全连接神经网络实战

PYTORCH 版

Dezeming Family

Dezeming

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列书，可以从我们的网站 *[https://dezeming.top/]* 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

0.1 本书前言

尽管各种关于神经网络 python 实战的资料已经很多了，但是这些资料也各有优点和缺点，有时候也很难让新手有比较好的选择。

当我们明白何为“神经网络”，何为“反向传播”时，我们就已经具备了开始搭建和训练网络的能力。此时，最好的方法就是给我们一个由简及难的程序示例，我们能够快速搭建出一个网络，我们可以开始训练，以及指导如何计算训练后的结果准确率等信息。

这也是我要开始写这么一本小书的初衷，我会把本小书控制在 3 小时的学习时间之内。也就是说，只知道一丁点 python 知识和神经网络的概念，而从未使用过 pytorch 的读者，只需要三个小时，就可以用 pytroch 搭建一个有模有样的神经网络系统了。

几年前，我在 Mooc 的《人工智能实战——Tensorflow 笔记》这门课上入门了 tensorflow，我很喜欢这种讲授的风格。尽管这门课讲到后面，代码量也因为过于巨大从而导致上课节奏不好控制，但它的目的达到了——学习者可以快速入门 tensorflow。而后来，因为很多项目的源码都是基于 pytorch 的，我也开始转战 pytorch。

pytorch 其实更为简单，只是很多教程会一次性给出过多内容，导致读者难以区分什么是必要的，什么是非必要的。这构成了我写这本书的初衷——从基础到模型结构的步步递进。我们不会一次性给出一大堆可选择的内容导致学习变得复杂化，而是用到什么就讲什么。本书不可避免要参考 [2] 的讲解方式，但我们对讲解顺序和内容，以及程序代码都做了大量的改进。说了那么多，总之，我们的目标是写一个最好的最容易上手的 pytorch 入门教程——从全连接网络开始。

书中的示例代码在网站页面可以找到。每节末尾会提示“本节代码见 chapterX.py”。

20211006：完成本书第一版。

1. 准备章节

本章节将神经网络训练之前的准备工作进行全面介绍。但我们并不介绍如何安装 *pytorch*，一是由于不同版本的 *pytorch* 会依赖于不同的 *cuda* 工具，二是因为官网资料非常齐全，也有很多博客来介绍，因此没有必要赘述。

1.1 导入 pytorch

首先我们需要明白一个术语：tensor。这个词被翻译为中文叫张量。1 维标量是一种 tensor；向量也是一种 tensor；而一些微分量，例如梯度、导数等也都是 tensor；矩阵也是张量；多张矩阵或者多张图像也是张量（3 维张量）。我们在做实验时，可以将 tensor 理解为是“data”。

我们需要先导入 pytorch，顺便导入 numpy：

```
import torch
import numpy as np
```

现在我们尝试将 list 或者 np.array 转换为 pytorch 的数组：

```
data1 = [[1, 2],[3, 4]]
data_tensor = torch.tensor(data1)
print(data_tensor.shape)
np_array1 = np.array(data1)
data_tensor = torch.from_numpy(np_array1)
print(data_tensor.shape)
```

输出都是：

```
torch.Size([2, 2])
```

对于二维 tensor 之间的相乘，@ 和 *.matmul* 函数表示矩阵相乘；* 和 *.mul* 表示矩阵元素之间相乘：

```
y = data_tensor @ data_tensor.T
print(y)
y = data_tensor * data_tensor
print(y)
```

输出分别是：

```
[[ 5, 11],
[11, 25]]

[[ 5, 11],
[11, 25]]
```

tensor 可以转化为 numpy：

```
np_array2 = data_tensor.numpy()
print(np_array2)
```

关于 pytorch 的基础准备工作就是这些。下一步我们开始导入数据。

前两节的源码参见 chapter1.py。

1.2 导入样本数据

把数据输入这个环节进行模块化和独立化，对于简化模型训练很有好处。pytorch 中有两个模块是用来导入数据的：torch.utils.data.Dataset 以及 torch.utils.data.DataLoader。

Dataset 存储样本以及它们的标签等信息，Dataset 可以使用预加载的数据集（例如 mnist），也可以使用自定义的数据集；而 DataLoader 是把样本进行访问和索引的工具，它实现了迭代器功能，也就是说它可以依次将 *batch_size* 数量的样本导出。

注意，前面已经导入过的 python 包我们就不再重复导入了。

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

前面说过，Dataset 可以存储自定义数据，我们可以继承 Dataset 类，在子类中实现一些固定功能的函数，这样就相当于封装了自己的数据为 Dataset 类型。为了方便起见，我们先描述如何使用预加载数据，然后第二章就开始构建神经网络模型。等第四章我们再描述如何自定义数据集。

我们一次写一个完整的程序来把数据可视化一下：

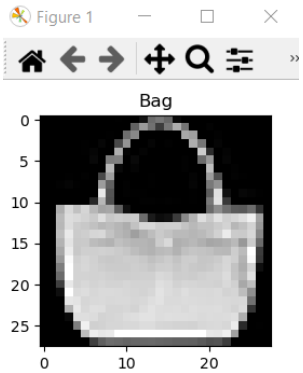
```
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
training_data = datasets.FashionMNIST(
    root="data",
    train=True, #用来训练的数据
```

```
    download=True, #如果根目录没有就下载
    transform=ToTensor()
)
test_data = datasets.FashionMNIST(
    root="data",
    train=False, #用来测试的数据
    download=True, #如果根目录没有就下载
    transform=ToTensor()
)
#把数据显示一下
labels_map = { 0: "T-Shirt", 1: "Trouser", 2: "Pullover", 3: "Dress"
    , 4: "Coat",
    5: "Sandal", 6: "Shirt", 7: "Sneaker", 8: "Bag", 9: "Ankle Boot"
        , }
import matplotlib.pyplot as plt
figure = plt.figure()
# 抽取索引为100的数据来显示
img, label = training_data[100]
plt.title(labels_map[label])
#squeeze函数把为1的维度去掉
plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

datasets 是 torchvision 的对象，它返回的数据就是 pytorch 的 Dataset 类型的。

参数 *transform* 表示导出的数据应该怎么转换，我们还可以使用参数 *target_transform* 表示导出的数据标签应该怎么转换。

注意显示时我们调用了 squeeze() 函数，这是因为原来的数据维度是 (1,28,28) 的三维数据，使用.squeeze() 函数可以把为 1 的维度去掉，即 shape 变为 (28,28)。程序得到显示结果：



随后我们再把数据导入到 DataLoader 里面：

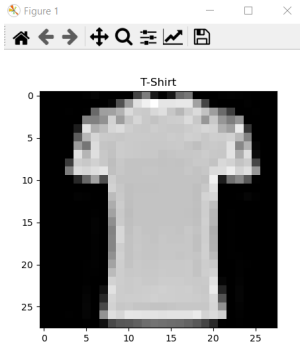
```
# batch_size: 每次迭代取出的数据量
# shuffle: 洗牌的意思，先把数据打乱，然后再分为不同的batch
```

```
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=
    True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

我们写点程序检测一下 DataLoader：

```
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
# 取出索引为23的数据
img = train_features[23].squeeze()
# 把train_labels先转为numpy，然后再取索引23的标签
label = train_labels.numpy()[23]
plt.title(labels_map[label])
plt.imshow(img, cmap="gray")
plt.show()
```

程序得到显示结果：



数据有时候并不适合直接丢进网络进行训练，因此我们需要把数据进行转换。由于 pytorch 会自动完成一些工作，因此我们没有必要自己去转换，比如像这样：

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True, #用来训练的数据
    download=True, #如果根目录没有就下载
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.
        float).scatter_(0, torch.tensor(y), value=1))
)
```

transform 是对数据的转换，ToTensor() 函数将 PIL 图像或者 NumPy 的 ndarray 转换为 FloatTensor 类型的，并且把图像的每个像素值压缩到 [0.0,1.0] 之间。

target_transform

是标签的转换，分类中我们需要将标签表示为向量的形式，例如一共有三类，则表示为：

$$[1 \quad 0 \quad 0] \tag{1.2.1}$$

$$[0 \quad 1 \quad 0] \tag{1.2.2}$$

$$[0 \quad 0 \quad 1] \tag{1.2.3}$$

Lambda 函数就是应用用户定义的 lambda 函数，首先使用 zeros 函数创建一个 10 维数组，然后调用.scatter 函数为每个向量的第 label 个索引赋值为 1。

由于 pytorch 的网络训练会自动帮你进行转换，所以我们不需要自己去操作，因此并不需要设置 $target\_transform$。

前两节的源码参见 chapter1.py。

2. 构建神经网络

本章描述如何构建神经网络模型。

2.1 基本网络结构

我们定义神经网络的结构。在 pytorch 中要想使用神经网络，需要继承 nn.Module：

```
class NeuralNetwork(nn.Module):
  def __init__(self):
      super(NeuralNetwork, self).__init__()
      # 把数组降到1维
      self.flatten = nn.Flatten()
      # 定义网络的计算顺序
      self.linear_relu_stack = nn.Sequential(
          nn.Linear(28*28, 512),
          nn.ReLU(), #使用ReLU做激活函数
          nn.Linear(512, 512),
          nn.ReLU(),
          nn.Linear(512, 10),
      )
  def forward(self, x):
      x = self.flatten(x)
      logits = self.linear_relu_stack(x)
      return logits
model = NeuralNetwork()
print(model)
```

这里使用 Relu 做激活函数，本想用最基本的 Sigmoid 函数，但 Sigmoid 函数训练效果实在是太差，因此还是换成 ReLU 吧。有的人可能会疑惑输出为什么不用在 forward 里面定义 Softmax 或者 Cross-Entropy，这是因为这些东西是在 NeuralNetwork 之外定义：

```
#损失函数为交叉熵
loss_function = nn.CrossEntropyLoss()
# 学习率
learning_rate = 1e-3
# 优化器为随机梯度下降
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

现在我们先构思一下训练的主体程序，该程序训练 10 轮，并且每轮会训练一次，然后测试一次准确率。训练函数的输入是训练数据、神经网络体、损失函数计算体以及优化器；测试函数不需要优化器：

```
epochs = 10
for t in range(epochs):
    print(f"Epoch␣{t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_function, optimizer)
    test_loop(test_dataloader, model, loss_function)
print("Done!")
```

然后就是训练和测试的程序，训练一轮的程序如下：

```
def train_loop(dataloader, model, loss_function, optimizer):
  size = len(dataloader.dataset)
  for batch, (X, y) in enumerate(dataloader):
      # Compute prediction and loss
      pred = model(X)
      loss = loss_function(pred, y)
      # Backpropagation
      optimizer.zero_grad() #梯度归0w
      loss.backward()
      optimizer.step()
      if batch % 100 == 0:
          loss, current = loss.item(), batch * len(X)
          print(f"loss:␣{loss:>7f}␣␣[{current:>5d}/{size:>5d}]")
```

在反向传播中，首先令优化器的梯度归 0，因为梯度是默认相加的（.step 函数会把 loss.backward() 得到的梯度累加到原来的梯度值上），因此为了防止梯度不断累积，需要归 0。

我们打印输出一下 len(dataloader.dataset)，发现训练集有 60000 个数据。因为每个 batch 个数为 64 个数据，因此训练集要训练 938 次，我们每 100 次输出一下。

测试集的程序如下：

```
def test_loop(dataloader, model, loss_function):
  size = len(dataloader.dataset)  # 10000
  print(f"size:{size}")
  num_batches = len(dataloader)
  print(f"num_batches:{num_batches}")
  test_loss, correct = 0, 0
  with torch.no_grad():
      for X, y in dataloader:
          pred = model(X)
          test_loss += loss_function(pred, y).item()
          correct += (pred.argmax(1) == y).type(torch.float).sum().
             item()
  test_loss /= num_batches
  correct /= size
  print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {
     test_loss:>8f} \n")
```

测试集有 10000 个数据，with torch.no_grad() 的意义是不再构建计算图。因为 pytorch 在运算时会首先构建计算图，用于后面的反向传播算法等操作，我们测试正确率时不需要构建计算图。pred.argmax(1) 表示向量中最大的一个数的索引，即为我们预测的当前数据类别。然后，.sum 函数得到一个 batch 里的所有预测正确的次数。

现在我们的网络已经可以训练了，我们可以看到，最终训练的模型在测试集上的准确率为百分之 70 左右。

本节源码参见 chapter2.py。

2.2 使用 cuda 来训练网络

首先，我们先定义用来训练网络的设备：

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
#把网络模型移到cuda中
model = NeuralNetwork().to(device)
print(model)
```

如果 cuda 可用，就会输出 cuda。

之后在训练和测试的每个 for 循环中，要把数据也迁移到 cuda 中：

```
for batch, (X, y) in enumerate(dataloader):
  # Compute prediction and loss
  X = X.cuda()
```

```
y = y.cuda()
......
```

我们不用担心数据释放的问题，因为 cuda 会自动管理不再引用它的内存空间，因此每轮训练完以后，cuda 内的内存都会被重新赋值使用，而不会使 cuda 的内存不断增长。

需要注意的是，把数据移动到 cuda 中也是比较浪费时间的，所以实际情况如何选择网络训练设备也是需要慎重考虑的。

本节源码参见 chapter2-2.py。

3. 更完善的神经网络

本章我们的目标是把神经网络做的更完善。

3.1 模型的加载与保存

有时候我们希望将训练了一定轮数的模型参数保存起来，这个时候我们就需要保存和恢复模型了。

model.state_dict() 函数可以得到模型的状态字典，里面包含了模型的参数权重与 bias 等信息，我们可以用下面的代码来保存和恢复模型：

```
# 保存模型
torch.save(model.state_dict(), path)
# 恢复模型
model.load_state_dict(torch.load(path))
```

其中，path 是保存模型的路径。有时候我们希望能同时保存模型的一些其他信息，比如 epoch 和优化器的类型，这时我们可以生成一个状态字典：

```
# 保存模型
state = {'model': model.state_dict(), 'optimizer': optimizer.
    state_dict(), 'epoch': epoch}
torch.save(state, path)
# 恢复模型
checkpoint = torch.load(path)
model.load_state_dict(checkpoint['model'])
optimizer.load_state_dict(checkpoint['optimizer'])
epoch = checkpoint['epoch']
```

现在我们修改一下我们的程序。首先我们先训练模型并保存，然后再把导出的模型参数导入到新模型中并测试正确率：

```
epochs = 10
for t in range(epochs):
    print(f"Epoch␣{t+1}\n-------------------------------")
    path = './model' + str(t) +'.pth'
    train_loop(train_dataloader, model, loss_function, optimizer)
    state = {'model': model.state_dict(), 'optimizer': optimizer.
        state_dict(), 'epoch': t}
    torch.save(state, path)
print("Done!")
#把最后一次训练得到的模型导入到模型中
path = './model' + str(9) +'.pth'
checkpoint = torch.load(path)
model2 = NeuralNetwork().to(device)
model2.load_state_dict(checkpoint['model'])
optimizer.load_state_dict(checkpoint['optimizer'])
test_loop(test_dataloader, model2, loss_function)
```

model2 的预测正确率为 70.5%，证明我们的模型保存和恢复机制是正确的。

本节代码见 chapter3.py。

3.2 初始化网络权重-方法一

我们通过自定义初始化函数，来实现对网络参数的初始化。有时候，好的初始化可以为网络的训练带来极大好处。

在 NeuralNetwork 内部定义函数：

```
def weight_init(self):
  #遍历网络的每一层
  for m in self.modules():
      #如果该层是线性连接层
      if isinstance(m, nn.Linear):
          print(m.weight.shape)
          print(m.bias.shape)
          #权重分布符合正态分布
          m.weight.data.normal_(0.0, 1)
          #偏置归0
          m.bias.data.zero_()
```

注意 bias 是权重，因为当前层的 bias 会连接下一层的每个神经元，所以 bias 的 shape 是下一层神经元个数。调用也很简单，定义网络对象后直接调用即可：

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

我们开始训练，发现第一个 epoch 训练的结果正确率就达到了 78%，而最终训练结果能达到百分之 81%。说明合理地初始化权重具有很重要的意义。

如果 weight 全都初始化成同一个值，例如：

```
m.weight.data.fill_(0.05)
```

训练效果就会变得特别差。

还有另一种初始化方式：

```
m.weight.data=torch.ones(m.weight.data.shape)*0.05
```

我们可以生成 tensor 进行赋值来初始化。这种方式可以让我们将任意方式生成的的权重赋值给网络模型。

本节代码见 chapter3-2.py。

3.3 初始化网络权重-方法二和三

为了防止初始化方式混在一起大家难以区分，因此初始化权重的方式分了两节来讲解。

方法二是通过修改状态字典的方法来修改权重的：

```
def weight_init(self):
    t=self.state_dict()
    for key, value in self.state_dict().items():
        # 根据命名来筛选，只要线性部分
        if 'linear' in key:
            # 方法一：
            a=torch.normal(0,0.1,t[key].shape)
            t[key].copy_(a)
            # 方法二：
            nn.init.normal_(value, 0,0.1)
            #nn.init.constant_(value, 0.01)
```

注意，我们把 key 打印出来以后得到：

```
linear_relu_stack.0.weight
linear_relu_stack.0.bias
linear_relu_stack.2.weight
linear_relu_stack.2.bias
linear_relu_stack.4.weight
```

```
  linear_relu_stack.4.bias
```

也就是说 weight 和 bias 是分开的，for 迭代中依次得到 weight 和 bias。

方法三通过参数来修改：

```
def weight_init2(self):
  for name, param in self.named_parameters():
      if 'linear' in name:
          a=torch.normal(0, 0.1, param.shape)
          #方法一：
          t[name].copy_(a)
          #方法二：
          #param.data.copy_(a)
```

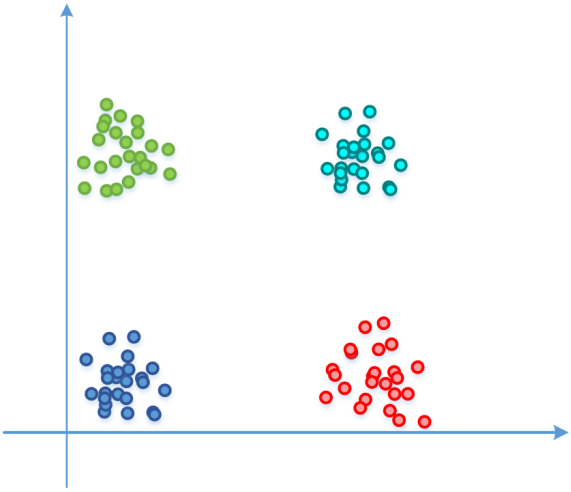
本节代码见 chapter3-3.py。

4. 构建自己的数据集

本章我们的目标是把构建自己的数据集，并来测试和可视化。

4.1 自定义 Variable 数据与网络训练

假如我们并没有图像数据，我们自己创造一些数据，并用它们来分类。
由于本节内容相对比较多，大家可以直接参考本节代码。本节代码见 chapter4.py。
我们假设一共有四类数据，分布如下：



设横纵坐标为 (x,y)，最左下角的一类中，$x \in [0,1], y \in [0,1]$。最右上角的一类中，$x \in [4,5], y \in [4,5]$。我们首先生成一下这四个类别：

```
import torch
import numpy as np
# 生成数据
def dataGenerate(data,label):
```

```
    for idata in data:
        if idata[0] < 0.5:
            # 把小于0.5的值压缩到 [0,1] 之间
            idata[0] = idata[0] * 2
            if idata[1] < 0.5:
                # 把小于0.5的值压缩到 [0,1] 之间
                idata[1] = idata[1] * 2
                label.append(0)
            else:
                # 把大于0.5的值压缩到 [4,5] 之间
                idata[1] = (idata[1] - 0.5) * 2.0 + 4.0
                label.append(1)
        else:
            # 把大于0.5的值压缩到 [4,5] 之间
            idata[0] = (idata[0] -0.5)*2.0 + 4.0
            if idata[1] < 0.5:
                # 把小于0.5的值压缩到 [0,1] 之间
                idata[1] = idata[1] * 2
                label.append(2)
            else:
                # 把大于0.5的值压缩到 [4,5] 之间
                idata[1] = (idata[1] - 0.5) * 2.0 + 4.0
                label.append(3)

data = np.random.rand(20000, 2)
label=[]
dataGenerate(data, label)
x_data = torch.tensor(data).float()
y_data = torch.tensor(label)
print(x_data)
print(y_data)
from torch.autograd import Variable
x_data,y_data = Variable(x_data),Variable(y_data)

data2 = np.random.rand(1000, 2)
label2=[]
dataGenerate(data2, label2)
x_data2 = torch.tensor(data2).float()
y_data2 = torch.tensor(label2)
```

```
x_data2,y_data2 = Variable(x_data2),Variable(y_data2)
```

生成数据的程序是将 [0,0.5] 的数据扩展到 [0,1.0]，将 [0.5,1] 的数据扩展到 [4,5]，总共分为 4 类。

数据要生成为 Variable 形式才能用于训练。*x_data,y_data* 表示训练集的数据和标签；*x_data2,y_data2* 表示测试集的数据和标签。

网络结构相对来说比较简单，由于并不是图像数据，所以设置的网络神经元数量大大减少：

```
import torch.nn as nn
class NeuralNetwork(nn.Module):
    def __init__(self):
      super(NeuralNetwork, self).__init__()
      # 把数组降到1维
      self.flatten = nn.Flatten()
      self.linear_relu_stack = nn.Sequential(
          nn.Linear(2, 7),
          nn.ReLU(),
          nn.Linear(7, 8),
          nn.ReLU(),
          nn.Linear(8, 4),
      )
    def weight_init(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                m.weight.data.normal_(0.0, 1.0)#.fill_(0.05)
                m.bias.data.zero_()
    def forward(self, x):
      #x = self.flatten(x)
      logits = self.linear_relu_stack(x)
      return logits
model = NeuralNetwork()
model.weight_init()
loss_function = nn.CrossEntropyLoss()
learning_rate = 1e-3
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

训练和测试程序改为：

```
def train_loop(dataset, label, model, loss_function, optimizer):
  size = len(dataset) #训练集有60000个数据
  # Compute prediction and loss
  pred = model(dataset)
```

```
    loss = loss_function(pred, label)
    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    loss = loss.item()
    print(f"loss: {loss:>7f}")

def test_loop(dataset, label, model, loss_function):
    size = len(dataset)  # 10000
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X,y in zip(dataset, label):
            pred = model(X)
            pred = torch.unsqueeze(pred, 0)
            y = torch.tensor([y])
            test_loss += loss_function(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().
                item()
    test_loss /= size
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {
        test_loss:>8f} \n")
```

注意，测试中，pred 的形式是 tensor([a b c d]) 的类型，需要用 unsqueeze 转化为二维 tensor，然后才能用于计算损失函数。

调用的程序与前面一样，我们一次将全部数据用于训练，并训练 1000 轮：

```
epochs = 1000
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(x_data, y_data, model, loss_function, optimizer)
    test_loop(x_data2, y_data2, model, loss_function)
print("Done!")
```

在训练中，可以看到，准确率在波动中不断上升，最终准确率能达到 98% 左右。我们下一节把训练过程中准确率的变化与最终在测试集上的分类结果可视化一下。

4.2 准确率的可视化

我们定义一个存放训练过程的 list：

```
correctCurve = []
```

然后在 test_loop 函数中把每次计算好的 correct 假如该列表：

```
correctCurve.append(correct * 100.0)
```

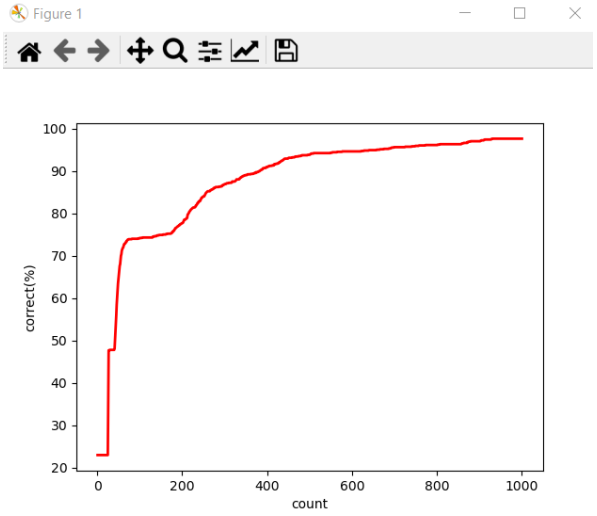
生成轮数横坐标：

```
count = list(range(epochs))
count = [i + 1 for i in count]
```

结果可视化的程序如下：

```
import matplotlib.pyplot as plt
fig ,ax= plt.subplots()
ax.set_xlabel('count')
ax.set_ylabel('correct(%)')
plt.plot(count,correctCurve,color='red',linewidth=2.0,linestyle='-')
plt.show()
```

我们可以得到结果（我训练了很多次，有时候训练 1000 轮以后的正确率只有 80%，有时候能到百分之 99%）：



本节代码见 chapter4-2.py。

4.3 分类结果的可视化

现在我们希望能够看一下训练的分类结果，为了方便起见我们的源码里删除了上一节的内容。

我们先实现一下模型的保存功能，否则每次都重新训练会非常麻烦：

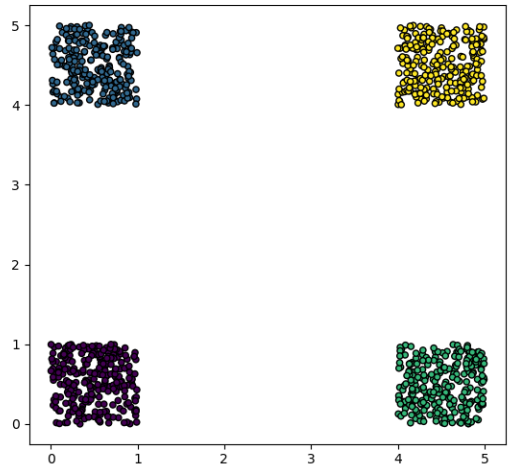
```
# 从第900轮恢复模型（取决于保存好的模型文件）
path = './model' + str(900) +'.pth'
```

```
checkpoint = torch.load(path)
model.load_state_dict(checkpoint['model'])
#optimizer.load_state_dict(checkpoint['optimizer'])
epochs = checkpoint['epoch']
# 再训练1000轮（或者不再训练，而是直接使用模型来预测，取决于你的需求）
for t in range(epochs, epochs+1000):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(x_data, y_data, model, loss_function, optimizer)
    test_loop(x_data2, y_data2, model, loss_function)
    #每训练100轮保存一次模型
    if t % 100 == 0:
        path = './model' + str(t) + '.pth'
        state = {'model': model.state_dict(), 'optimizer': optimizer.
            state_dict(), 'epoch': t}
        torch.save(state, path)
print("Done!")
```

我们先把正确的数据类别可视化一下：

```
# 取出每一列
data2_x1 = [i[0] for i in data2]
data2_x2 = [i[1] for i in data2]
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
plt.scatter(data2_x1,data2_x2,s = 20, c=label2, zorder=2, linewidths
    =1, edgecolors='k')
plt.show()
```

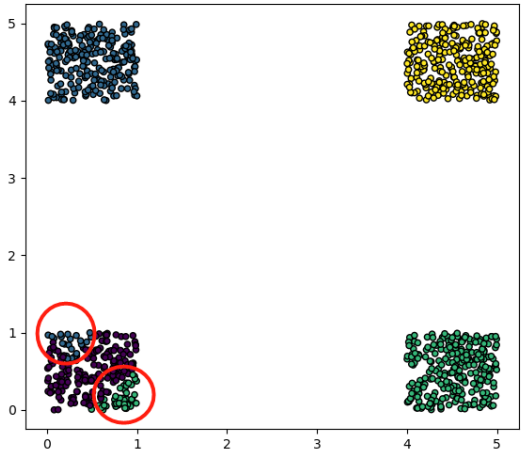
得到可视化结果：



可以看到不同的类别显示都是正确的。下面我们把模型分类的结果显示一下：

```
label2_test = []
with torch.no_grad():
    for X, y in zip(x_data2, y_data2):
        pred = model(X)
        # print(pred.argmax(0))
        pred = torch.unsqueeze(pred, 0)
        label2_test.append(pred.argmax(1).item())
        y = torch.tensor([y])
fig, ax = plt.subplots()
plt.scatter(data2_x1,data2_x2,s = 20, c=label2_test, zorder=2,
   linewidths=1, edgecolors='k') #画点
plt.show()
```

运行程序，发现这些地方有分类错误的数据：



本节代码见 chapter4-3.py。

4.4 自定义 Dataset 数据集

假设我们现在已经产生了 *x_data,y_data* 以及 *x_data2,y_data2*，我们要把它们进行封装。我们只需要继承 Dataset，然后实现三个函数即可，即初始化函数，求长度的函数以及根据索引返回某一个样本的函数：

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
class CustomDataset(Dataset):
    def __init__(self, data, label, transform=None, target_transform=
        None):
        self.transform = transform
        self.target_transform = target_transform
```

```
        self.datas = data
        self.labels = label
    def __len__(self):
        return len(self.labels)
    def __getitem__(self, idx):
        data_ = self.datas[idx]
        label_ = self.labels[idx]
        if self.transform:
            data_ = self.transform(data_)
        if self.target_transform:
            label_ = self.target_transform(label_)
        return data_, label_
```

之后我们分别实现训练集和测试集的 DataLoader：

```
train_dataset = CustomDataset(data, label)
train_dataloader = DataLoader(train_dataset, batch_size=50, shuffle=
    True)
test_dataset = CustomDataset(data2, label2)
test_dataloader = DataLoader(test_dataset, batch_size=50, shuffle=True
    )
```

神经网络的结构不需要修改，但我们需要修改训练函数和测试函数。首先是训练函数：

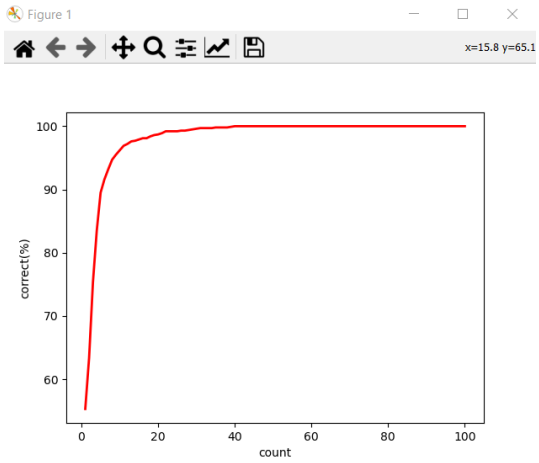
```
def train_loop(dataloader, model, loss_function, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        X = torch.tensor(X, dtype=torch.float32)
        y = torch.tensor(y)
        pred = model(X)
        loss = loss_function(pred, y)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f} batch:[{batch:>5d}] len(X):[{len(
                X)}] [{current:>5d}/{size:>5d}]")
```

注意只有 32 位浮点数才能送入网络训练，因此我们需要把数据转为 float32 位的；y 也要转为 tensor。

然后是测试函数：

```
def test_loop(dataloader, model, loss_function):
    size = len(dataloader.dataset)  # 10000
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X = torch.tensor(X, dtype=torch.float32)
            y = torch.tensor(y)
            pred = model(X)
            test_loss += loss_function(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().
                item()
    test_loss /= size
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
        {test_loss:>8f} \n")
```

然后我们就可以开始训练了。可以发现，分成不同的 batch 分批送入网络训练得到的结果大大变好，有时候只需要几十轮，分类正确率就能达到百分之百（相比于以前训练 1000 轮才能到百分之 90，好了太多）：



本节代码见 chapter4-4.py。

4.5 总结

历时不到三天的业余时间完成了这本小书的计划和写作，从叙述的详细程度和阅读学习的简单性而言，还是比较符合我的预期的。

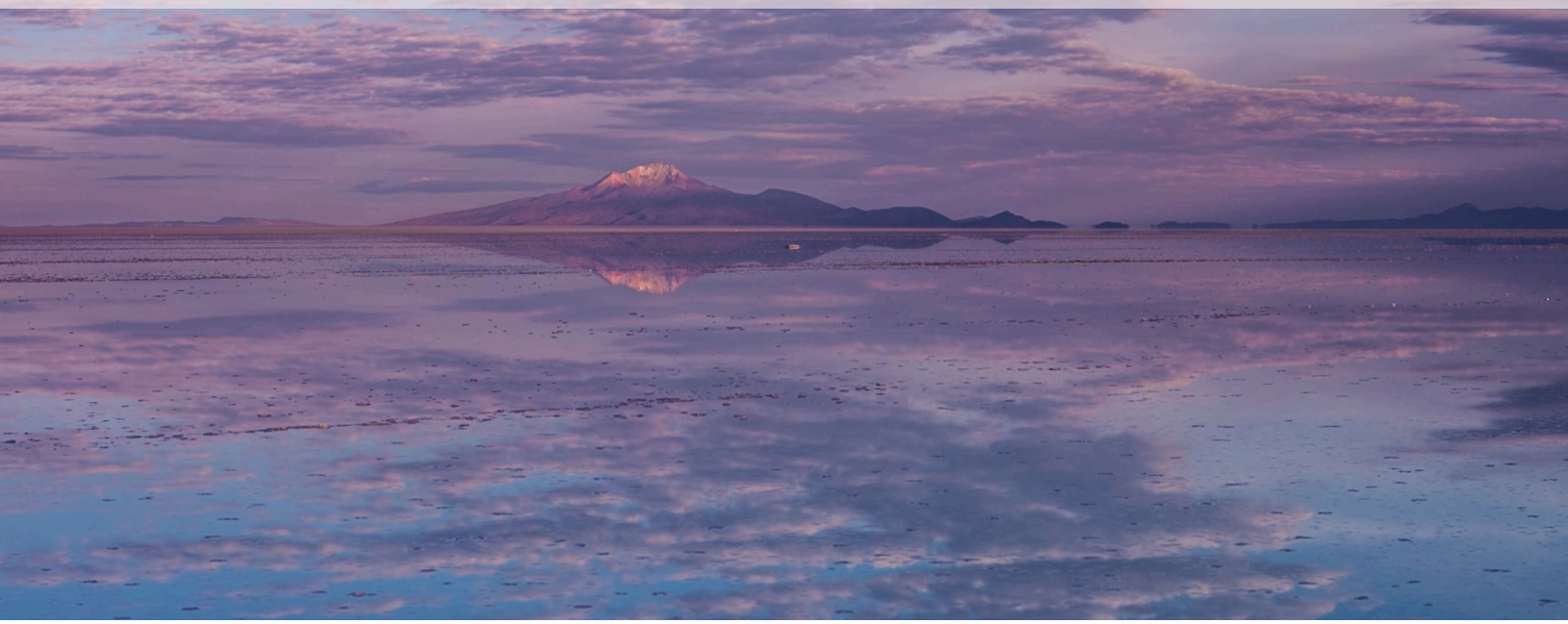
本来这部入门小书的原名叫做《卷积网络实战》，但我思考再三，感觉卷积网络不应该作为入

门来学习的神经网络，因为卷积层包含了一些关于感受野方面的思想。如果给一个网络构建了卷积层，它也就失去了普适性（尤其是不再适用于我们第四章的人造数据集了）。

书中对一些常见的优化方法（例如指数衰减学习率、L1 和 L2 正则化等）并没有实现，一是因为借助 pytorch 实现非常简单，二是为了保证网络的简洁性。

我相信这部小书比以往任何您阅读过的 pytorch 入门书都要通俗和容易上手，在前人的教程参考下，我主要对本书的叙事顺序和结构安排费了比较多的心力，而知识结构并没有做太多的改动。

Bibliography



[1] https://zhuanlan.zhihu.com/p/48982978

[2] https://pytorch.org/tutorials/index.html

[3] https://www.jianshu.com/p/1cd6333128a1 模型保存与恢复

[4] https://www.cnblogs.com/tangjunjun/p/13731276.html 预定义权重