PyTorch-CNN

王文中

2019 年 7 月

1 使用 PyTorch 编写卷积神经网络

使用 PyTorch 开发卷积神经网络，一般需要导入以下模块：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models
```

1.1 使用 PyTorch 实现卷积网的基本运算

一个典型的卷积神经网络中主要包含了三类操作: 卷积 (Convolution)、池化 (Pooling)、全连接 (FC)，并且卷积与全连接层一般都会使用非线性响应函数 (Activation Function)。

在 PyTorch 中，torch.nn 实现了以上各种操作，它们对应的函数分别是：

- **卷积：`conv2d`**

该函数有几个重要的参数，对应了卷积层的超参数：- in_channels: 卷积层的输入层特征图的通道数 - out_channels: 卷积层输出的特征图通道数 (即卷积核数目) - kernel_size: 卷积核的大小 - padding: 输入特征图每边的填充量（填充值为 0)。默认为 padding=0，表明不扩充，对应 VALID CONVOLUTION；padding=$\frac{kernel\_size-1}{2}$，对应 SAME CONVOLUTION, 输出的特征图的尺寸与输入特征图相同
- stride: 跨度，通常为 0(该参数的默认值为 0) - bias: 布尔型参数，False 表示该层没有偏置参数，默认为 True，表示该层有偏置参数

例子：下面的代码定义了一个卷积操作，其中输入特征图的通道为 16，输出通道数为 32，卷积核大小为 3×3，使用 VALID CONVOLUTION：

`conv = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=0, bias=True)`

- **池化:MaxPool2d,AvgPool2d**

这两个函数的主要参数是 kernel_size 和 stride, 通常 stride=kernel_size。

例子：下面的代码定义了一个 2×2 的最大池化操作：

`pool = nn.MaxPool2d(kernel_size=2, stride=2)`

- **全连接：Linear**

该函数有三个参数，分别为：- in_features: 表示输入特征的维数 - out_features: 表示输出特征的维数 - bias: 布尔型参数，False 表示该层没有偏置参数，默认为 True，表示该层有偏置参数

例子：下面的代码定义了一个全连接层，输入与输出特征维数分别是 128,256:

`fc = nn.Linear(in_features=128, out_features=256, bias=True)`

- **非线性响应函数**

由 torch.nn.functional 提供，最常用的是

`torch.nn.functional.relu`

`torch.nn.functional.sigmoid`

`torch.nn.functional.softmax`。

其中第一个用作神经元的响应函数，后两个通常用作分类函数。

1.2 卷积与池化示例

下面的例子演示了卷积与池化运算。卷积运算部分用 Sobel 边缘检测器对一幅 3 通道图像计算水平和竖直方向的边缘强度。非线性运算采用 Relu（请思考：对 Sobel 结果进行 Relu 运算的意义是什么?)。池化运算使用了 2×2 最大池化。

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline
```

```
# 读入图像
im = Image.open('data/Gates-Vassar.png')
im = np.array(im, dtype='float32') / 255.0
```

```
plt.imshow(im)
```

```
<matplotlib.image.AxesImage at 0x7f8978cf3240>
```


```
# 把图像转换为 pytorch Tensor，并调整为卷积层的维度要求，即卷积层输入的张量维度为 (N,C,H,W)
# 其中，N 为图像张量的数量，C 为通道数，H，W 分别是图像的高度和宽度
print(im.shape)
im = torch.from_numpy(np.transpose(im, (2, 0, 1)))
im.unsqueeze_(0)
print(im.shape)
```

```
(360, 540, 3)
torch.Size([1, 3, 360, 540])
```

```
# 定义水平方向的 Sobel 算子
sobel_x = np.array([[-1.0, 0.0, 1.0], [-2.0, 0.0, 2.0], [-1.0, 0.0, 1.0]]).
 ↪astype(dtype='float32')
sobel_x = np.reshape(sobel_x, (1, 3, 3))
print(sobel_x.shape)
```

```
print(sobel_x)
```

```
(1, 3, 3)
[[[-1.  0.  1.]
  [-2.  0.  2.]
  [-1.  0.  1.]]]
```

```
# 定义竖直方向的 Sobel 算子
sobel_y = np.array([[-1.0, -2.0, -1.0], [0.0, 0.0, 0.0], [1.0, 2.0, 1.0]]).
 ↪astype(dtype='float32')
sobel_y = np.reshape(sobel_y, (1, 3, 3))
print(sobel_y.shape)
print(sobel_y)
```

```
(1, 3, 3)
[[[-1. -2. -1.]
  [ 0.  0.  0.]
  [ 1.  2.  1.]]]
```

```
# 把水平和竖直方向的 Sobel 算子拼接为 (Out,In,KH,HW) 的阵列
# 其中 Out=2 为卷积层输出通道数目 (即卷积核的数目，本例中为 2，表示有两个卷积核
Sobel_x,Sobel_y)
#In=3 为卷积层输入图像的通道数目 (本例中输入图像是 RGB 三通道图像，因此 In=3)
#KH，KW 是卷积核的大小，本例为 3
sobel3d = np.concatenate(
    (np.repeat(sobel_x, 3, axis=0).reshape((1, 3, 3, 3)),
     np.repeat(sobel_y, 3, axis=0).reshape((1, 3, 3, 3))),
    axis=0)
print(sobel3d.shape)
```

```
(2, 3, 3, 3)
```

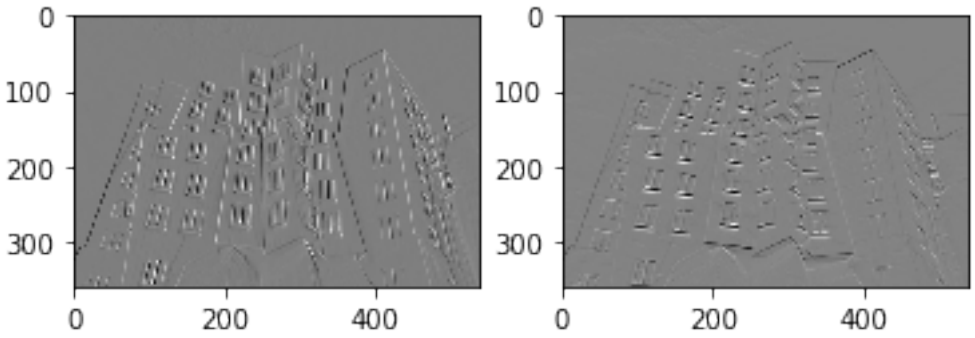
```
# 定义一个卷积操作，并设置该卷积操作的参数为 Sobel_3d:
# 注意，卷积操作的参数设置要与 Sobel3d 匹配
conv = nn.Conv2d(in_channels=3, out_channels=2, kernel_size=3, padding=1,␣
 ↪stride=[1, 1], bias=False)
```

```
conv.weight.data = torch.from_numpy(sobel3d)
```

```
# 用卷积操作对图像进行卷积:
edge = conv(im).data
print(edge.shape)
# 分别获取水平和竖直方向的边缘响应
edge_x = edge[0, 0, :, :].squeeze().numpy()
edge_y = edge[0, 1, :, :].squeeze().numpy()

plt.subplot(1, 2, 1)
plt.imshow(edge_x, cmap='gray')
plt.subplot(1, 2, 2)
plt.imshow(edge_y, cmap='gray')
```

```
torch.Size([1, 2, 360, 540])
```

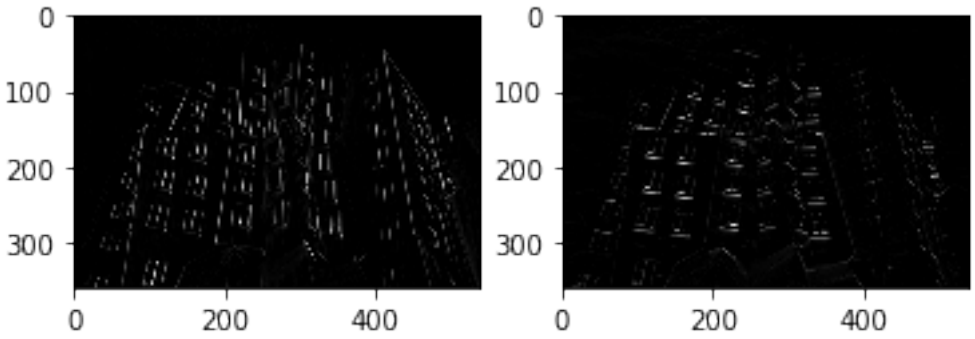
```
<matplotlib.image.AxesImage at 0x7f89780f12b0>
```




```
# 使用非线性响应函数 ReLU:
# 注意显示的图像与上图的区别
edge = F.relu(edge)
edge_x = edge[0, 0, :, :].squeeze().numpy()
edge_y = edge[0, 1, :, :].squeeze().numpy()

plt.subplot(1, 2, 1)
plt.imshow(edge_x, cmap='gray')
plt.subplot(1, 2, 2)
```

```
plt.imshow(edge_y, cmap='gray')
```

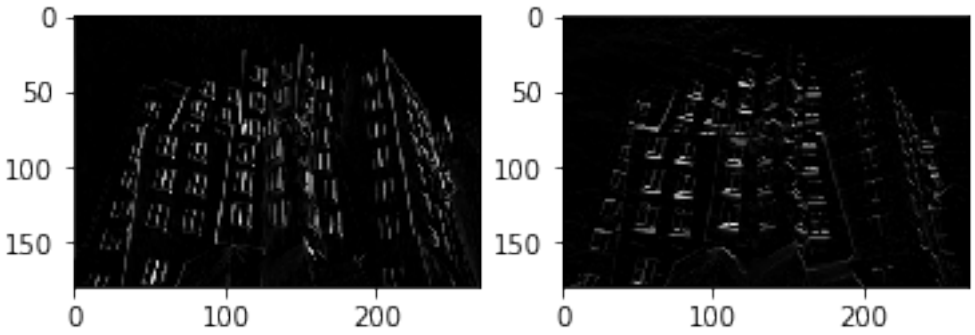
```
<matplotlib.image.AxesImage at 0x7f89780cc0f0>
```




```
# 使用最大池化操作:
pool = nn.MaxPool2d(kernel_size=2, stride=2)
edge = pool(edge)

# 注意池化后的 edge 大小的变化
print(edge.shape)

edge_x = edge[0, 0, :, :].squeeze().numpy()
edge_y = edge[0, 1, :, :].squeeze().numpy()

plt.subplot(1, 2, 1)
plt.imshow(edge_x, cmap='gray')
plt.subplot(1, 2, 2)
plt.imshow(edge_y, cmap='gray')
```

```
torch.Size([1, 2, 180, 270])
```

```
<matplotlib.image.AxesImage at 0x7f89780314e0>
```



1.3 编写简单的卷积神经网络

利用上一节介绍的运算，可以构建一个简单的卷积神经网络。

要创建一个神经网络，需要从 nn.Module 派生出一个子类，并实现该子类的初始化方法以及前向传播方法。PyTorch 通过自动求导 (AutoGrad) 机制实现后向传播。

下面的代码实现了一个简单的卷积神经网络，这个网络的输入为 28×28 的彩色图像 (CIFAR10)，输出为 10 个类别：

```
class CNN(nn.Module):
    def __init__(self):
        # 注意: 首先调用父类的初始化函数
        super(CNN, self).__init__()

        # 定义卷积、池化以及全连接操作
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=3)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=128, kernel_size=3)
        self.fc1 = nn.Linear(128, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        # 在前向函数中构造出卷积网络
        # 注意这里的 x 把不同层连接起来
```

```
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.conv3(x)

        # 使用 torch.Tensor.view 函数，把一个多维张量拉直为一个 1 维张量（向量）
        x = x.view(-1, 128)

        # 全连接层
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

1.4 训练神经网络

1.4.1 准备训练样本

训练神经网络需要先准备好训练数据，卷积网的训练数据通常都是图像数据，torchvision 模块提供了很多常见的图像数据库的封装，比如 MNIST，CIFAR，IMAGENET 等，这些标准数据库都定义为 torchvision.datasets 模块中的类，比如 MNIST 数据集对应于 torchvision.datasets.MNIST 类。

- 构造图像数据集

除了这些标准数据集之外，也可以使用 torchvision.datasets.ImageFolder 类构造图像数据集。

构造自己的图像数据集时，需要把图像按照不同类别放入不同目录中。比如：'dog' 类别的图像都放入到'dog' 目录，'cat' 类别的图像都放入'cat' 目录。'cat','dog' 目录都放入到'data' 目录。如下所示：

```
data/dog/0001.jpg
data/dog/0002.jpg
data/dog/harris.png
......
data/cat/0001.png
data/cat/ricky.png
data/cat/adam_1.png
......
```

下面的代码定义了一个数据集，该数据集中的数据位于目录'./data' 中，并且按照以上方式组织。

```
data_set = torchvision.datasets.ImageFolder(root='./data')
```

- 定义数据变换

在卷积网络的训练中，通常会对图像数据做预处理和各种变换，比如对图像像素值做规一化、图像翻转、裁剪等等。除此之外，在 PyTorch 中，图像数据要转换为 torch.Tensor 才能输入到网络中。

torchvision.transforms 模块提供了大量图像变换的类和函数。常见的有：

`torchvision.transforms.ToTensor`, 把一个 PIL 图像或 numpy.ndarray 转换为 torch.Tensor

`torchvision.transforms.ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)`，对 PIL 图像的亮度、对比度、饱和度以及色调进行随机扰动。

`torchvision.transforms.RandomCrop(size, padding=None, pad_if_needed=False, fill=0, padding_mode='constant')`，对图像进行随机裁剪，裁剪后的图像大小由参数 `size` 确定。

`torchvision.transforms.RandomHorizontalFlip(p=0.5)`，根据指定的概率 `p` 随机对图像做左右翻转

`torchvision.transforms.RandomResizedCrop(size, scale=(0.08, 1.0), ratio=(0.75, 1.3333333333333333), interpolation=2)`，根据指定的缩放系数范围 scale 和长宽比例范围 ratio，对图像进行随机缩放和比例变换，然后随机裁剪为一幅 size 大小的图像。

`torchvision.transforms.Resize(size, interpolation=2)`，对图像进行缩放，缩放后的尺寸为 size。这是最常用的图像变换，因为卷积网络的输入图像大小是恒定的，因此所有图像样本都必须要缩放到符合网络的输入尺寸。

`torchvision.transforms.Normalize(mean, std, inplace=False)`，对一个**张量**进行规范化处理，这里的两个参数分别表示均值和标准差。

除了以上变换之外，torchvision.transforms 还提供了一个类 Compose，可以把多个变换组合在一起。

下面定义个数据变换，它由随机缩放裁剪、随机水平翻转以及规范化等变换组合而成：

```
normalize = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
data_transform = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),# 把图像变换为张量
        normalize,# 注意规范化要在 ToTensor 之后
    ])
```

如果需要对图像数据做变换，那就在构造数据集对象时指定初始化参数 transform 的值。下面的代码构造了一个数据集对象 data_set，使用了上面定义的数据变换。

```
data_set = torchvision.datasets.ImageFolder(root='./data', transform=data_transform)
```

- 构造数据加载器

torch.utils.data.DataLoader 类是一个数据加载器类，实现了对数据集进行随机采样和多轮次迭代的功能。

在网络训练过程中，借助这个加载器可以非常方便的实现多轮次小批量梯度随机梯度下降训练。下面的代码构造了一个数据加载器，用于加载上面定义的数据集 data_set，并指定 batch_size 为 32，其中参数 shuffle=True 表明每一个轮次之前都会对数据做随机排列，这在随机梯度下降法中是必要的：

```
data_loader = torch.utils.data.DataLoader(dataset=data_set,
                                          batch_size=32,
                                          shuffle=True)
```

1.4.2 训练网络参数

- 定义损失函数

torch.nn 模块中定义了很多标准的损失函数。下面的代码定义了一个交叉熵损失函数对象：

```
xentropy = nn.CrossEntropyLoss()
```

损失函数对象有一个重要的方法：backward()，该方法是使用 BP 算法训练网络参数的关键，通过这个方法实现了误差反向传播。

假设网络的输出为 outputs，真实类标为 labels，下面的代码计算了两者之间的损失函数值，然后把误差反向传播：

```
loss = xentropy(outputs, labels)
loss.backward()
```

- 定义优化器

torch.optim 模块提供了很多优化算法类，比如：torch.optim.SGD，torch.optim.Adam，torch.optim.RMSprop。

构造优化器对象有两个重要参数，一个是优化变量，另一个是学习率。下面的代码构造了一个 SGD 优化器，用于优化前面定义的 CNN 网络模型的参数，设置学习率为 0.01，动量参数为 0.9：

```
net = CNN()# 构造一个 CNN 对象
optimizer = torch.optim.SGD(params=net.parameters(), lr=0.01, momentum=0.9)
```

优化器的方法 step() 用于执行一步梯度下降，更新模型参数。

- 优化过程

神经网络训练过程的一步迭代包含四个主要步骤：1）前向运算，计算给定输入的预测结果；2）计算损失函数值；3）后向传播 (BP)，计算参数梯度；4）使用梯度下降法更新参数值。

在 PyTorch 中，参数的梯度由变量维护，在每一步计算梯度之前应该先把前一步计算的梯度清除掉 (调用优化器的 zero_grad() 函数)。下面的代码演示了一步迭代过程：

```
#(inputs,labels) 为训练样本

#0) 首先要把前一步的梯度清除，即设置梯度值为 0:
optimizer.zero_grad()

#1) 前向运算，计算网络在 inputs 上的输出 outputs
outputs = net(intputs)

#2) 计算损失函数值
loss = xentropy(outputs, labels)

#3) 后向传播，计算梯度
loss.backward()

#4) 梯度下降，更新模型参数
optimizer.step()
```

- 完整的训练代码

下面的函数 train 实现了模型训练的完整过程。

注意：训练模型之前需要调用模型的 train() 函数，进入训练模式。通过调用模型和数据张量的 to 函数，实现在指定的设备 (GPU、CPU) 上训练。

```
def train(net, optimizer, loss_fn, num_epoch, data_loader, device):
    net.train()# 进入训练模式

    for epoch in range(num_epoch):
        running_loss = 0.0
```

```
        for i, data in enumerate(data_loader):
            inputs, labels = data[0].to(device), data[1].to(device)

            optimizer.zero_grad()

            outputs = net(inputs)

            loss = loss_fn(outputs, labels)
            loss.backward()

            optimizer.step()

            running_loss += loss.item()

            if i % 100 == 99:
                print('[%d, %5d] loss = %.3f' % (epoch+1, i+1, running_loss/
↪100))
                running_loss = 0.0
```

- 完整的测试代码

测试模型的精度时，需要调用模型的 eval() 函数，进入评估模式。

下面的函数 evaluate 实现了完整的模型测试过程，函数返回分类准确率。

```
def evaluate(net, data_loader, device):
    net.eval()# 进入模型评估模式
    correct = 0
    total = 0
    with torch.no_grad():
        for data in data_loader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    acc = correct / total
    return acc
```

1.4.3 实例：CIFAR10 图像分类

下面利用上一小节构造的网络 CNN，对 CIFAR10 图像进行分类。

下面的代码演示了使用 CIFAR10 对网络进行训练和测试的过程：

```
def main():

    transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
    )

    train_set = torchvision.datasets.CIFAR10(root='./data',
                                             train=True,
                                             download=True,
                                             transform=transform)

    train_loader = torch.utils.data.DataLoader(dataset=train_set,
                                               batch_size=32,
                                               shuffle=True,
                                               num_workers=2)

    test_set = torchvision.datasets.CIFAR10(root='./data',
                                            train=False,
                                            download=True,
                                            transform=transform)

    test_loader = torch.utils.data.DataLoader(dataset=test_set,
                                              batch_size=32,
                                              shuffle=False,
                                              num_workers=2)

    device = torch.device("cuda:0")

    net = CNN()
    net.to(device)

    xentropy = nn.CrossEntropyLoss()
```

```
    optimizer = optim.RMSprop(net.parameters(), lr=0.001)
    num_epoch = 20

    train(net=net,
          optimizer=optimizer,
          loss_fn=xentropy,
          num_epoch=num_epoch,
          data_loader=train_loader,
          device=device)

    train_acc = evaluate(net=net,
                         data_loader=train_loader,
                         device=device)

    test_acc = evaluate(net=net,
                        data_loader=test_loader,
                        device=device)
    print('Training Accuracy: %.2f %%' % (100 * train_acc))
    print('Test Accuracy: %.2f %%' % (100 * test_acc))


if __name__ == '__main__':
    main()
```

1.4.4 保存与加载模型

模型优化好之后，通常要把模型参数保存下来以便后续使用。

PyTorch 推荐的保存模型的方法是保存状态词典 (state_dict)，状态词典是网络模型没一层到对应参数张量的映射，通过调用 `nn.Module` 类的方法 `state_dict()` 获得模型的状态词典。

下面的代码输出了前面定义的网络 CNN 的状态词典中每个参数张量的名称和大小：

```
model = CNN()
state_dict = model.state_dict()
print('State_dict of CNN model:')
for param_tensor in state_dict:
    print(param_tensor, '\t', state_dict[param_tensor].size())
```

```
State_dict of CNN model:
```

```
conv1.weight      torch.Size([16, 3, 3, 3])
conv1.bias        torch.Size([16])
conv2.weight      torch.Size([32, 16, 5, 5])
conv2.bias        torch.Size([32])
conv3.weight      torch.Size([128, 32, 3, 3])
conv3.bias        torch.Size([128])
fc1.weight        torch.Size([128, 128])
fc1.bias          torch.Size([128])
fc2.weight        torch.Size([64, 128])
fc2.bias          torch.Size([64])
fc3.weight        torch.Size([10, 64])
fc3.bias          torch.Size([10])
```

状态词典的保存和加载分别由 `torch.save()` 和 `torch.load()` 实现。状态词典通常保存为一个.pth 文件。

如果要使用已经保存的模型参数继续训练模型或者对新的数据做预测，需要调用模型的 `load_state_dict()` 函数把从磁盘文件中读入的状态词典加载到模型中。

下面的代码演示了保存和加载状态词典到一个模型中：

```
model = CNN()
# 训练模型......

# 保存训练好的参数到文件 my_model.pth 中
state_dict = model.state_dict()
torch.save(state_dict,'my_model.pth')

# 使用保存的模型参数:
state_dict = torch.load('my_model.pth')
# 把读入的模型参数加载到模型 model 中:
model.load_state_dict(state_dict)
```

2 使用预训练模型

卷积神经网络在图像识别、图像分割等领域取得了非常大的成功，出现了很多非常好的网络模型，比如 ResNet、GoogLeNet 等图像分类模型、Faster RCNN、YOLO 等目标检测模型。

在使用卷积网络解决我们自己遇到的图像识别任务时，通常先用这些优秀的模型作为基准，再针对自己所处理的识别问题的特点做有针对性的修改和设计，因为从 0 开始设计一个全新的网络结构并且能取得非常好的效果是很困难的。

torchvision.models 模块中给出了很多优秀的卷积网络的定义，比如 VGG、ResNet、SqueezeNet、DenseNet、Inception、GoogLeNet、MobileNet 等。这些模型都有对应的类，可以通过类的构造函数创建相应的模型。如下面代码所示 (https://pytorch.org/docs/stable/torchvision/models.html)：

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
vgg16 = models.vgg16()
squeezenet = models.squeezenet1_0()
densenet = models.densenet161()
inception = models.inception_v3()
googlenet = models.googlenet()
shufflenet = models.shufflenet_v2_x1_0()
mobilenet = models.mobilenet_v2()
resnext50_32x4d = models.resnext50_32x4d()
```

PyTorch 的 Model Zoo 中还提供了这些模型在 ImageNet 等数据集上训练得到的权值参数。如果要使用这些预训练的参数，只需要在模型类的构造函数中指定参数 `pretrained` 值为 `True`。在加载模型时，会从指定的网址下载模型参数 (.pth 文件)。

下面的代码使用预训练权值创建了一个 ResNet50 模型：

```
resnet50 = models.resnet50(pretrained=True)
```

如果本地已经下载了权值 pth 文件，也可以在创建模型时，指定 `pretrained=False`(默认参数)，然后从本地磁盘文件中加载状态词典文件，并加载到模型中。如下面代码所示：

```
resnet50 = models.resnet50()
# 使用本地磁盘上的模型参数文件
state_dict = torch.load('resnet50.pth')
# 把读入的模型参数加载到模型 model 中：
resnet50.load_state_dict(state_dict)
```

使用预训练模型时要注意对输入图像做指定的规一化操作，在 ImageNet 上训练的模型，大部分输入图像的大小是 $224 \times 224 \times 3$，并且要转换为 $(3 \times 224 \times 224)$ 形状的张量。图像的像素值做规一化，均值与标准差分别为 `mean = [0.485, 0.456, 0.406],std = [0.229, 0.224, 0.225]`。

下面的代码定义了这个标准化操作：

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

2.1 使用预训练模型解决新的图像识别问题

面对一个新的图像识别任务，常见的做法是使用 ImageNet 上训练的模型做迁移学习：即使用已经训练好的模型，把最后的分类层 (Softmax) 改造为新的分类层，用新的图像数据做进一步训练，得到一个针对新任务的识别模型。

根据新任务中训练样本数量的多少，又可以分为以下三种迁移方式（以 ResNet 为例）：

1. 数据量非常少（比如每个类别只有几十张图像）：

   用 ResNet 做为特征提取器，只用新数据训练 Softmax 层分类器。

2. 数据量比较少（比如总数据量比较少，只有几千张图像）：

   固定 ResNet 低层网络权值，用新数据训练高层权值，并且使用预训练的权值初始化高层权值。

3. 数据量比较大：

   用预训练权值初始化整个网络，然后用新的数据微调全部参数。

下面以花卉识别为例，介绍如何使用上面三种方式把 ImageNet 数据集上训练的 ResNet50 模型迁移到花卉识别任务上。

我们使用 OxfordFlower102 数据集，该数据集中有 102 种不同的花卉，每一种花卉有几十张图像。

下面的代码构造了训练集与测试集的数据加载器：

```
train_dir = './data/OxfordFlowers102/train'
val_dir = './data/OxfordFlowers102/val'
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
train_set = datasets.ImageFolder(
    train_dir,
    transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
    ]))

test_set = datasets.ImageFolder(
    val_dir,
    transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
```

```
        transforms.ToTensor(),
        normalize,
    ]))


train_loader = torch.utils.data.DataLoader(train_set, batch_size=32,␣
 ↪shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=32,␣
 ↪shuffle=False, num_workers=2)
```

接下来，构造一个 ResNet50 模型，并且使用本地磁盘上的预训练权值参数：

```
net = models.resnet50(pretrained=False)
state_dict = torch.load('./model/resnet50.pth')
net.load_state_dict(state_dict=state_dict)
```

2.1.1 使用 ResNet50 作为特征提取器

这种情况下，完全不需要训练模型各个层的参数。只需要把各个参数的属性 `requires_grad` 设置为 `False` 即可。如下方代码所示：

```
for param in net.parameters():
    param.requires_grad=False
```

2.1.2 部分微调 ResNet50

这种情况下，我们对 ResNet50 的高层参数做微调，因此，只需要把低层参数的的属性 `requires_grad` 设置为 `False` 即可。

下面的代码列出了网络的各个层参数名称：

```
for name, _ in net.named_parameters():
    print(name)
```

```
conv1.weight
bn1.weight
bn1.bias
layer1.0.conv1.weight
layer1.0.bn1.weight
layer1.0.bn1.bias
layer1.0.conv2.weight
```

```
layer1.0.bn2.weight
layer1.0.bn2.bias
layer1.0.conv3.weight
layer1.0.bn3.weight
layer1.0.bn3.bias
layer1.0.downsample.0.weight
layer1.0.downsample.1.weight
layer1.0.downsample.1.bias
 ......
 layer4.2.bn1.weight
layer4.2.bn1.bias
layer4.2.conv2.weight
layer4.2.bn2.weight
layer4.2.bn2.bias
layer4.2.conv3.weight
layer4.2.bn3.weight
layer4.2.bn3.bias
fc.weight
fc.bias
```

假设只需要训练 layer4 的权值，那么可以把 layer1~layer3 的参数的梯度属性置为 False，如下方代码所示：

```
exclude_layers = ['layer1', 'layer2', 'layer3']# 这些层不训练
for name, param in net.named_parameters():
    for layer in exclude_layers:
        if name.startswith(layer):
            param.requires_grad = False
            break
```

2.1.3 微调全部参数

这种情况不需要做特殊处理，net 的各层参数的 `requires_grad` 属性默认为 `True`。

2.1.4 改造分类层

原始 ResNet 模型是用于对 ImageNet 图像做分类的，分类层有 1000 个输出节点。在 Oxford-Flower102 数据集上，要把这个分类层改造为输出 102 个节点。这只需要把原始网络中的 FC 层替换为新的全连接层即可，如下代码所示：

```
num_classes = 102
featureSize = net.fc.in_features
net.fc = nn.Linear(featureSize, num_classes)
```

2.1.5 训练网络

网络的训练过程与前文一样。