


# pytorch图像分类篇：2.pytorch官方demo实现一个分类器(LeNet)

原创 Fun' 2020-07-05 22:10:12 5168 收藏 113 原力计划

分类专栏: # pytorch图像分类 文章标签: 神经网络 Deep Learning 图像分类 pytorch 深度学习 机器学习

 pytorch图像分类 专栏收录该内容

11 订阅 6 篇文章 订阅专栏

## 前言

最近在b站发现了一个非常好的 **计算机视觉 + pytorch实战** 的教程，相见恨晚，能让初学者少走很多弯路。因此决定按着up给的教程路线：图像分类→目标检测→...一步步学习用 pytorch 实现深度学习在 cv 上的应用，并做笔记整理和总结。

up主教程给出了pytorch和tensorflow两个版本的实现，我暂时只记录pytorch版本的笔记。

参考内容来自：

- up主的b站链接：<https://space.bilibili.com/18161609/channel/index>
- up主将代码和ppt都放在了github：<https://github.com/WZMIAOMIAO/deep-learning-for-image-processing>
- up主的CSDN博客：[https://blog.csdn.net/qq\\_37541097/article/details/103482003](https://blog.csdn.net/qq_37541097/article/details/103482003)

## pytorch官网入门demo——实现一个图像分类器

参考：

1. 哔哩哔哩：pytorch官方demo(Lenet)
2. pytorch官网demo（中文版戳[这里](#)）
3. pytorch中的卷积操作详解

注：关于pytorch等环境的搭建，可以参考我之前写的 [win10+MX350显卡+CUDA10.2+PyTorch 安装过程记录](#)

## demo的流程

1. model.py ——定义LeNet网络模型
2. train.py ——加载数据集并训练，训练集计算loss，测试集计算accuracy，保存训练好的网络参数
3. predict.py——得到训练好的网络参数后，用自己找的图像进行分类测试

## 1. model.py

先给出代码，模型是基于LeNet做简单修改，层数很浅，容易理解：

```
1 # 使用torch.nn包来构建神经网络。
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class LeNet(nn.Module):                # 继承于nn.Module这个父类
6     def __init__(self):                # 初始化网络结构
7         super(LeNet, self).__init__()  # 多继承需用到super函数
8         self.conv1 = nn.Conv2d(3, 16, 5)
9         self.pool1 = nn.MaxPool2d(2, 2)
10        self.conv2 = nn.Conv2d(16, 32, 5)
11        self.pool2 = nn.MaxPool2d(2, 2)
12        self.fc1 = nn.Linear(32*5*5, 120)
13        self.fc2 = nn.Linear(120, 84)
14        self.fc3 = nn.Linear(84, 10)
15
16    def forward(self, x):                # 正向传播过程
```

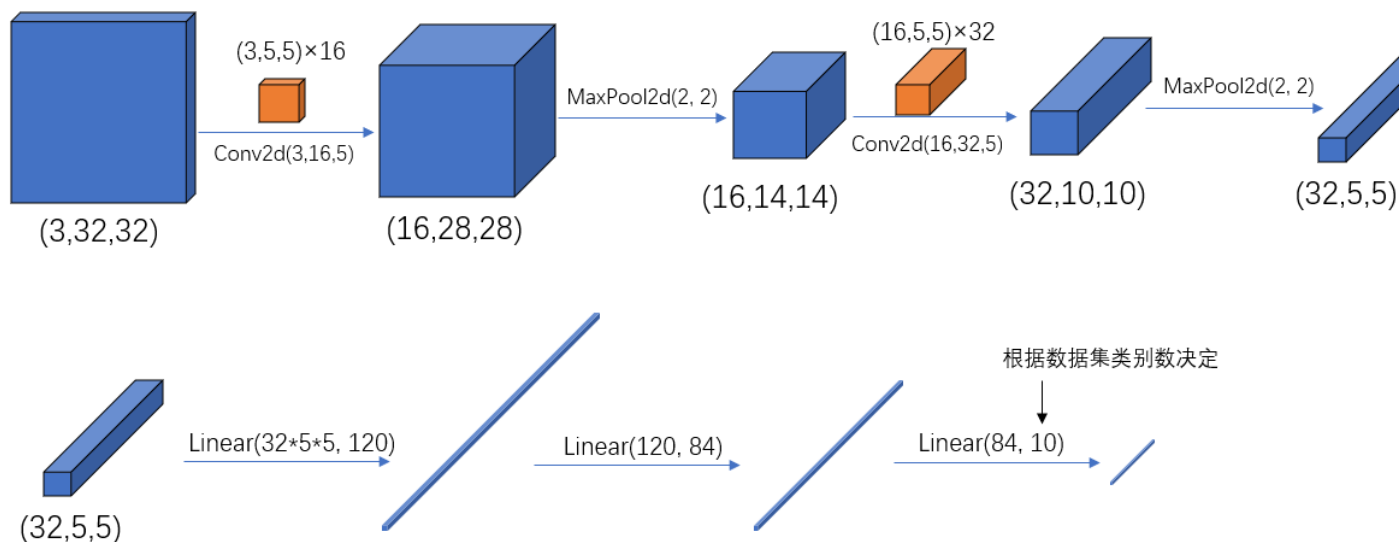
```

17 |         x = F.relu(self.conv1(x))      # input(3, 32, 32) output(16, 28, 28)
18 |         x = self.pool1(x)              # output(16, 14, 14)
19 |         x = F.relu(self.conv2(x))      # output(32, 10, 10)
20 |         x = self.pool2(x)              # output(32, 5, 5)
21 |         x = x.view(-1, 32*5*5)         # output(32*5*5)
22 |         x = F.relu(self.fc1(x))         # output(120)
23 |         x = F.relu(self.fc2(x))         # output(84)
24 |         x = self.fc3(x)                 # output(10)
25 |     return x

```

需注意：

- pytorch 中 tensor（也就是输入输出层）的通道排序为：`[batch, channel, height, width]`
- pytorch中的卷积、池化、输入输出层中参数的含义与位置，可配合下图一起食用：



## 1.1 卷积 Conv2d

我们常用的卷积（Conv2d）在pytorch中对应的函数是：

```
1 | torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

一般使用时关注以下几个参数即可：

- **in\_channels**: 输入特征矩阵的深度。如输入一张RGB彩色图像，那in\_channels=3
- **out\_channels**: 输出特征矩阵的深度。也等于卷积核的个数，使用n个卷积核输出的特征矩阵深度就是n
- **kernel\_size**: 卷积核的尺寸。可以是int类型，如3代表卷积核的height=width=3，也可以是tuple类型如(3, 5)代表卷积核的height=3, width=5
- **stride**: 卷积核的步长。默认为1，和kernel\_size一样输入可以是int型，也可以是tuple类型
- **padding**: 补零操作，默认为0。可以为int型如1即补一圈0，如果输入为tuple型如(2, 1)代表在上下补2行，左右补1列。

#### Parameters

- **in\_channels** (*int*) – Number of channels in the input image
- **out\_channels** (*int*) – Number of channels produced by the convolution
- **kernel\_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding\_mode** (*string, optional*) – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

#### Shape:

- Input:  $(N, C_{in}, H_{in}, W_{in})$
- Output:  $(N, C_{out}, H_{out}, W_{out})$  where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

[https://blog.csdn.net/m0\\_37867091](https://blog.csdn.net/m0_37867091)

附上pytorch官网上的公式：

经卷积后的输出层尺寸计算公式为：

$$Output = \frac{(W - F + 2P)}{S} + 1$$

- 输入图片大小  $W \times W$  (一般情况下Width=Height)
- Filter大小  $F \times F$
- 步长  $S$
- padding的像素数  $P$

若计算结果不为整数呢？参考 [pytorch中的卷积操作详解](#)

## 1.2 池化 MaxPool2d

最大池化 (MaxPool2d) 在 pytorch 中对应的函数是：

```
1 | MaxPool2d(kernel_size, stride)
```

## 1.3 Tensor的展平：view()

注意到，在经过第二个池化层后，数据还是一个三维的Tensor (32, 5, 5)，需要先经过展平后(32\*5\*5)再传到全连接层：

```
1 | x = self.pool2(x)           # output(32, 5, 5)
2 | x = x.view(-1, 32*5*5)      # output(32*5*5)
3 | x = F.relu(self.fc1(x))     # output(120)
```

## 1.4 全连接 Linear

全连接 (Linear) 在 pytorch 中对应的函数是：

```
1 | Linear(in_features, out_features, bias=True)
```

## 2. train.py

### 2.1 导入数据集

导入包

```
1 import torch
2 import torchvision
3 import torch.nn as nn
4 from model import LeNet
5 import torch.optim as optim
6 import torchvision.transforms as transforms
7 import matplotlib.pyplot as plt
8 import numpy as np
9 import time
```

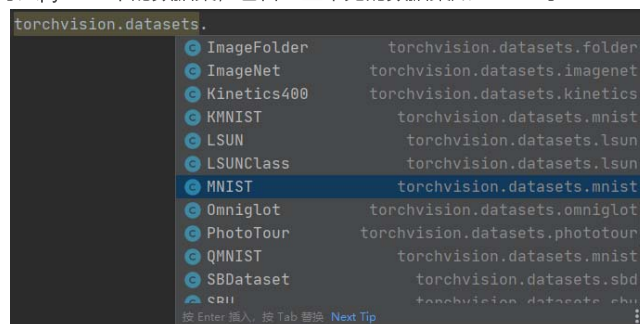
### 数据预处理

对输入的图像数据做预处理，即由shape (H x W x C) in the range [0, 255] → shape (C x H x W) in the range [0.0, 1.0]

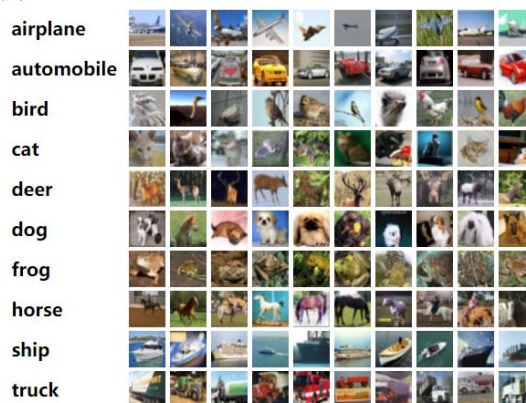
```
1 transform = transforms.Compose(
2     [transforms.ToTensor(),
3      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

### 数据集介绍

利用 `torchvision.datasets` 函数可以在线导入pytorch中的数据集，包含一些常见的数据集如MNIST等



此demo用的是CIFAR10数据集，也是一个很经典的图像分类数据集，由 Hinton 的学生 Alex Krizhevsky 和 Ilya Sutskever 整理的一个用于识别普适物的小型数据集，一共包含 10 个类别的 RGB 彩色图片。



### 导入、加载 训练集

```
1 # 导入50000张训练图片
2 train_set = torchvision.datasets.CIFAR10(root='./data',          # 数据集存放目录
3                                           train=True,           # 表示是数据集中的训练集
4                                           download=True,        # 第一次运行时为True, 下载数据集, 下载完成后改为False
5                                           transform=transform) # 预处理过程
6 # 加载训练集, 实际过程需要分批次 (batch) 训练
7 train_loader = torch.utils.data.DataLoader(train_set,          # 导入的训练集
8                                             batch_size=50,     # 每批训练的样本数
9                                             )
```

## 导入、加载 测试集

```
1 # 导入10000张测试图片
2 test_set = torchvision.datasets.CIFAR10(root='./data',
3                                         train=False, # 表示是数据集中的测试集
4                                         download=False, transform=transform)
5 # 加载测试集
6 test_loader = torch.utils.data.DataLoader(test_set,
7                                           batch_size=10000, # 每批用于验证的样本数
8                                           shuffle=False, num_workers=0)
9 # 获取测试集中的图像和标签, 用于accuracy计算
10 test_data_iter = iter(test_loader)
11 test_image, test_label = test_data_iter.next()
```

## 2.2 训练过程

名词	定义
epoch	对训练集的全部数据进行一次完整的训练, 称为一次 epoch
batch	由于硬件算力有限, 实际训练时将训练集分成多个批次训练, 每批数据的大小为 batch_size
iteration 或 step	对一个batch的数据训练的过程称为一个 iteration 或 step

以本demo为例, 训练集一共有50000个样本, batch\_size=50, 那么完整的训练一次样本: iteration或step=1000, epoch=1

```
1 net = LeNet() # 定义训练的网络模型
2 loss_function = nn.CrossEntropyLoss() # 定义损失函数为交叉熵损失函数
3 optimizer = optim.Adam(net.parameters(), lr=0.001) # 定义优化器 (训练参数, 学习率)
4
5 for epoch in range(5): # 一个epoch即对整个训练集进行一次训练
6     running_loss = 0.0
7     time_start = time.perf_counter()
8
9     for step, data in enumerate(train_loader, start=0): # 遍历训练集, step从0开始计算
10         inputs, labels = data # 获取训练集的图像和标签
11         optimizer.zero_grad() # 清除历史梯度
12
13         # forward + backward + optimize
14         outputs = net(inputs) # 正向传播
15         loss = loss_function(outputs, labels) # 计算损失
16         loss.backward() # 反向传播
17         optimizer.step() # 优化器更新参数
18
19         # 打印耗时、损失、准确率等数据
20         running_loss += loss.item()
21         if step % 1000 == 999: # print every 1000 mini-batches, 每1000步打印一次
22             with torch.no_grad(): # 在以下步骤中 (验证过程中) 不用计算每个节点的损失梯度, 防止内存占用
23                 outputs = net(test_image) # 测试集传入网络 (test_batch_size=10000), output维度为[10000,10]
24                 predict_y = torch.max(outputs, dim=1)[1] # 以output中值最大位置对应的索引 (标签) 作为预测输出
25                 accuracy = (predict_y == test_label).sum().item() / test_label.size(0)
26
27                 print('%d, %5d] train_loss: %.3f test_accuracy: %.3f' % # 打印epoch, step, loss, accuracy
28                     (epoch + 1, step + 1, running_loss / 500, accuracy))
29
30                 print('%f s' % (time.perf_counter() - time_start)) # 打印耗时
31                 running_loss = 0.0
32
33 print('Finished Training')
34
35 # 保存训练得到的参数
36 save_path = './Lenet.pth'
37 torch.save(net.state_dict(), save_path)
```

打印信息如下:

```
1 [1, 1000] train_loss: 1.537 test_accuracy: 0.541
2 35.345407 s
3 [2, 1000] train_loss: 1.198 test_accuracy: 0.605
4 40.532376 s
5 [3, 1000] train_loss: 1.048 test_accuracy: 0.641
6 44.144097 s
7 [4, 1000] train_loss: 0.954 test_accuracy: 0.647
8 41.313228 s
9 [5, 1000] train_loss: 0.882 test_accuracy: 0.662
10 41.860646 s
11 Finished Training
```

## 2.3 使用GPU/CPU训练

使用下面语句可以在有GPU时使用GPU，无GPU时使用CPU进行训练

```
1 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
2 print(device)
```

也可以直接指定

```
1 device = torch.device("cuda")
2 # 或者
3 # device = torch.device("cpu")
```

对应的，需要用 `to()` 函数来将Tensor在CPU和GPU之间相互移动，分配到指定的device中计算

```
1 net = LeNet()
2 net.to(device) # 将网络分配到指定的device中
3 loss_function = nn.CrossEntropyLoss()
4 optimizer = optim.Adam(net.parameters(), lr=0.001)
5
6 for epoch in range(5):
7
8     running_loss = 0.0
9     time_start = time.perf_counter()
10    for step, data in enumerate(train_loader, start=0):
11        inputs, labels = data
12        optimizer.zero_grad()
13        outputs = net(inputs.to(device)) # 将inputs分配到指定的device中
14        loss = loss_function(outputs, labels.to(device)) # 将Labels分配到指定的device中
15        loss.backward()
16        optimizer.step()
17        running_loss += loss.item()
18        if step % 1000 == 999:
19            with torch.no_grad():
20                outputs = net(test_image.to(device)) # 将test_image分配到指定的device中
21                predict_y = torch.max(outputs, dim=1)[1]
22                accuracy = (predict_y == test_label.to(device)).sum().item() / test_label.size(0) # 将test_Label分配到指定的device中
23
24                print('%d, %5d] train_loss: %.3f test_accuracy: %.3f' %
25                      (epoch + 1, step + 1, running_loss / 1000, accuracy))
26
27                print('%f s' % (time.perf_counter() - time_start))
28                running_loss = 0.0
29
30    print('Finished Training')
31
32    save_path = './Lenet.pth'
33    torch.save(net.state_dict(), save_path)
```

打印信息如下：

```
1 cuda
2 [1, 1000] train_loss: 1.569 test_accuracy: 0.527
3 18.727597 s
4 [2, 1000] train_loss: 1.235 test_accuracy: 0.595
5 17.367685 s
```

```
6 | [3, 1000] train_loss: 1.076 test_accuracy: 0.623
7 | 17.654908 s
8 | [4, 1000] train_loss: 0.984 test_accuracy: 0.639
9 | 17.861825 s
10 | [5, 1000] train_loss: 0.917 test_accuracy: 0.649
11 | 17.733115 s
12 | Finished Training
```

可以看到，用GPU训练时，速度提升明显，耗时缩小。

### 3. predict.py

```
1 | # 导入包
2 | import torch
3 | import torchvision.transforms as transforms
4 | from PIL import Image
5 | from model import LeNet
6 |
7 | # 数据预处理
8 | transform = transforms.Compose(
9 |     [transforms.Resize((32, 32)), # 首先需resize成跟训练集图像一样的大小
10 |      transforms.ToTensor(),
11 |      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
12 |
13 | # 导入要测试的图像（自己找的，不在数据集中），放在源文件目录下
14 | im = Image.open('horse.jpg')
15 | im = transform(im) # [C, H, W]
16 | im = torch.unsqueeze(im, dim=0) # 对数据增加一个新维度，因为tensor的参数是[batch, channel, height, width]
17 |
18 | # 实例化网络，加载训练好的模型参数
19 | net = LeNet()
20 | net.load_state_dict(torch.load('Lenet.pth'))
21 |
22 | # 预测
23 | classes = ('plane', 'car', 'bird', 'cat',
24 |            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
25 | with torch.no_grad():
26 |     outputs = net(im)
27 |     predict = torch.max(outputs, dim=1)[1].data.numpy()
28 | print(classes[int(predict)])
```

输出即为预测的标签。

其实预测结果也可以用 **softmax** 表示，输出10个概率：

```
1 | with torch.no_grad():
2 |     outputs = net(im)
3 |     predict = torch.softmax(outputs, dim=1)
4 | print(predict)
```

输出结果中最大概率值对应的索引即为 预测标签 的索引。

```
1 | tensor([[2.2782e-06, 2.1008e-07, 1.0098e-04, 9.5135e-05, 9.3220e-04, 2.1398e-04,
2 |          3.2954e-08, 9.9865e-01, 2.8895e-08, 2.8820e-07]])
```



Fun'

关注

45

25

113



专栏目录

### 相关推荐

关于我们 招贤纳士 广告服务 开发助手 400-660-0108 kefu@csdn.net 在线客服 工作时间 8:30-22:00

公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心  
网络110报警服务 中国互联网举报中心 家长监护 Chrome商店下载 ©1999-2021北京创新乐知网络技术有限公司 版权与免责声明 版权申诉  
出版物许可证 营业执照