Deep Learning Lab 2
Binary Semantic Segmentation via PyTorch

110201033 陳威志[1]

[1]國立中央大學 數學系 計算與資料科學組

Abstract

This Lab is to implement 2 different Binary Semantic Segmentation models, which are UNet and ResNet34+Unet, we train these models to depends on Oxford-IIIT Pet Dataset, and we expect that the output of the model will be a matrix which only includes 0s and 1s, means the background and foreground, respectively. We use Dice Score to evaluate the performance of the model, and the goal of this lab is to promote this Dice Score.

1. Inplementation Details

1.1 Details of training, evaluating and inferencing code

1.1.1 Training

The training process is to declare the model (UNet and ResNet34_Unet), the desired optimizer (we use Adam here) and the loss functions (Dice loss). During training, we take forward, calculate the lost, backward, and update in rotation. After training, evaluate the trained parameter via the evaluation function. we tried several of seed and save the result into a .txt file, then choose the better result.

Below is the training source code, note that all source code in this report, the part of import packages and comments are omitted:

```
def train(): # one tab are omitted

seed = args.seed
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic
= True
torch.backends.cudnn.benchmark = True

net = args.model
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

if net == "unet":
    model = UNet().to(device)
elif net == "res34unet":
    model = ResNet34_UNet().to(device)

data_path = args.data_path
train_dataset = load_dataset
                (data_path, "train")
valid_dataset = load_dataset
                (data_path, "valid")

batch_size = args.batch_size
train_loader = DataLoader(train_dataset,
               batch_size=batch_size,
               shuffle=False)

optimizer = optim.Adam(model.parameters(),
            lr = args.learning_rate)
scheduler = StepLR(optimizer, step_size=5,
            gamma=0.1)
scaler = GradScaler()

for epoch in range(args.epochs):
    for batch_idx, batch in
    enumerate(train_loader):
        image = batch["image"]
        mask = batch["mask"]
        trimap = batch["trimap"]

        image = image.to(dtype=torch,
                float32,device=device)
        mask = mask.to(device)
        trimap = trimap.to(device)
        mask = mask.unsqueeze(1)

        optimizer.zero_grad()

        with autocast():
            pred = model(image)
            loss = LossFunction
                   (pred, mask)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
```

```
        dic = CalculateDiceScore
              (pred, mask)

        print(f"epoch {epoch+1},
         batch index:
         {batch_idx:4}({100*batch_idx/
         len(train_loader):.1f}%),
         {net} score: {dic.item():.4f}
         loss: {loss.item():.4f}")

    evaluate_score =
    evaluate(net, valid_dataset, device)

    log_file = f"./{net}_evaluate.txt"
    log_msg = f"seed: {args.seed},
              Avg dice score:
              {evaluate_score:4f}\n"
    with open(log_file, "a") as f:
        f.write(log_msg)

    print(f"{net} Average dice score:
    {evaluate_score:4f}")

torch.save(model.state_dict(),
f"../saved_models/{net}.pth")
```

1.1.2 Evaluating

The evaluating is simmilar to training process, but there are some point need to set, we have to met the model to eval mode to close the BatchNorm, and since the model has no need to update the parameters, so we use no_grad during the process to lighten the memory use. And since there were some issues occured when testing, we also use torch.checkpoint to further lighten the memory use.
Below is the evaluating source code:

```
def evaluate(net, data, device):

if net == "unet":
    model = UNet()
    model.load_state_dict(torch.load
          ("../saved_models/unet.pth",
          map_location=device))
    model.to(device)
elif net == "res34unet":
    model = ResNet34_UNet()
    model.load_state_dict(torch.load
          ("../saved_models/
          res34unet.pth",
          map_location=device))
    model.to(device)

model.eval()
total_dice = 0.0
total_samples = 0

with torch.no_grad():
    idx = 0
    for batch in data:
        images = batch["image"].
                 to(device)
        masks = batch["mask"].
                to(device)
        if images.dim() == 3:
            images = images.
                     unsqueeze(0)

        pred = checkpoint.checkpoint
               (model, images)

        dice = CalculateDiceScore
               (pred, masks)
        total_dice = total_dice
                     + dice
        total_samples += 1
        idx += 1
avg_dice = total_dice / total_samples
return avg_dice
```

1.1.3 Inferencing

In inferencing, we need to load the trained model first, then we do the process similar to evaluation, but this time we has no need to calculate the loss, but only the dice score.
Below is the evaluating source code:

```
def test(args):
data = load_dataset(args.data_path,
       "test")

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

if (args.model_type == "unet"):
    model = UNet().to(device)
    model.load_state_dict(torch.load
          (args.model, weights_only=True))
elif (args.model_type == "res34unet"):
    model = ResNet34_UNet().to(device)
    model.load_state_dict(torch.load
           (args.model, weights_only=True))

model.eval()
total_dice = 0.0
dic_list = []
test_loader = DataLoader(data,
              args.batch_size,
              shuffle= False)
```

```
for batch_idx, batch in enumerate
        (test_loader):

        image = batch["image"].to
                (device, dtype=torch
                .float32)
        mask = batch["mask"].to(device)

        pred = checkpoint.checkpoint
               (model, image)
        dice = CalculateDiceScore
               (pred, mask)
               /args.batch_size
        total_dice = total_dice + dice
        dic_list.append(dice)
        print(f"batch: {batch_idx}
              dic: {dice:.5f}")

avg_dice = np.mean([d.cpu().numpy()
           for d in dic_list])
print(f"dice score: {avg_dice:.4f}")

return avg_dice
```

1.2 Details of models

1.2.1 UNet

In the UNet architecture, each block are contained with two 2D convolution, so I defined a **DoubleConv** class to finish this task:

```
class DoubleConv(nn.Module):
def __init__(self, in_channel, out_channel):
    super(DoubleConv, self).__init__()
    self.double_conv = nn.Sequential(
        nn.Conv2d(in_channel, out_channel,
                  kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channel),
        nn.ReLU(),

        nn.Conv2d(out_channel, out_channel,
                  kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channel),
        nn.ReLU()
    )

def forward(self, x):
    return self.double_conv(x)
```

In the second half of the UNet architecture, that is, after the bottleneck, the part of upward, we define a **Upward** class to run this process:

```
class Upward(nn.Module):
def __init__(self, in_channel, out_channel):
    super(Upward, self).__init__()
    self.up_ward = nn.Sequential(
        nn.Upsample(scale_factor=2,
        mode='bilinear',
        align_corners=True),
        nn.Conv2d(in_channel, out_channel,
                  kernel_size=3, padding=1),
    )

def forward(self, x):
    return self.up_ward(x)
```

The architecture of UNet is simple, combined with 4 down blocks, a bottleneck and 4 up blocks, at the end we do convolution one more time and a sigmoid to generate a matrix which has the same size as the input matrix, and this output matrix only contains numbers between 0 and 1.
Below is the UNet architecture source code:

```
class UNet(nn.Module):

def __init__(self):
    super(UNet, self).__init__()
    self.down1 = DoubleConv(1,64)
    self.pool1 = nn.MaxPool2d
                 (kernel_size=2, stride=2)
    self.down2 = DoubleConv(64,128)
    self.pool2 = nn.MaxPool2d
                 (kernel_size=2, stride=2)
    self.down3 = DoubleConv(128,256)
    self.pool3 = nn.MaxPool2d
                 (kernel_size=2, stride=2)
    self.down4 = DoubleConv(256,512)
    self.pool4 = nn.MaxPool2d
                 (kernel_size=2, stride=2)
    self.middle = DoubleConv(512,1024)
    self.up1 = Upward(1024,512)
    self.up_conv1 = DoubleConv(1024,512)
    self.up2 = Upward(512,256)
    self.up_conv2 = DoubleConv(512,256)
    self.up3 = Upward(256,128)
    self.up_conv3 = DoubleConv(256,128)
    self.up4 = Upward(128,64)
    self.up_conv4 = DoubleConv(128,64)
    self.final_conv = nn.Conv2d(64, 1,
                      kernel_size=1)

def forward(self, x):
    encoder1 = self.down1(x)
    max_pool1 = self.pool1(encoder1)
    encoder2 = self.down2(max_pool1)
    max_pool2 = self.pool2(encoder2)
    encoder3 = self.down3(max_pool2)
    max_pool3 = self.pool3(encoder3)
    encoder4 = self.down4(max_pool3)
    max_pool4 = self.pool4(encoder4)
    middle_layer = self.middle(max_pool4)
    upward1 = self.up1(middle_layer)
```

```
    upward1 = F.interpolate(upward1,
              size=encoder4.shape[2:],
              mode="bilinear",
              align_corners=True)
    decoder1 = self.up_conv1(torch.cat
               ([upward1, encoder4],
               dim=1))
    upward2 = self.up2(decoder1)
    upward2 = F.interpolate(upward2,
              size=encoder3.shape[2:],
              mode="bilinear",
              align_corners=True)
    decoder2 = self.up_conv2(torch.cat
               ([upward2, encoder3],
               dim=1))
    upward3 = self.up3(decoder2)
    upward3 = F.interpolate(upward3,
              size=encoder2.shape[2:],
              mode="bilinear",
              align_corners=True)
    decoder3 = self.up_conv3(torch.
               cat([upward3,encoder2],
               dim=1))
    upward4 = self.up4(decoder3)
    upward4 = F.interpolate(upward4,
              size=encoder1.shape[2:],
              mode="bilinear",
              align_corners=True)
    decoder4 = self.up_conv4(torch.cat
               ([upward4, encoder1],
               dim=1))
    out = self.final_conv(decoder4)
    out = torch.sigmoid(out)
    out = torch.round(out)
    out = out
    return out
```

1.2.2 ResNet34+UNet

In ResNet34_UNet, ResNet34 is consisted of Convolution Block and the Residual Block, the convolution block is Convolution + BatchNorm + ReLU, the noteworthy point is that since there might exist a downsampling part in ResNet34, so we have to bother setting the stride of convolution.

In residual block, since the input of each block is from the previous one, we expressly notice that when downsampling, the dimension of skip connection will be different, so if there needs a downsampling, we need an additional convolution to do dimensionality reduction.

Convolution Block:

```
class Convolution(nn.Module):
def __init__(self, in_channel,
             out_channel):
    super(Convolution, self).__init__()
    self.convolution = nn.Sequential(
        nn.Conv2d(in_channel +
        out_channel, out_channel,
        kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channel),
        nn.ReLU(),
    )

def forward(self, x):
    return self.convolution(x)
```

Residual Block:

```
class ResBlock(nn.Module):
def __init__(self, in_channel,
    out_channel, stride=1,
    downsample=None):
    super(ResBlock, self).__init__()
    self.conv1 = nn.Conv2d(in_channel,
                 out_channel,
                 kernel_size=3,
                 stride=stride,
                 padding=1,
                 bias=False)
    self.batchnorm1 = nn.BatchNorm2d
                      (out_channel)
    self.relu1 = nn.ReLU(inplace=True)

    self.conv2 = nn.Conv2d(out_channel,
                 out_channel,
                 kernel_size=3,
                 stride=1, padding=1,
                 bias=False)
    self.batchnorm2 = nn.BatchNorm2d
                      (out_channel)
    self.stride = stride
    self.downsample = downsample

def forward(self, x):
    out = self.conv1(x)
    out = self.batchnorm1(out)
    out = self.relu1(out)

    out = self.conv2(out)
    out = self.batchnorm2(out)

    if self.downsample is not None:
        x = self.downsample(x)

    out = out + x
    out = self.relu1(out)
    return out
```

ResNet34

```
class ResNet34(nn.Module):
def __init__(self):
    super(ResNet34, self).__init__()
    self.conv1 = nn.Conv2d(1, 64,
                  kernel_size=7,
                  stride=2, padding=3,
                  bias=False)
    self.batchnorm1 = nn.BatchNorm2d
                       (64)
    self.relu1 = nn.ReLU(inplace=True)
    self.pool = nn.MaxPool2d
                 (kernel_size=3,
                 stride=2, padding=1)
    self.layer1 = self.Layer(64,
                   64, 3)
    self.layer2 = self.Layer(64,
                   128, 4, stride=2)
    self.layer3 = self.Layer(128,
                   256, 6, stride=2)
    self.layer4 = self.Layer(256,
                   512, 3, stride=2)

def Layer(self, in_channel,
        out_channel, block,
        stride=1):
    downsample = None
    if (stride != 1) or
       (in_channel != out_channel):
        downsample = nn.Sequential(
            nn.Conv2d(in_channel,
                      out_channel,
                      kernel_size=1,
                      stride=stride,
                      bias=False),
            nn.BatchNorm2d(out_channel),
        )
    layers = []
    layers.append(ResBlock(in_channel,
                   out_channel, stride,
                   downsample))
    for i in range(1, block):
        layers.append(ResBlock
               (out_channel,
                out_channel))
    return nn.Sequential(*layers)

def forward(self, x):
    x = self.conv1(x)
    x = self.batchnorm1(x)
    x = self.relu1(x)
    x = self.pool(x)
    layer1 = self.layer1(x)
    layer2 = self.layer2(layer1)
    layer3 = self.layer3(layer2)
    layer4 = self.layer4(layer3)

    return layer1, layer2,
           layer3, layer4
```

ResNet34 + UNet

```
class ResNet34_UNet(nn.Module):
def __init__(self):
    super(ResNet34_UNet, self).
          __init__()
    self.res34 = ResNet34()
    self.up1 = Convolution(512, 256)
    self.up2 = Convolution(256, 128)
    self.up3 = Convolution(128, 64)
    self.up4 = nn.Sequential(
        nn.Conv2d(64, 32, kernel_size=3,
                  padding=1),
        nn.BatchNorm2d(32),
        nn.ReLU(),
    )
    self.final_conv = nn.Conv2d(32, 1,
                       kernel_size=1)

def forward(self, x):
    layer1, layer2, layer3, layer4 =
            self.res34(x)
    x = F.interpolate(layer4,
                size=layer3.shape[2:],
                mode='bilinear',
                align_corners=False)
    x = self.up1(torch.cat(
                 [x, layer3], dim=1))
    x = F.interpolate(x,
                size=layer2.shape[2:],
                mode='bilinear',
                align_corners=False)
    x = self.up2(torch.cat(
                 [x, layer2], dim=1))
    x = F.interpolate(x,
                size=layer1.shape[2:],
                mode='bilinear',
                align_corners=False)
    x = self.up3(torch.cat(
                 [x, layer1], dim=1))
    x = F.interpolate(x,
                scale_factor=2,
                mode='bilinear',
                align_corners=False)
    x = self.up4(x)
    x = self.final_conv(x)
    x = F.interpolate(x,
                size=(256, 256),
                mode='bilinear',
                align_corners=False)
    x = torch.sigmoid(x)
```

```
    x = 1-torch.round(x)
    return x
```

2. Data Preprocessing

Compare to the standard approaches, we transform the image to grayscale since we thought that the grayscale image might be easier to read than the original images by some degrees. Those teaching guide or demonstration on the Internet often use the data only went through the resizing and normalization, since we found that the teaching assistant(s) had help us resize the image size into 256×256, thus the only thing we did is to transform the image to grayscale.

We prepared another transform function also, that function extract the edges and turn the image to grayscale, then merge these 2 layers into 2 channel. But the effect was not well, so at the end we use the grayscale only.
Below is the grayscale transform function:

```
def grayscale_transform(image,
        mask, trimap):

    image = Image.fromarray(image)
    mask = Image.fromarray(mask)
    trimap = Image.fromarray(trimap)

    image_size = (256, 256)
    image = image.resize(image_size,
            Image.BILINEAR)
    mask = mask.resize(image_size,
           Image.NEAREST)
    trimap = trimap.resize(image_size,
           Image.NEAREST)

    image = np.round(np.array
            (image.convert("L")) /
              255.0)
    mask = np.array(mask)
    trimap = np.array(trimap)

    gray = np.expand_dims(image,
                              axis=-1)

    gray = torch.from_numpy(gray).
           float().permute(2, 0, 1)
    mask = torch.from_numpy(mask)
           .float()
    trimap = torch.from_numpy(trimap)
             .float()
    return dict(image=gray,
           mask=mask, trimap=trimap)
```

3. Analyze the Experiment Results

Below are the Hyperparameter settings:

- Training Batch Size: 8
- Evaluate Function: Dice Score (Given by Teaching Assistants)
- Loss Function: Dice Lose (1 - Dice Score)
- Optimizer: Adam
- Epoch: 1
- Leraning Rate: 1e-4

The test dice score: about 0.45 $\sim$ 0.60
PS: This score may different caused by using different seed. In short, the experiment failed.
In some reason that we cannot find out why it occurs, the dice score did not change at all while the epoch iterating, we tried to change the transform function to use different method preprocessing the data, there is a function in the source code, oxford_pet.py, called normalize_transform, we originally wanted use others method(for example, we heard that someone just resize the image into 256×256 and divide each entries with 255.0, they had about 0.95 dice score), but since we use the grayscale images, the in channel we use was 1, but the in channel of the others were 3, and if we adjust the parameter in the model, it occurs more warnings and errors. furthermore, we found that there are several difference between ours and others model.

4. Execution steps

Our code is simple, just follow these 2 steps:

1. Run train.py
2. Run inference.py

5. Discussion

5.1 Methods may bring better results

Basic input image assumption:

- size: 256 $\times$ 256
- color: RGB

1. k-means clustering
   this method may work well if the following condition satisfies:
   (a) The number of fur color class of the pet does not too many (counterexample: calico cat)

   (b) The color difference between foreground and background is not too small (counterexample: an orange cat with mud background)

2. Ensemble learning
   One of classicial ensemble learning: Adaboost, which merges several weak classifiers into a single strong classifier, for example we could merge UNet and ResNet34_UNet model into an Adaboost classifier, it may further increase the accuracy.

5.2 Potential research topics

We believe that the task of Image Semantic Segmentation is highly suitable as a downstream task in various image processing applications. For instance, it can be utilized for identifying Regions of Interest (ROI) or selectively processing specific objects within an image. Applications such as object-specific style transfer or image manipulation can greatly benefit from this approach. By first obtaining the object mask through semantic segmentation, operations can be confined to the masked region, thereby reducing the effective image size that requires processing. This, in turn, minimizes unnecessary computational costs and enhances efficiency.