dropout)

        self.pool = pool
        self.to_latent = nn.Identity()

        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)

        x = self.transformer(x)

        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

        x = self.to_latent(x)
        return self.mlp_head(x)

# training
v = ViT(
```

```
    image_size = 256,
    patch_size = 32,
    num_classes = 2,
    dim = 1024,
    depth = 6,
    heads = 16,
    mlp_dim = 2048,
    dropout = 0.1,
    emb_dropout = 0.1
)

inputs = []
for sample in samples['train']:
    rgb_img = sample.img
    rgb_img = cv2.resize(rgb_img, (256, 256))
    rgb_img = np.float32(rgb_img) / 255
    imgs.append(rgb_img)
    inputs.append(preprocess_image(rgb_img, mean=norm_mean,
                                   std=norm_std))

train_imgs = torch.cat(inputs, dim=0)

v.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, v.parameters()),
lr=0.001)
v.train()

# training
num_epochs = 100
for epoch in range(num_epochs):
    if torch.cuda.is_available():
        inputs = Variable(train_imgs).cuda()
        target = Variable(train_labels).cuda()
    else:
        inputs = Variable(train_imgs)
        target = Variable(train_labels)

    # forward
    out = v(inputs)
    loss = criterion(out, target)

    # backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 20 == 0:
        print('Epoch[{}/{}], loss: {:.6f}'
              .format(epoch+1, num_epochs, loss.item()))
```