

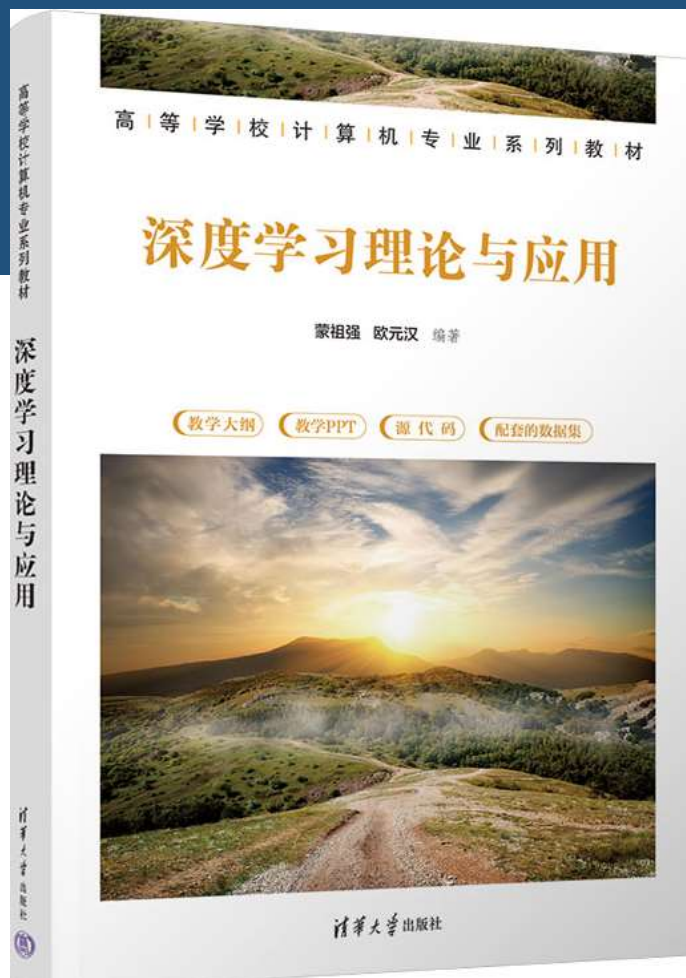
# 深度学习理论与应用

## Deep Learning Theory and Applications

蒙祖强，欧元汉 编著

# 教材

全国各大  
书店网店  
均有销售

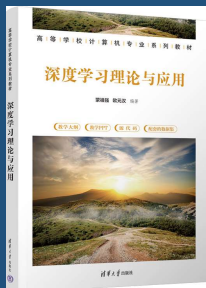


- **教学大纲**: 提供面向教育工程认证的教学大纲
- **教学PPT**: 提供课堂教学用的PPT课件
- **源代码**: 提供教材涉及的全部源代码
- **数据集**: 提供教材示例、案例用到的全部数据集

获取教学资源:

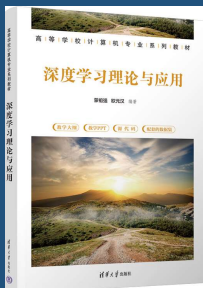
[http://www.tup.tsinghua.edu.cn/booksCenter/book\\_09988101.html](http://www.tup.tsinghua.edu.cn/booksCenter/book_09988101.html)

教材: 蒙祖强, 欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社, 2023年7月. (书号: 978-7-302-63508-6)



# 第6章 卷积神经网络的应用案例

蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社，2023年7月.



## 本章内容

contents

### 6.1 人脸识别

### 6.2 语义分割

### 6.3 目标检测

### 6.4 生成对抗网络

# 6.1 人脸识别



## 6.1.1 人脸识别的设计思路

人脸识别的**关键**：对比两张照片的相似度。

**相似度计算器**：

- 如果一张照片跟数据库中的某张照片“很相似”，那么就可以断定它们为同一个人的照片；如果跟所有照片“都不相似”，则该照片“不属于”数据库中的照片。因此，我们可以训练出一个“**相似度计算器**”，使之可以判别给定的两张照片是否是同一个人的照片：如果相似度很高（如大于给定的阈值），则表示是同一个人的照片，否则不是。
- “**相似度计算器**”的构造一般使用**孪生网络（Siamese Network）**来实现。孪生网络是一种“连体”的神经网络，这种“连体”实际上就是共享权值。可用图 6-1 来表示孪生网络的基本原理。

# 6.1 人脸识别



## 6.1.1 人脸识别的设计思路

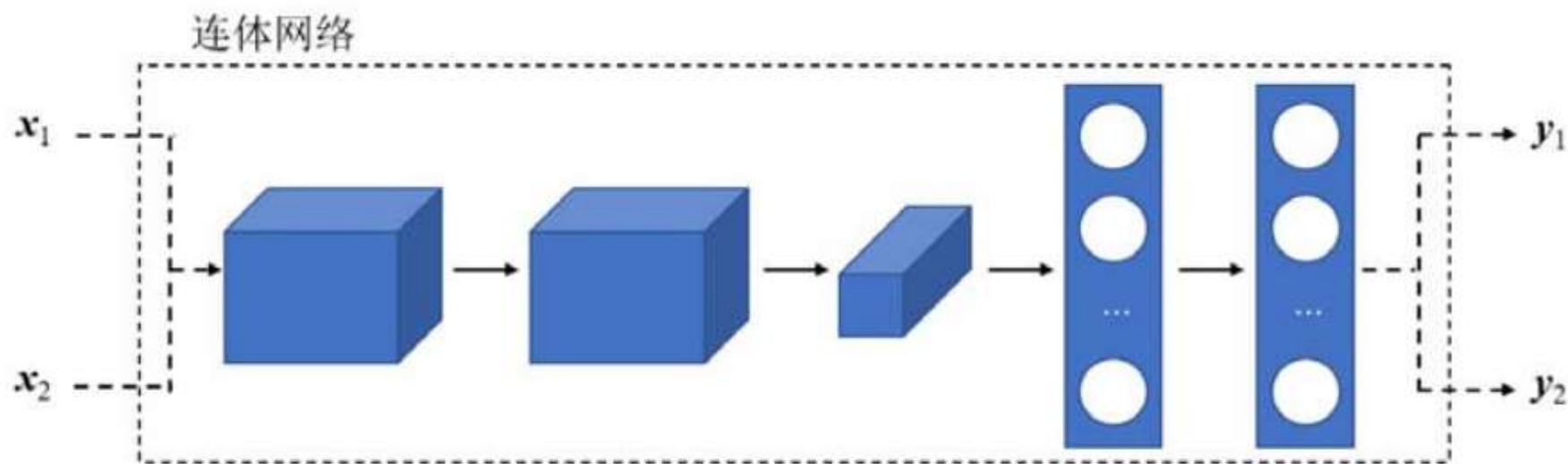


图 6-1 孪生网络的原理示意图

其中，图像  $x_1$  和  $x_2$  似乎分别输入到两个不同的网络  $f_1$  和  $f_2$  中，结果分别得到不同的输出  $y_1$  和  $y_2$ （即  $y_1 = f_1(x_1)$ ,  $y_2 = f_2(x_2)$ ），实际上这两个网络是共享了一个网络——连体网络。

# 6.1 人脸识别



## 6.1.1 人脸识别的设计思路

在图 6-1 所示的深度分类网络模型中，去掉最后的 softmax 层，然后通过共享剩下所有网络层的方法来构造两个深度网络。输入待识别的两张照片（指脸部照片，下同），分别将它们输入到这两个网络，然后通过参数训练，使得同属一个人的两张照片的相似度很高，否则相似度很低。一般情况下，用下列三元组表示一个样本：

$$(x_1, x_2, \text{label})$$

其中， $x_1$  和  $x_2$  分别表示输入的两张照片，label 为类别标识：如果 label = 0，则表示  $x_1$  和  $x_2$  为同一个人的脸部照片；如果 label = 1，则表示不是同一个人的。

两张照片  $x_1$  和  $x_2$  的相似度可用**欧氏距离**来度量，记为  $d(x_1, x_2) = \|y_1 - y_2\|$ ，即  $d(x_1, x_2)$  表示二者之间的欧氏距离。欧氏距离与相似度成反比，即欧氏距离  $d(x_1, x_2)$  越小，则  $x_1$  和  $x_2$  之间的相似度越高，否则越低。

# 6.1 人脸识别



## 6.1.1 人脸识别的设计思路

对于样本 $(x_1, x_2, 0)$ ，希望通过训练网络  $f_1$  和  $f_2$  的参数，使得  $d(x_1, x_2)$  越小越好；而对于样本 $(x_1, x_2, 1)$ ，则  $d(x_1, x_2)$  越大越好。为实现这一点，对每个输入样本 $(x_1, x_2, \text{label})$ ，可以通过构造并极小化下列损失函数来完成：

$$\mathcal{L} = (1 - \text{label}) \|\mathbf{y}_1 - \mathbf{y}_2\|^2 + \text{label} [\max(C - \|\mathbf{y}_1 - \mathbf{y}_2\|, 0)]^2$$

上式中， $C$  为某一常数。当  $\text{label} = 0$  时， $\mathcal{L} = \|\mathbf{y}_1 - \mathbf{y}_2\|^2$ ，因而对  $\mathcal{L}$  极小化实际上就是修正网络参数，使得  $x_1$  和  $x_2$  的欧氏距离  $\|\mathbf{y}_1 - \mathbf{y}_2\|^2$  不断缩小；当  $\text{label} = 1$  时， $\mathcal{L} = [\max(C - \|\mathbf{y}_1 - \mathbf{y}_2\|, 0)]^2$ ，极小化  $\mathcal{L}$  实际上意味着在  $[0, C^2]$  的范围内增大  $x_1$  和  $x_2$  的欧氏距离  $\|\mathbf{y}_1 - \mathbf{y}_2\|^2$ 。也就是说，如果  $x_1$  和  $x_2$  是同一个人的照片，则训练好后的网络能够增大二者之间的相似度，反之会减少二者的相似度。显然，这样的网络实际上就是上述的“相似度计算器”。



# 6.1 人脸识别



## 6.1.2 人脸识别程序

**【例 6.1】** 利用给定的数据集，设计并训练一个人脸识别网络，使之能够计算任意给定两张人脸照片的相似度，并测试网络模型的准确率。

**数据集：**




- 使用的是 **ORL 人脸数据集**，保存在 **./data/faces/** 目录下。该数据集一共包含 40 个不同人的 400 张人脸图片，每个人的照片保存在一个目录下，每个目录均包含 10 张 PGM 格式的图片，这些照片均为  $92 \times 112$  的灰度图像。
- **ORL 人脸数据集** 分为训练集和测试集，分别位于 **./data/training** 和 **./data/testing** 目录下。**Training** 目录包含 37 个子目录（37 人的照片），一共 370 张照片；**testing** 目录 3 个子目录（3 人的照片），一共 30 张照片。

# 6.1 人脸识别










## 6.1.2 人脸识别程序

程序的设计步骤：

(1) 采用三元组结构来表示一个样本。在程序代码中通常表示为, , )，它对应上述提及的( $\mathbf{x}_1$ ,  $\mathbf{x}_2$ , label)。

```
def __getitem__(self, idx):  
    img1, label1 = self.file_labels[idx]  
    fg = random.randint(0, 1) #随机生成 0 或 1  
    if fg == 1: #生成同一个人照片的三元组  
        k = idx + 1 #从下一条开始  
        while True:  
            k = 0 if k >= len(self.file_labels) else k  
            img2, label2 = self.file_labels[k]  
            k += 1  
            if int(label1) == int(label2):  
                break
```

构造样本集的方法：对给定 ，从  的下一张图片开始，循环搜索跟  表示同一个人或不同人的照片 ；如果为同一个人的照片，则相应的label 设置为 0，否则设置为 1；

至于选择同一个人的还是不同一个人的，则由**随机函数**决定，0 类样本和 1 类样本各自占大约 50%。当让  逐一遍历所有的图像，便可得到 370 条形状为(, , label)的三元组构成的训练集。

# 6.1 人脸识别



## 6.1.2 人脸识别程序

```
else: #生成不同一个人照片的三元组
    k = idx + 1 #从下一条开始
    while True:
        k = 0 if k >= len(self.file_labels) else k
        img2, label2 = self.file_labels[k]
        k += 1
        if int(label1) != int(label2):
            break
    img1 = getImg(img1)
    img2 = getImg(img2)
    label = torch.Tensor(np.array([int(label1 != label2)], dtype=np.float32))
    return img1, img2, label #三元组
```

该程序的核心代码（全部代码见教材P148-151）

# 6.1 人脸识别



## 6.1.2 人脸识别程序

(2) 使用 VGG19 来提取图像的特征，再加上三个全连接层，构成连体网络。然后定义 forward\_one() 函数来调用连体网络，而在 forward() 函数中编写代码来实现孪生网络功能。

① 导入 VGG19:

```
vgg19 = models.vgg19(pretrained=True) #导入 VGG19
vgg19_cnn = vgg19.features #只是利用 VGG19 的卷积网络部分
for param in vgg19_cnn.parameters():
    param.requires_grad = False #冻结参数
```

# 6.1 人脸识别



## 6.1.2 人脸识别程序

### ②定义孪生网络:

```
class SiameseNet(nn.Module):
    def __init__(self):
        super(SiameseNet, self).__init__()
        self.cnn = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(1, 3, 3, padding=1),
            vgg19_cnn, #用于特征提取
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(512) )
        self.fc1 = nn.Sequential(
            nn.Linear(512 * 3 * 3, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512))
```

```
def forward_one(self, x): #调用连体网络
    o = x
    o = self.cnn(o)
    o = o.reshape(x.size(0), -1)
    o = self.fc1(o)
    return o
def forward(self, img1, img2):
    o1 = self.forward_one(img1)
    o2 = self.forward_one(img2)
    return o1, o2
```

该程序的核心代码（全部代码见教材P148-151）

# 6.1 人脸识别



## 6.1.2 人脸识别程序

(3) 按照上面介绍的损失函数 $\mathcal{L}$ 的公式来设计本例的损失函数。

#定义损失函数类

```
class LossFun(torch.nn.Module):  
    def __init__(self, margin=2.0):  
        super(LossFun, self).__init__()  
    def forward(self, o1, o2, y): #欧氏距离:  
        dist = torch.pairwise_distance(o1, o2, keepdim=True) #形状必须是两个维度以上  
        loss = torch.mean((1 - y) * torch.pow(dist, 2) + y * torch.pow(torch.clamp(2.0 - dist, min=0.0), 2))  
        return loss
```

该程序的核心代码（全部代码见教材P150页）

# 6.1 人脸识别



## 6.1.2 人脸识别程序

执行上述代码，输出如下结果：

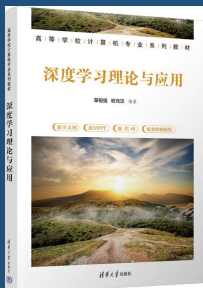
```
... ..  
97 0.021036222577095032  
98 0.020819857716560364  
99 0.010070913471281528  
训练耗时： 4.5 分钟  
一共测试了 40 张图片，准确率为 100.0%
```

加上语句 `showTwoImages(imgs, stitle, 1, 2)`即可观察对比的两张照片， 如图 6-2所示。

Similarity: 0.14



图 6-2 被程序认为是同一个人的两张人脸照片



## 本章内容

contents

### 6.1 人脸识别

### 6.2 语义分割

### 6.3 目标检测

### 6.4 生成对抗网络



## 6.2 语义分割



**图像语义分割：**将图像划分为不同的语义区域，其中每个语义区域表示同一个语义对象，不同的语义区域会加上不同的语义标签（颜色）。

例如，一张人骑自行车的图像，在语义分割后，人所在的区域和车所在的区域分别形成两个不同的语义区域，用不同的颜色标注，如图 6-3 所示。

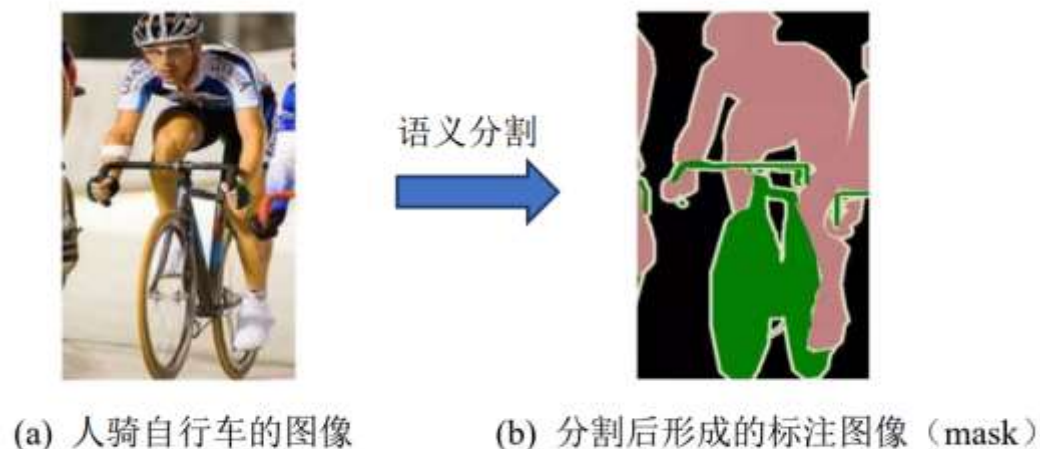


图 6-3 图像语义分割的例子

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

#### UNet 网络：

- 图像语义分割有多种方法，其中 **UNet 网络**可以说是当前最为流行的语义分割网络模型之一。由于该网络在逻辑结构上长得像字母“U”，因而称为“UNet”。
- 该网络是一个 **Encoder-Decoder** 结构网络，其基本操作是**卷积**、**最大池化**、**拼接**和**反卷积**。“U”的左边是 Encoder 部分，主要操作是卷积和最大池化；其右边是 Decoder 部分，主要操作是反卷积、拼接和卷积。

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

【例 6.2】构建一个 U 型语义分割网络。

数据集：

- 本例使用的数据集是一个关于汽车图片的数据集，来自 Kaggle Carvana Image Masking Challenge (<https://www.kaggle.com/competitions/carvana-image-masking-challenge/data>)；
- 该数据集包含 5088 张汽车图像以及与其一一对应的掩码图像，选择 4044 张图像及其掩码图像为训练集，剩下的 1044 张作为测试集；
- **训练集**的汽车图像及其掩码图像分别保存在 `./data/semantic-seg/train_imgs` 和 `./data/semantic-seg/train_masks` 目录下；
- **测试集**的汽车图像及其掩码图像则分别保存在 `./data/semantic-seg/val_imgs` 和 `./data/semantic-seg/val_masks` 目录下。

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

**语义掩码图：**汽车图像及其语义掩码图像，其对应关系是通过文件名来关联。汽车图像是  $3 \times 160 \times 240$  的图像，即其形状为**(3, 160, 240)**，图像有 3 个通道、高和宽分别为 160 和 240 个像素；**语义掩码图**是  $1 \times 160 \times 240$  的灰度图像。如图6-4所示。

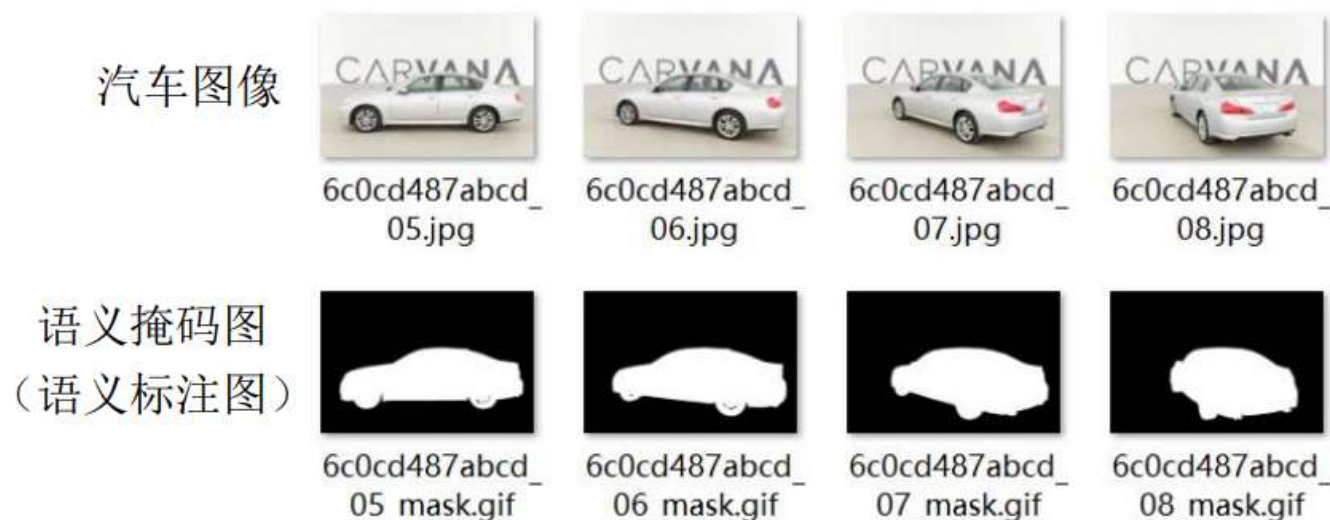


图 6-4 四张汽车图像及其语义标注图

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

UNet 网络的逻辑结构图，如图 6-5 所示。

- 左边（**Encoder 部分**）d1、d2、d3 和 d4 每个方框表示一个基本的单元模块，由卷积模块 conv 和最大池化操作 pooling 组成；
- 右边（**Decoder 部分**），u1、u2、u3 和 u4，每个单元模块由反卷积操作 convT、拼接操作 cat 和卷积模块 conv 组成；
- 图中箭头方向就是数据流动的方向，用横向虚线箭头来表示**跳层连接**
- 输出的是单通道的灰度图像，它实际上就是原来图像的语义标注图，也称为**掩码图（Mask）**。

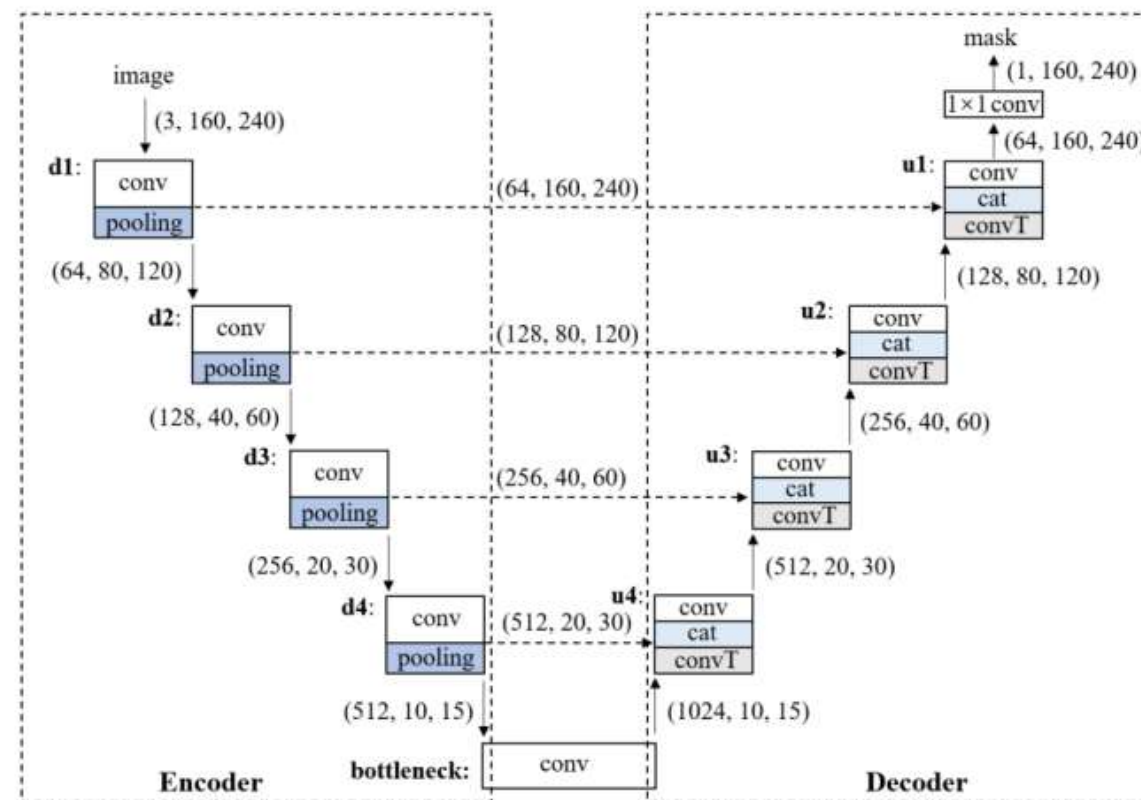


图 6-5 UNet 网络的逻辑结构

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

图 6-5 中，conv、pooling、convT、cat 分别表示卷积模块、最大池化操作、反卷积操作和拼接操作。

- **卷积模块 conv**：由两个卷积操作及相关的其他辅助操作组成，采用  $3 \times 3$  卷积核、填充数为 1、默认步长为 1，卷积模块只改变特征图的通道数，不改变其宽和高；
- **最大池化操作 pooling**：一般采用  $2 \times 2$  的池化核、默认步长为 2，经此操作后特征图的尺寸减半，通道数保持不变；
- **反卷积操作 convT**：一种特殊的卷积操作，通过补 0 的方式扩大特征图的尺寸，实现向上采样的功能；一般采用  $2 \times 2$  的卷积核、步长为 2，经过反卷积操作后，特征图的宽和高会加倍，通道数则由设定的参数决定；
- **拼接操作 cat**：对通道进行拼接，要求除了通道所在的维以外，参与拼接的两个特征图的其他维的大小要一样，拼接后只有通道数增加；



## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

构建U 型语义分割网络的步骤：

(1) 为卷积模块 **conv** 定义一个类——**OneModule**，代码如下：

```
class OneModule(nn.Module): #卷积模块类
    def __init__(self, n1, n2):
        super(OneModule, self).__init__()
        self.cnn = nn.Sequential( #定义主要操作
            nn.Conv2d(n1, n2, 3, padding=1, bias=False),
            nn.BatchNorm2d(n2),
            nn.ReLU(inplace=True),
            nn.Conv2d(n2, n2, 3, padding=1, bias=False),
            nn.BatchNorm2d(n2),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        o = self.cnn(x)
        return o
```

类 **OneModule** 设置了两个参数：n1 和 n2，分别为输入和输出特征图的通道数。包含了两个卷积层及相关的其他操作（如批量归一化等）。实例化时，通道数将被固定下来，相应的实例将成为图 6-5 中具体的卷积模块。

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

(2) 定义 U 型网络类——**UNet** 类，相应的网络层代码如下：

```
class UNet(nn.Module):
    def __init__(self, n1, n2):
        super(UNet, self).__init__()
        self.cnn1 = OneModule(n1, 64)
        self.cnn2 = OneModule(64, 128)
        self.cnn3 = OneModule(128, 256)
        self.cnn4 = OneModule(256, 512)
        self.bottleneck = OneModule(512, 1024)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.ucnn4 = OneModule(1024, 512)
        self.ucnn3 = OneModule(512, 256)
        self.ucnn2 = OneModule(256, 128)
        self.ucnn1 = OneModule(128, 64)
        self.contr4 = nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2)
        self.contr3 = nn.ConvTranspose2d(512, 256, 2, 2)
        self.contr2 = nn.ConvTranspose2d(256, 128, 2, 2)
        self.contr1 = nn.ConvTranspose2d(128, 64, 2, 2)
        self.conv1_1 = nn.Conv2d(64, 1, kernel_size=1) #1*1 卷积，用于降维
```

该程序的核心代码（全部代码见教材P155页）



## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

基本模块 d1 的功能主要是由下面两段代码实现：

```
(1) self.cnn1 = OneModule(n1, 64)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

(2) d1 = self.cnn1(d1) #d1 = x, 形状为(3, 160, 240)
    skip_cons.append(d1)
    d1 = self.pool(d1)
```

第一段代码：定义相应的网络层。

第二段代码：

- 第一句：对输入的 x 执行一次卷积模块 conv 操作，结果特征图的形状由(3, 160, 240)变为(64, 160, 240)，其中通道数改变，尺寸不变；
- 第二句：将形状为(64, 160, 240)的特征图保存在列表 skip\_cons 中，以备后面直接将其融合到深层的语义信息中；
- 第三句：执行一次  $2 \times 2$  的最大池化操作，结果特征图的形状变为(64, 80, 120)，特征图的尺寸减半。

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

与模块 d1 对应的模块u1的功能主要由下面两段代码来实现：

```
(1) self.contr1 = nn.ConvTranspose2d(128, 64, 2, 2)
    self.ucnn1 = OneModule(128, 64)
(2) u1 = self.contr1(u1) #输入的 u1 的形状为(128, 80, 120)
    u1 = torch.cat((skip_cons[0], u1), dim=1)
    u1 = self.ucnn1(u1)
```

第一段代码：定义相应的网络层。

第二段代码：

- 第一句：对 u1 执行一次反卷积操作 convT，特征图的形状由(128, 80, 120)改为(64, 160, 240)；
- 第二句：将此 特征图和列表 skip\_cons 中形状同样为(64, 160, 240)的特征图进行通道拼接，结果得到形状 为(128, 160, 240)的特征图；
- 第三句：卷积模块 conv 的操作，形状变为(64, 160, 240)

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

(3) **选择损失函数**。语义分割问题实则是像素的二分类问题，用 `nn.BCEWithLogitsLoss()` 作为损失函数。在测试时先做一次 sigmoid，再做其他的测试计算。

(4) **选择评价指标：Dice score**。指标 Dice score 的计算公式可表达如下：

$$\frac{2|\hat{Y} \cap Y|}{|\hat{Y}| + |Y|}$$

其中， $Y$  表示事先标注的语义掩码图像， $\hat{Y}$  表示模型预测时输出的语义掩码图像，二者都是二值图像，其像素值为 0 或 1。

(5) 编写其余代码，包括读取数据和打包数据、训练模型、测试模型的代码等。

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

```
class GetDataset(Dataset): #定义数据集类
    def __init__(self, img_dir, mask_dir, flag):
        self.img_dir = img_dir
        self.mask_dir = mask_dir
        self.imgs = os.listdir(img_dir)
        self.flag = flag
    def __len__(self):
        return len(self.imgs)
    def __getitem__(self, index):
        img_path = os.path.join(self.img_dir, self.imgs[index])
        mask_path = os.path.join(self.mask_dir,
            self.imgs[index].replace(".jpg", "_mask.gif"))
        img = Image.open(img_path).convert("RGB")
        mask = Image.open(mask_path).convert("L")
        img = np.array(img)
        mask = np.array(mask)
```

```
if self.flag == 'train':
    img = train_transform(img)
elif self.flag == 'test':
    img = test_transform(img)
else:
    print('Error! ')
    exit(0)
mask = test_transform(mask)
mask[mask >= 0.5] = 1.0
mask[mask < 0.5] = 0.0
return img, mask
```

该程序的部分核心代码（全部代码见教材P158页）

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

上述代码用到的库和相关模块由如下代码导入：

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from PIL import Image
from torch.utils.data import Dataset
import numpy as np
import random
import matplotlib.pyplot as plt
import os
from torch.utils.data import DataLoader
```

## 6.2 语义分割



### 6.2.1 从零开始构建语义分割网络

输出结果如下：

准确率为：  $37534540.0/40089600 = 93.63\%$

指标 Dice score 的值为: 0.83

加上语句 `showTwoimgs(imgs, ", 1, 2)`，可以看到事先标记的语义掩码图和预测的语义掩码图，如图 6-6 所示。



图 6-6 事先标记的语义掩码图和预测的语义掩码图

## 6.2 语义分割



### 6.2.2 使用预训练模型构建语义分割网络

常用的预训练模型：fcf\_resnet50、fcf\_resnet101、deeplabv3\_resnet50、deeplabv3\_resnet101、deeplabv3\_mobilenet\_v3\_large等。

【例 6.3】使用预训练模型构建一个语义分割网络。

预训练模型：**deeplabv3\_resnet101**，该模型是在COCO train2017的一个子集上进行预训练。

该模型的输出一共有 21 个类，分别是：

- 人：人
- 背景：背景
- 动物：鸟、猫、牛、狗、马、羊（6 类）
- 车辆：飞机、自行车、船、巴士、汽车、摩托车、火车（7 类）
- 室内：瓶、椅子、餐桌、盆栽植物、沙发、电视/监视器（6 类）

## 6.2 语义分割



### 6.2.2 使用预训练模型构建语义分割网络

为导出该模型，使用如下 API 调用：

```
model = models.segmentation.deeplabv3_resnet101(pretrained=True, progress=True)
```

只有一个分割对象——汽车，修改模型的 classifier 部分将类别数量改为 1。

```
model.classifier = DeepLabHead(2048, 1)
```

```
model = model.to(device)
```

```
for param in model.parameters():
```

```
    param.requires_grad = False #先冻结所有参数
```

```
for param in model.classifier.parameters():
```

```
    param.requires_grad = True #放开分类器的参数，使之可训练
```

将例 6.2 中模型的标识符“unet\_model”改为“model”，并在调用模型的时候改为下列形式：

```
pre_y = model(x)['out']
```



## 6.2 语义分割



### 6.2.2 使用预训练模型构建语义分割网络

在文件开头处引入下列模块：

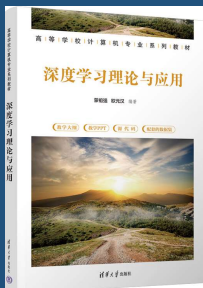
```
from torchvision import models
```

```
from torchvision.models.segmentation.deeplabv3 import DeepLabHead
```

执行后的结果如下：

准确率为：  $39080688.0/40089600 = 97.48\%$   
指标 Dice score 的值为: 0.94

与例6.2的结果相比，使用同样的训练集和测试集，使用了预训练模型的结果更好。



## 本章内容

contents

### 6.1 人脸识别

### 6.2 语义分割

### 6.3 目标检测

### 6.4 生成对抗网络

## 6.3 目标检测



### 6.3.1 从零开始构建目标检测网络

【例 6.4】构建一个用于实现目标检测的深度神经网络，使之可以识别四类路标。

数据集：

- 使用的数据集是路标检测 (**Road Sign Detection**) 数据集，其官方网站是 <https://www.kaggle.com/datasets/andrewmvd/road-sign-detection>。
- 数据集一共有**877**张图片，各图片尺寸大小不一；每张图片均有与之对应的 xml 文件，文件描述了对应图像的尺寸以及标注目标位置的 box 框的坐标。
- 将这 877 张图片均调整为  $3 \times 300 \times 447$  的图片，同时按比例缩放 box 的坐标参数；
- 新形成的 877 张图片保存在 `./data/object_detection/road_signs/images_resized` 目录下；
- 相关的类别信息、box 坐标信息则保存在 `./data/object_detection` 目录下的 `dataset.csv` 文件中，文件 `dataset.csv` 的格式如图 6-7 所示。

# 6.3 目标检测



## 6.3.1 从零开始构建目标检测网络

index	class	new_path	new_bb
0	3	.\data\object_detection\road_signs\images_resized\road0.png	46 164 174 348
1	3	.\data\object_detection\road_signs\images_resized\road1.png	66 172 297 288
2	3	.\data\object_detection\road_signs\images_resized\road10.png	3 118 295 272
3	0	.\data\object_detection\road_signs\images_resized\road100.png	3 39 254 405
4	0	.\data\object_detection\road_signs\images_resized\road101.png	10 217 291 438
5	0	.\data\object_detection\road_signs\images_resized\road102.png	42 41 275 271
6	0	.\data\object_detection\road_signs\images_resized\road103.png	24 99 271 331
7	0	.\data\object_detection\road_signs\images_resized\road104.png	11 53 284 384
8	0	.\data\object_detection\road_signs\images_resized\road105.png	109 40 282 418
9	0	.\data\object_detection\road_signs\images_resized\road106.png	112 169 169 272
10	0	.\data\object_detection\road_signs\images_resized\road107.png	27 125 174 325
11	0	.\data\object_detection\road_signs\images_resized\road108.png	156 194 169 207

图 6-7 文件 dataset.csv 的格式

第二列表示类别索引（0、1、2、3 分别表示类 speedlimit、stop、crosswalk 和类 trafficlight），第三列是图像的文件名（含路径），第四列表示 box 的坐标参数（以空格隔开）。

## 6.3 目标检测



### 6.3.1 从零开始构建目标检测网络

定义如下的数据集类：

```
class RoadDataset(Dataset):
    def __init__(self, paths, bb, y):
        self.paths = paths.values #图像的路径
        self.bb = bb.values #各图像标注框 box 的坐标，与图像一一对应
        self.y = y.values #图像类别索引
    def __len__(self):
        return len(self.paths)
    def __getitem__(self, idx):
        path = self.paths[idx]
        y_class = self.y[idx]
        img = cv2.imread(str(path)).astype(np.float32) #读取指定的图像
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) / 255 #归一化
        y_bb = self.bb[idx] #读取 box 的坐标，包含四个数字，
        #前面两个为 box 的左上角坐标，后两个为右下角坐标
        tmp_list = y_bb.split(' ')
```

```
y_bb = [int(e) for e in tmp_list]
y_bb = torch.tensor(y_bb)
img = torch.Tensor(img).permute([2,0,1])
return img, y_class, y_bb
```

该类的作用是，读取由路径参数 **paths** 指定的图像并转化为张量，同时获得该图像的类别信息和标注框的四个坐标值，并将这些数据作为一条记录返回。

## 6.3 目标检测



### 6.3.1 从零开始构建目标检测网络

读取数据并划分为训练集和测试集并打包数据：

```
df_train=pd.read_csv('./data/object_detection/dataset.csv')
X = df_train[['new_path', 'new_bb']] #包含图片的路径以及 box 的坐标
Y = df_train['class'] #类别标签
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
            random_state=42, shuffle=False) #划分为训练集和测试集，大小分别为 701 和 176
train_ds = RoadDataset(X_train['new_path'], X_train['new_bb'], y_train)
test_ds = RoadDataset(X_test['new_path'], X_test['new_bb'], y_test)
train_loader = DataLoader(train_ds, batch_size=16, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=16)
```

## 6.3 目标检测



### 6.3.1 从零开始构建目标检测网络

预测 box 框的四个坐标参数则属于回归问题，定义如下的目标检测网络：

#定义目标检测网络

```
class Detect_model(nn.Module):
```

```
    def __init__(self):
```

```
        super(Detect_model, self).__init__()
```

```
        resnet = models.resnet34(pretrained=True) #导入 resnet34
```

```
        layers = list(resnet.children())[:8] #取 resnet 的前八层
```

```
        self.features = nn.Sequential(*layers) #用于图像的特征提取
```

```
        #图像分类网络，有 4 个类别
```

```
        self.classifier = nn.Sequential(nn.BatchNorm1d(512), nn.Linear(512, 4))
```

```
        #box 坐标的预测网络，有 4 个参数需要预测
```

```
        self.bb = nn.Sequential(nn.BatchNorm1d(512), nn.Linear(512, 4))
```

```
    def forward(self, x):
```

```
        o = x
```

```
        o = self.features(o)
```

```
        o = torch.relu(o)
```

```
        o = nn.AdaptiveAvgPool2d((1,1))(o)
```

```
        o = o.reshape(x.shape[0], -1)
```

```
        return self.classifier(o), self.bb(o)
```

该程序的部分核心代码（全部代码见教材P163页）



## 6.3 目标检测



### 6.3.1 从零开始构建目标检测网络

在文件开头引入下面的模块：

```
import random
import pandas as pd
import numpy as np
import cv2
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



## 6.3 目标检测



### 6.3.1 从零开始构建目标检测网络

执行上述代码，结果输出如下：

在测试集上的分类准确率为：93.18%

用 cv2 可显示目标检测效果，图 6-8 展示了网络模型的两个检测结果，两个目标都被框住了。



图 6-8 目标检测的效果

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

**【例 6.5】** 使用 Faster-rcnn 检测一张图片中的多个目标。  
**torchvision.models** 包含了 Faster-rcnn 模型，该模型是在 COCO 数据集上训练得到的，可以识别 81 个类（背景算为一个类）。但类别的索引不是连续的，最大为 90。类别索引和名称的对应关系用字典的键-值对表示，内容如下：

```
names = {'0': 'background', '1': 'person', '2': 'bicycle', '3': 'car', '4': 'motorcycle', '5': 'airplane',  
'6': 'bus', '7': 'train', '8': 'truck', '9': 'boat', '10': 'traffic light', '11': 'fire hydrant', '13': 'stop sign', '14':  
'parking meter', '15': 'bench', '16': 'bird', '17': 'cat', '18': 'dog', '19': 'horse', '20': 'sheep', '21': 'cow',  
'22': 'elephant', '23': 'bear', '24': 'zebra', '25': 'giraffe', '27': 'backpack', '28': 'umbrella', '31': 'handbag',  
'32': 'tie', '33': 'suitcase', '34': 'frisbee', '35': 'skis', '36': 'snowboard', '37': 'sports ball', '38': 'kite', '39':  
'baseball bat', '40': 'baseball glove', '41': 'skateboard', '42': 'surfboard', '43': 'tennis racket', '44':  
'bottle', '46': 'wine glass', '47': 'cup', '48': 'fork', '49': 'knife', '50': 'spoon', '51': 'bowl', '52': 'banana',  
'53': 'apple', '54': 'sandwich', '55': 'orange', '56': 'broccoli', '57': 'carrot', '58': 'hot dog', '59': 'pizza',  
'60': 'donut', '61': 'cake', '62': 'chair', '63': 'couch', '64': 'potted plant', '65': 'bed', '67': 'dining table',  
'70': 'toilet', '72': 'tv', '73': 'laptop', '74': 'mouse', '75': 'remote', '76': 'keyboard', '77': 'cell phone', '78':  
'microwave', '79': 'oven', '80': 'toaster', '81': 'sink', '82': 'refrigerator', '84': 'book', '85': 'clock', '86':  
'vase', '87': 'scissors', '88': 'teddybear', '89': 'hair drier', '90': 'toothbrush'}
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

为利用 Faster-rcnn 检测一张图片中的多个目标，先导入该模型：

```
mobj_model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
```

```
mobj_model = mobj_model.to(device)
```

```
mobj_model.eval()
```

加载图像，转为 RGB 图像，同时归一化，加入列表 imgs 当中：

```
fn = r'data\object_detection\eating.jpg' #图像的路径
```

```
img = cv2.imread(fn) #加载图像
```

```
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
img2 = torch.Tensor(img2 / 255.).permute(2, 0, 1).to(device) #归一化
```

```
imgs = [img2] #放到列表 imgs 当中（列表中只有一个对象）
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

调用模型 `mobj_model` 对 `imgs` 中的图像进行目标检测：

```
outs = mobj_model(imgs)
```

返回的 `outs` 跟列表 `imgs` 中的对象一一对应，`outs` 中每个对象都是一个字典（dict），每个字典都包含三个键-值对。键分别为 'boxes'、'labels' 和 'scores'，它们的值分别是检测到所有目标的坐标参数（四个参数）、类别索引和检测的置信度。

下列代码可以获得图片 `img` 包含所有目标的坐标信息、类别索引和置信度值：

```
out = outs[0] #由于中只有一张图片，因此 outs 中只有一个字典，即 outs[1]是无效的
```

```
boxes = out['boxes'].data.cpu() #坐标信息
```

```
labels = out['labels'].data.cpu() #类别索引
```

```
scores = out['scores'].data.cpu() #置信度值
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

在图片上用方框将置信度值大于等于 0.8 的目标框住，代码如下：

```
for i in range(len(boxes)):
    score = scores[i].item()
    if score < 0.8:
        continue
    class_index = str(labels[i].item()) #获得下标（索引）
    class_name = names[class_index] #获得类别名称
    box = np.array(boxes[i])
    #框住目标（“框”的坐标参数在 box 中）
    cv2.rectangle(img, (box[0], box[1]), (box[2], box[3]), color=getcolor(), thickness=2)
    cv2.putText(img, text=class_name, org=(int(box[0]), #在框上方添加类别名称
        int(box[1]) - 5), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.5,
        thickness=1, lineType=cv2.LINE_AA, color=(0, 0, 255))
img = np.array(img)
cv2.imshow('11',img) #显示效果
cv2.waitKey(0)
```

`len(boxes)`、`len(labels)`和 `len(scores)`均相等且等于所检测到的目标的数量；`boxes` 有四个维度，其他张量只有一个维度。  
`scores` 取值在  $[0, 1]$  范围内，该值越大，表示检测的可信度越高。



## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

添加相应的模块引入语句及一个随机产生颜色值的函数，代码如下：

```
import torch
import torchvision
import numpy as np
import cv2
import random
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
def getcolor(): #随机产生颜色
    b = random.randint(0, 255)
    g = random.randint(0, 255)
    r = random.randint(0, 255)
    return (b, g, r)
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

执行代码，得到输出如图 6-9 所示的图片。



图 6-9 模型 mobj\_model 的多目标分割效果

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

【例 6.6】下面以人脸检测为例，运用预训练模型 Faster-rcnn 去检测特定的目标。

**目标：**通过迁移方法，使用预训练模型 **Faster-rcnn** 构建并训练一个新的深度网络，使之可以检测给定图片中的所有人脸。

**数据集：**

- 本例使用的人脸数据集来自 **WIDER FACE** 数据集 (<http://shuoyang1213.me/WIDERFACE/>)。WIDER FACE 数据集包含 32203 图像以及 393703 个标注人脸，随机选择了 9146 图像及相应的标注文件（xml 文件）作为训练集，3226 图像及相应的标注文件作为测试集。
- 训练集和测试集分别位于 `./data/face_detection/train` 目录和 `./data/face_detection/valid` 目录下。



## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

搭建基于预训练模型 Faster-rcnn 的人脸识别网络，步骤如下：

(1) 定义数据集类 GetDataset，代码如下：

```
class GetDataset(Dataset):
    def __init__(self, dir_path):
        #获取所有文件的路径和文件名
        self.all_images = [os.path.join(dir_path,file)
            for file in os.listdir(dir_path) if '.jpg' in file]
    def __getitem__(self, idx):
        img_name = self.all_images[idx]
        img = cv2.imread(img_name) #读取图像
        img = cv2.cvtColor(img,
            cv2.COLOR_BGR2RGB).astype(np.float32)
        img_resized = cv2.resize(img, (600, 600)) #调整尺寸
        img_resized /= 255.0 #归一化
        annot_fn = img_name[:-4] + '.xml' #获得标注文件名 (xml 文件)
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

```
boxes, labels = [], []
img_width = img.shape[1]
img_height = img.shape[0] tree = et.parse(annot_fn)
root = tree.getroot()
#一个框 (box) 由其左上角坐标和右下角坐标确定 (一个框确定一个人脸)
#一个坐标有两参数, 因此一个 box 有四个参数
for member in root.findall('object'): #寻找所有的 box(人脸)
    label = 1 if member.find('name').text == 'face' else 0 #1 表示人脸, 0 表示不是
    labels.append(label)
    x1 = int(member.find('bndbox').find('xmin').text) #左上 x 坐标
    y1 = int(member.find('bndbox').find('ymin').text) #左上 y 坐标
    x2 = int(member.find('bndbox').find('xmax').text) #右下 x 坐标
    y2 = int(member.find('bndbox').find('ymax').text) #右下 y 坐标
    #根据所需的 width, height, 按比例缩放框的大小
    xx1 = int((x1 / img_width) * 600)
    yy1 = int((y1 / img_height) * 600)
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

```
xx2 = int((x2 / img_width) * 600)
yy2 = int((y2 / img_height) * 600)
boxes.append([xx1, yy1, xx2, yy2])
boxes = torch.LongTensor(boxes) #张量化
labels = torch.LongTensor(labels)
target = {} #长度为 2
target["boxes"] = boxes #有多少个对象，就有多少个 box
target["labels"] = labels #每一个 box（人脸）有一个类别索引
T = transforms.Compose([transforms.ToTensor()])
img_resized = T(img_resized)
return img_resized, target
def __len__(self):
    return len(self.all_images)
```

该类的实例，由参数 `dir_path` 将数据集的路径传递进去，然后读取指定的图像文件和xml文件。xml文件采用树形结构来存放数据，其中每个box用一个标签对来刻画。

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

下面文本是 XML 文件的一部分：

```
<object>
  <name>face</name>
  <pose>Unspecified</pose>
  <truncated>0</truncated>
  <difficult>0</difficult>
  <bndbox>
    <xmin>495</xmin>
    <ymin>177</ymin>
    <xmax>532</xmax>
    <ymax>228</ymax>
  </bndbox>
</object>
```

该 xml 标签刻画一个 box（人脸）：box 的左上 x 坐标和 y 坐标分别为 495 和 177，右下 x 坐标和 y 坐标分别为 532 和 228。通过搜索所有的<object>标签即可获得所有 box 的坐标。

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

(2) 重写函数 `collate_fn`。

- 在创建 **DataLoader** 类的实例时，该实例默认会按照批量的大小 `batch_size`，将 `batch_size` 个 `x` 和 `batch_size` 个 `y` 分别组装成为长度为 `batch_size` 的两个张量。在这种默认情况下，要求个 `x` 的尺寸要彼此相同，`y` 的尺寸也要相同。
- 由于本例中各张图片中人脸的数量是不一样的，即 `target` 中 `box` 个数不一样，并且 `target` 是字典，不是张量，因此不能使用默认的函数 `collate_fn`。需要重新定义函数 `collate_fn`，使得 `DataLoader` 类的实例按照我们指定的方法组装数据。

函数 `collate_fn` 的定义代码如下：

```
def collate_fn(batch):  
    return tuple(zip(*batch)) #按列组装 batch 中的元素，这样列就变成行
```

该函数的作用是，使用 `zip(*batch)` 操作，按列组装 `batch` 中的数据。例如，如果 `batch = [['a', 1], ('b', 2), ('c', 4)]`，则 `zip(*batch)` 可视为 `[('a', 'b', 'c'), (1, 2, 4)]`。

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

(3) 加载数据，使用了上面重写过的函数 `collate_fn`，代码如下：

```
train_dataset = GetDataset('./data/face_detection/train')
test_dataset = GetDataset('./data/face_detection/valid')
train_loader = DataLoader(train_dataset, batch_size=16,
                           shuffle=True, num_workers=0, collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=True,
                          num_workers=0, collate_fn=collate_fn)
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

#### (4) 构建新的网络模型

```
face_model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True,  
                                                                    face_model.roi_heads.box_predictor = FastRCNNPredictor(1024,  
                                                                    num_classes=2) #修改最后一个输出层，把类别个数改为 2  
  
face_model = face_model.to(device)  
  
params = [p for p in face_model.parameters() if p.requires_grad]  
  
optimizer = torch.optim.SGD(params, lr=0.001, momentum=0.9, weight_decay=0.0005)
```

对于最先导入的模型 **face\_model**，用 **print(face\_model)** 语句查看其结构，修改最后一个输出层，把类别个数改为 2（因为我们只需判断是否是人脸）。在上述代码中，我们用对象 **FastRCNNPredictor** 来修改输出层。

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

(5) 编写训练代码和测试代码:

```
face_model.train() #设置为训练模式
for epoch in range(5):
    for i, (imgs, targets) in enumerate(train_loader):
        try:
            imgs = list(img.to(device) for img in imgs)
            targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
            pre_dict = face_model(imgs, targets)
            losses = sum(loss for loss in pre_dict.values()) #设计损失函数
            print(epoch, losses.item())
            optimizer.zero_grad()
            losses.backward()
            optimizer.step()
        except: #数据集中有部分标注有问题, 会产生异常, 用 try 结构忽略掉
            print('标注可能有问题.....')
torch.save(face_model, 'face_model')
```



## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

```
face_model = torch.load('face_model')
face_model.eval() #设置为测试模式
with torch.no_grad():
    for i, (imgs, targets) in enumerate(test_loader):
        imgs = list(img.to(device) for img in imgs) #img: torch.Size([3, 600, 600])
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
        pre_dict = face_model(imgs) #注：在测试模式下不需要用 targets，也不能用
        index = 0 #仅选择其中的一张照片来展示人脸检测的效果
        adict = pre_dict[index] #字典的长度为 3，关键字为 boxes, labels, scores
        img = imgs[index].permute([1, 2, 0]).data.cpu()
        img = np.array(img)
        for k, box in enumerate(adict['boxes']):
            score = adict['scores'][k]
            if score < 0.6: #忽略置信度过小的 box
                continue
            cv2.rectangle(img, (box[0], box[1]), (box[2], box[3]), color=(0, 0, 255), thickness=2)
        cv2.imshow('Face detection', img)
        cv2.waitKey(0)
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

(6) 继续添加相关模块导入语句:

```
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
import cv2
import numpy as np
import os
from xml.etree import ElementTree as et
from torch.utils.data import Dataset, DataLoader
import random
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

执行代码，可看到部分图像中人脸的检测效果，如图 6-10 所示。



图 6-10 一张图像中人脸的检测效果

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

查看指定照片中的人脸检测效果，可适当修改程度代码，加入下列代码来实现：

```
#----- 检测单张照片中的人脸 -----  
img = cv2.imread('./data/object_detection/eating.jpg') #读取图像  
#img = cv2.imread(r'data\face_detection\test\21_Festival_Festival_21_106.jpg')  
img = cv2.resize(img, (600, 600)) #调整尺寸  
img2 = img  
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).astype(np.float32)  
img /= 255.0 #归一化  
T = transforms.Compose([transforms.ToTensor()])  
img = T(img).to(device)  
imgs = [img]  
pre_dict = face_model(imgs)  
adict = pre_dict[0]  
for k, box in enumerate(adict['boxes']):  
    score = adict['scores'][k]  
    if score < 0.6:  
        continue
```

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

```
cv2.rectangle(img2, (box[0], box[1]), (box[2], box[3]),  
              color=(0, 0, 255), thickness=2)  
cv2.imshow('Face detection', img2)  
cv2.waitKey(0)
```

该代码用于检测 `./data/object_detection/` 目录下图片 `eating.jpg` 中的人脸，结果如图 6-11 所示。

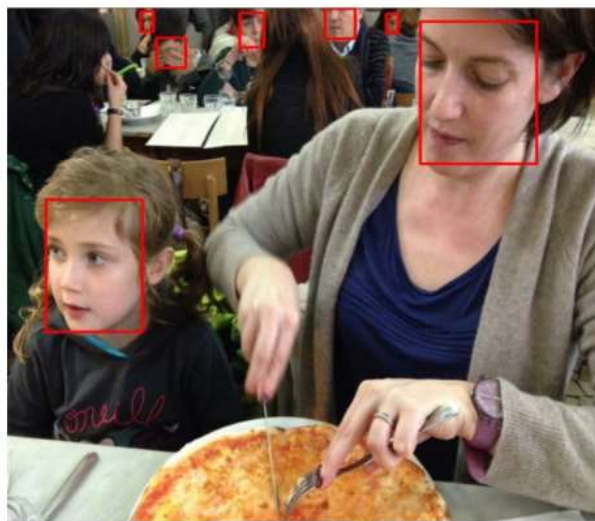


图 6-11 检测指定照片中的人脸

## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

利用例 6.6 训练产生的模型 face\_model，可实现对视频中的人脸进行检测，代码如下：

```
import torch
import torchvision.transforms as transforms
import cv2
import numpy as np
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
face_model = torch.load('face_model').to(device) #加载训练好的模型
face_model.eval()
def detect_face(img_t): #定义函数，其作用是：对输入的图像进行人脸检测
    #img_t = cv2.imread(fn) #读取图像
    img_t = cv2.resize(img_t, (600, 600)) #调整尺寸
    img_t2 = img_t
    img_t = cv2.cvtColor(img_t, cv2.COLOR_BGR2RGB).astype(np.float32)
    img_t /= 255.0 #归一化
    T = transforms.Compose([transforms.ToTensor()])
    img_t = T(img_t).to(device)
```

实现原理：视频是由一系列的帧（一幅图像）构成，取出其中的每一帧，调用模型 face\_model 对其进行人脸检测，然后放到一个播放器中去播放即可（每秒播放 24 帧左右）。

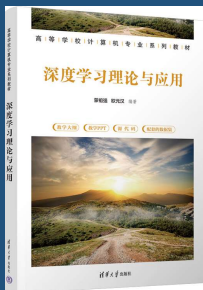
## 6.3 目标检测



### 6.3.2 使用 Faster-rcnn 构建目标检测网络

```
img_ts = [img_t]
pre_dict = face_model(img_ts) #调用训练好的模型
adict = pre_dict[0]
for k, box in enumerate(adict['boxes']):
    score = adict['scores'][k]
    if score < 0.6:
        continue
    cv2.rectangle(img_t2, (box[0], box[1]), (box[2], box[3]), color=(0, 0, 255), thickness=2)
return img_t2 #返回已标记的图像
cap = cv2.VideoCapture(0)
while True:
    ret, image_np = cap.read() #打开摄像头
    img = cv2.resize(image_np, (600, 600))
    img = detect_face(img) #调用函数
    cv2.imshow('Face detection', img )
    if cv2.waitKey(42) == 27: #当按 【esc】 键退出, 或者强行终止程序运行也可以退出
        break
```





## 本章内容

contents

### 6.1 人脸识别

### 6.2 语义分割

### 6.3 目标检测

### 6.4 生成对抗网络



## 6.4 生成对抗网络



**生成对抗网络 (Generative Adversarial Network, GAN)**：一种基于无标记样本进行网络训练的方法，训练后可以产生无限量的同类图像。

**生成器 (Generator)**：用于学习输入数据的分布，以生成类似的图像（但与训练用的图像不完全相同，更不是拷贝过来）。

**辨识器 (Discriminator)**：对图像的真实性进行“打分”：如果认为图像的真实性越高，则打的分值越接近于 1；反之，则分值越接近于 0，即分值在(0, 1)范围内。

**生成对抗网络训练的基本思想**：生成器通过参数学习，不断生成更逼真的图像，一般称为**假图像 (Fake Image)**；辨识器则通过参数学习，不断提高其判别真图像和假图像的能力。在训练过程中辨识器和生成器的损失函数值都不能太低，二者应维持在大抵相同的一个数量级上（保持一种对抗状态）。

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

**【例 6.7】** 构建一个 GAN，用于生成手写数字图片，步骤如下：

**数据集：** 手写数字图片数据集 MNIST（没有使用图片的标记信息）。

(1) 设计生成器网络，代码如下：

```
class Generator(nn.Module):
    def __init__(self, noise_len, h, w): #h, w 为图像的高和宽， 单通道图像
        super().__init__()
        self.h = h
        self.w = w
        self.fc = nn.Sequential(
            nn.Linear(noise_len, 2048),
            nn.ReLU(True),
            nn.Linear(2048, 1024),
            nn.ReLU(True),
            nn.Linear(1024, self.h*self.w),
            nn.Tanh(),
        )
```

```
def forward(self, x):
    o = x
    o = self.fc(o)
    o = o.reshape(x.shape[0], 1, self.h, self.w)
    return o
```

生成器网络由三个全连接层组成，用于生成手写数字图片，其输出的形状为(1, h, w)。

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

(2) 设计辨识器网络，代码如下：

```
class Discriminator(nn.Module):
    def __init__(self,h,w): #h,w 分别为图像的高和宽， 单通道图像
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(h*w, 512),
            nn.ReLU(0.2),
            nn.Linear(512, 256),
            nn.ReLU(0.2),
            nn.Linear(256, 1),
        )
    def forward(self, x):
        o = x
        o = o.reshape(x.shape[0],-1)
        o = self.fc(o)
        return o
```

辨识器网络由三层全连接网络组成，用于对图像的真实性进行打分，其输出为一个实数。计算损失函数值的时候，对该实数运用函数 sigmoid，使其归一化到(0,1)中。

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

(3) 创建网络实例，代码如下：

```
noise_len = 96 #噪声的维度
discriminator = Discriminator(28,28).to(device)
generator = Generator(noise_len,28,28).to(device)
d_optimizer = torch.optim.Adam(discriminator.parameters(), lr=5e-4, betas=(0.5, 0.999))
g_optimizer = torch.optim.Adam(generator.parameters(), lr=5e-4, betas=(0.5, 0.999))
```

(4) 对生成器网络和辨识器网络进行训练，代码如下：

```
for ep in range(100):
    for k, (img, _) in enumerate(train_data): #只利用样本的图像数据，不利用其标签信息
        real_img = img.to(device)
        size = real_img.shape[0]
        one_labels = torch.ones(size, 1).float().to(device)
        zero_labels = torch.zeros(size, 1).float().to(device)
        #训练辨识网络
        score_real = discriminator(real_img) #给真实图像打分
```

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

```
noise_data = torch.rand(size, noise_len).to(device) #生成噪声数据
noise_data = (noise_data-0.5)/0.5
fake_img = generator(noise_data) #利用噪声数据生成假的图像
score_fake = discriminator(fake_img) #给假图像打分
#希望识别器能够识别真图和假图
d_loss = nn.BCEWithLogitsLoss()(score_real, one_labels) \
        + nn.BCEWithLogitsLoss()(score_fake, zero_labels)
d_optimizer.zero_grad()
d_loss.backward()
d_optimizer.step() #更新识别器网络的参数
#-----
#训练生成器网络
noise_data = torch.rand(size, noise_len).to(device) #生成噪声数据
noise_data = (noise_data - 0.5) / 0.5
fake_img = generator(noise_data) #利用噪声数据生成假的图像
score_fake = discriminator(fake_img) #给假图像打分
```

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

```
#希望生成的假图像尽可能逼真
g_loss = nn.BCEWithLogitsLoss()(score_fake,one_labels)
g_optimizer.zero_grad()
g_loss.backward()
g_optimizer.step() #更新生成器网络的参数
if k % 20 == 0:
    print(ep, d_loss.item(), g_loss.item())

#-----训练结束-----
torch.save(generator,'generator')
```

(5) 添加必要的模块引入语句、数据读入语句和测试语句等，代码如下：

```
import torch
from torch import nn
from torchvision import datasets, transforms
import torchvision.transforms as tfs
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
```

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

```
import numpy as np
import matplotlib.pyplot as plt
import os #cuda
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#-----
batch_size = 512
def tfm(x):
    x = tfs.ToTensor()(x)
    return (x - 0.5) / 0.5
train_set = MNIST(root='./data/mnist2',train=True,download=True,transform=tfm)
train_data = DataLoader(train_set, batch_size=batch_size, shuffle=True)
size = 16
generator = torch.load('generator').to(device)
noise_data = torch.rand(size, noise_len).to(device) #生成噪声数据
noise_data = (noise_data-0.5)/0.5
fake_img = generator(noise_data)
```

## 6.4 生成对抗网络



### 6.4.1 生成手写数字图片

```
imgs = np.array(fake_img.data.cpu())
_, axu = plt.subplots(4, 4, figsize=(4, 4))
row_img_n = 4 #一行上显示的图像数
for i in range(row_img_n*row_img_n):
    axu[i // row_img_n][i % row_img_n].imshow(np.reshape(imgs[i],
(28, 28)), cmap='gray')
    axu[i // row_img_n][i % row_img_n].set_xticks(())
    axu[i // row_img_n][i % row_img_n].set_yticks(())
plt.suptitle('The 16 handwritten digits')
plt.show()
```

注：请出版社网站上查看或下载全部代码



## 6.4 生成对抗网络

### 6.4.1 生成手写数字图片

执行代码，随机生成 16 张手写数字图片，结果如图 6-12 所示。



图 6-12 随机生成的 16 张手写数字图片



## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

生成图像的结构比较复杂时，生成器和辨识器也随之变得复杂，以增加网络的表达能力，即网络的参数量和训练的时间都要增加。

**【例 6.8】** 构建一个 GAN，使之能够生成花卉图片。

**数据集：** 花卉图片数据集下载自 <https://www.robots.ox.ac.uk/~vgg/data/flowers/17/>，保存在 **./data/flower\_dataset** 目录下，一共有 1360 张图片。

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

生成器类和辨识器类代码如下：

(1) 定义生成器类代码：

#定义生成器类

```
class Generator(nn.Module):  
    def __init__(self):  
        super(Generator, self).__init__()  
        self.cnn = nn.Sequential( #利用反卷积来生成图像  
            nn.ConvTranspose2d(100, 512, kernel_size=(4, 4)),  
            nn.BatchNorm2d(512),  
            nn.ReLU(),  
            nn.ConvTranspose2d(512, 1024, kernel_size=(4, 4), stride=(2, 2),  
                               padding=(1, 1)),  
            #上面语句等价于： nn.ConvTranspose2d(512, 1024, 4, 2, 1)  
            nn.ReLU(),  
            nn.ConvTranspose2d(1024, 512, 4, 2, 1),  
            nn.BatchNorm2d(512),
```

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

```
        nn.ReLU(),
        nn.ConvTranspose2d(512, 64, 4, 2, 1),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.ConvTranspose2d(64, 3, 4, 2, 1),
        nn.Tanh(),
    )
    def forward(self, x):
        nn.BatchNorm2d(1024),
        o = self.cnn(x)
        return o
```

生成器网络以噪声数据作为输入，利用反卷积操作（nn.ConvTranspose2d）来生成花卉图片，一共有五个反卷积层。

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

(2) 定义辨别器类代码:

#定义判别器类

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, 4, 2, 1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Conv2d(128, 256, 4, 2, 1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),
            nn.Conv2d(256, 128, 4, 2, 1),
            nn.BatchNorm2d(128),
```

```
            nn.LeakyReLU(negative_slope=0.2),
            nn.Conv2d(128, 1, kernel_size=(4, 4)),
            nn.Sigmoid(),
        )
    def forward(self, x):
        o = self.cnn(x)
        o = o.squeeze()
        return o
```

辨别器网络有五个卷积层，用于对花卉图片的真实性打分，分值在(0, 1)范围内。

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

(3) 编写其他代码，如读入数据、训练数据、测试输出等，代码如下：

```
import numpy as np
import os
import torchvision.transforms as transforms
from torchvision.utils import save_image
from torchvision import datasets
import torch.nn as nn
import torch
from torch.utils.data import DataLoader
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#加载数据集
dataset = datasets.ImageFolder(
    root="./data/flower_dataset",
    transform=transforms.Compose([
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ]),
```

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

```
)
data_loader = DataLoader(dataset=dataset, batch_size=128, shuffle=True)
.....#生成器类和辨识器类代码所在的大致位置
generator = Generator().to(device)
discriminator = Discriminator().to(device)
g_optimizer = torch.optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
d_optimizer = torch.optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))
EPOCH = 500
for epoch in range(EPOCH):
    for i, (imgs, _) in enumerate(data_loader):
        imgs = imgs.to(device)
        one_labels = torch.ones(imgs.size(0)).to(device)
        zero_labels = torch.zeros(imgs.size(0)).to(device)
        #训练生成器
        g_optimizer.zero_grad()
        z = torch.randn(imgs.size()[0], 100, 1, 1).to(device) #生成噪声
```



## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

```
fake_imgs = generator(z) #生成假图
fake_scores = discriminator(fake_imgs)
#计算生成器的损失函数值
g_loss = torch.nn.BCELoss()(fake_scores, one_labels)
g_loss.backward()
g_optimizer.step() #更新生成器的参数
#训练判别器
d_optimizer.zero_grad()
real_imgs = imgs
real_scores = discriminator(real_imgs)
#使得辨识器能够分辨真图
real_loss = torch.nn.BCELoss()(real_scores, one_labels)
fake_scores = discriminator(fake_imgs.detach())
#使得辨识器能够分辨假图
fake_loss = torch.nn.BCELoss()(fake_scores, zero_labels)
d_loss = real_loss + fake_loss #使得辨识器能够同时分辨真图和假图
d_loss.backward()
```

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

```
d_optimizer.step() #更新辨识器的参数
print(
    "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
    % (epoch, EPOCH, i, len(data_loader), d_loss.item(), g_loss.item())
)
torch.save(generator, 'generator') generator = torch.load('generator').to(device)
generator.eval()
z = torch.randn(128, 100, 1, 1).to(device)
gen_imgs = generator(z)
save_image(gen_imgs.data[:16], "sample.png", nrow=4, normalize=True)
```

注：请从出版社网站上查看或下载全部代码

## 6.4 生成对抗网络



### 6.4.2 生成花卉图片

执行代码，生成图像效果如图 6-13 所示。



图 6-13 生成的 16 张花卉图片

## 6.4 生成对抗网络



### 6.4.3 条件性生成对抗网络

**条件性生成对抗网络（Conditional GAN）：**以一定的输入作为导向，以引导生成对抗网络生成目标图像。

**【例 6.9】** 构建一个条件性生成对抗网络，使之能够生成指定的手写数字图片。

**基本思路：**通过向量嵌入方法，为每个图像类别索引构建一个嵌入向量。需要在生成器和辨识器网络中都要进行向量嵌入，关键代码如下：

```
self.label_embedding = nn.Embedding(10, 32)
```

一个类别用一个长度为 32 的向量来表示，一共有 10 个类。将该类别向量跟噪声向量拼接，如：

```
gen_input = torch.cat((label_emb, noise), -1)
```

```
img_label = torch.cat((img, label_emb), -1)
```

## 6.4 生成对抗网络



### 6.4.3 条件性生成对抗网络

该程序的生成器和辨识器代码如下：

```
class Generator(nn.Module): #生成器类
    def __init__(self):
        super(Generator, self).__init__()
        self.label_emb = nn.Embedding(10, 32) #嵌入向量
        self.fc = nn.Sequential(
            nn.Linear(132, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2, True),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2, True),
            nn.Linear(512, 784),
            nn.BatchNorm1d(784),
            nn.LeakyReLU(0.2, True),
            nn.Linear(784, 784),
            nn.Tanh(),
        )
```

```
def forward(self, noise, labels):
    label_emb = self.label_emb(labels)
    gen_input = torch.cat((label_emb, noise), -1)
    img = self.fc(gen_input)
    img = img.reshape(img.size(0), -1, 28, 28)
    return img
```

## 6.4 生成对抗网络



### 6.4.3 条件性生成对抗网络

```
class Discriminator(nn.Module): #辨别器类
    def __init__(self):
        super(Discriminator, self).__init__()
        self.label_embedding = nn.Embedding(10, 32) #嵌入向量
        self.model= nn.Sequential(
            nn.Linear(816, 512),
            nn.LeakyReLU(0.2, True),
            nn.Linear(512, 512),
            nn.Dropout(0.4, False),
            nn.LeakyReLU(0.2, True),
            nn.Linear(512, 512),
            nn.Dropout(0.4, False),
            nn.LeakyReLU(0.2, True),
            nn.Linear(512, 1),
        )
```

```
def forward(self, img, labels):
    img = img.reshape(img.size(0), -1)
    label_emb = self.label_embedding(labels)
    #将噪声或图像向量与嵌入向量拼接
    img_label = torch.cat((img, label_emb), -1)
    out = self.model(img_label)
    return out
```

注：请从出版社网站上查看或下载全部代码

## 6.4 生成对抗网络

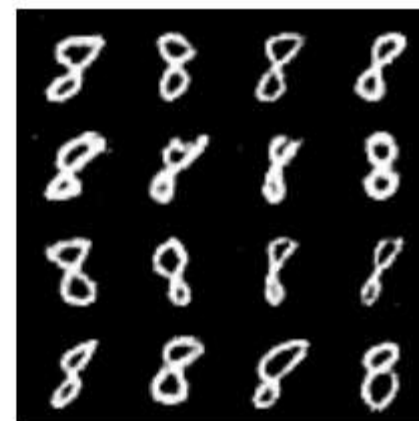
### 6.4.3 条件性生成对抗网络

修改变量 `digit` 的值，执行程序，可以产生相应的手写数字图片。

如图 6-14(a)和(b)分别是在 `digit = 5` 时和 `digit = 8` 时产生的手写数字图片。



(a) 数字 5 的图片



(b) 数字 8 的图片

图 6-14 指定生成的数字图片



## 6.5 本章小结



### 本章内容：

- 人脸识别
- 语义分割
- 预训练模型
- 目标检测
- 图像生成
- 生成对抗网络与数据增强