无监督学习与 AutoEncoder 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

在《PCA 主成分分析法》中，我们描述了降维的原理和意义，但是在很多情况下，由于数据本身过于复杂，使用 PCA 降维得到的结果不一定是我们最想要的结果。

最简单书降维方法就是人工去除一些不重要的维度，但有时候所有维度似乎都比较重要，一般方法很难降低维度，而高维度数据又很难进行分析。我们可以这么思考，构建一个神经网络，使其输入和输出是同样的结构，而网络的某个中间层只有较少的神经元，这样我们就可以把中间层之前的网络作为一个编码器，把中间层之后的网络作为一个解码器，而中间层就是编码器输出得到的降维以后的数据了。这个过程就可以理解为是 Auto-encoder，中文称为自编码器。

本书会介绍 Auto-encoder 的基本原理，然后通过 pytorch 代码实战来深入学习和掌握自编码器。

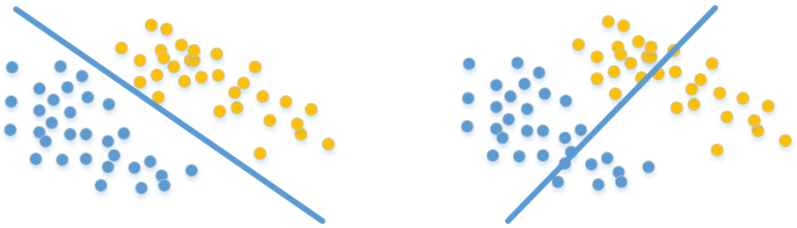
20211027：完成本书的第一版。

1. 自编码器简介

本章介绍自编码器的基本原理。本章的讲解内容参考自 *[4]*，没有对一些复杂的概念进行深入说明。本书的重点是实战，我们的实战内容并不会非常复杂。

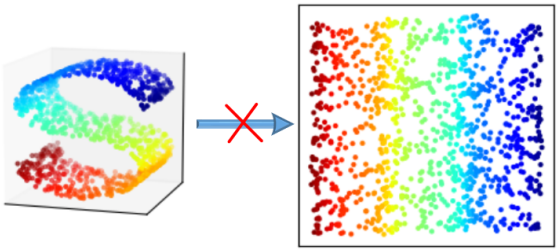
1.1 自编码器的结构

PCA 可以找到最大的映射维度，但 PCA 可能会有个问题，比如数据降维以后更难进行分类（如上图右）：

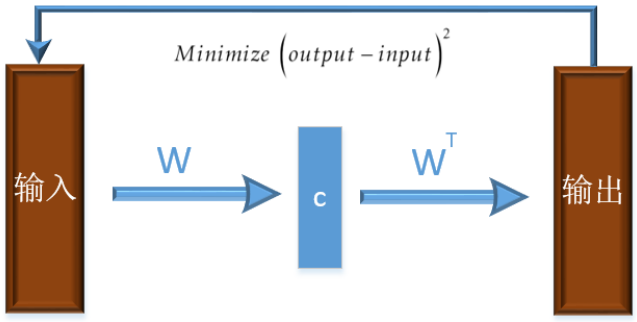


PCA　　LDA

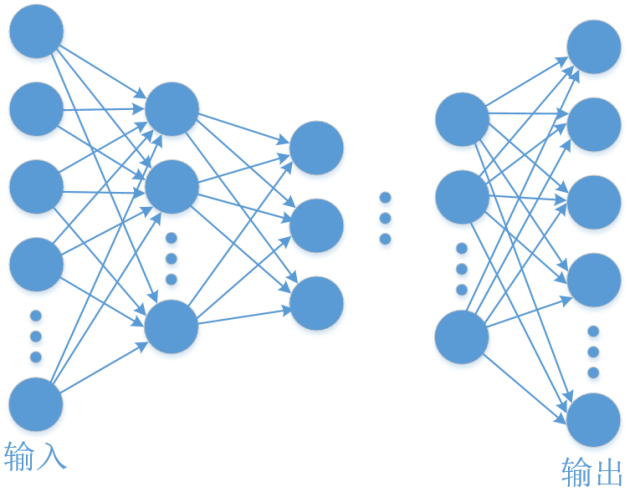
可以用 LDA 来根据标签进行映射，但它们只能处理线性情况，对于非线性的降维，例如下面的三维数据，最好能拉成一个面：



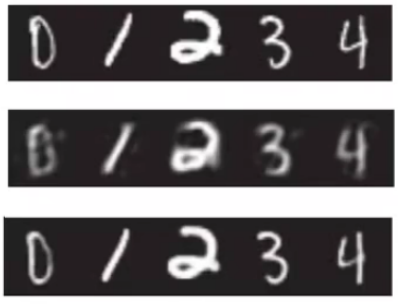
PCA 可以理解为是一个一层的网络：



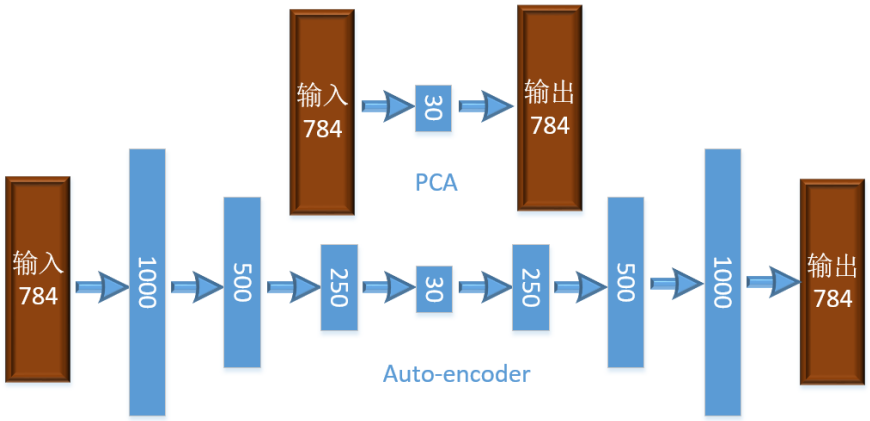
而一层只能处理线性映射，因此自然而然想到了多层结构：



最早的 Auto-encoder 一般认为来自于论文 [2]。在该论文中，取得了当时非常惊人的结果，在下图中，最上面是原始图像，中间是 PCA 降到 30 维再恢复以后的结果，最下面是用了多层 Auto-encoder 来降维到 30 维然后再恢复输出的结果，可以看到 Auto-encoder 的效果非常棒：



网络结构表示如下：



1.2 自编码器作为预训练网络权重

我们可以利用一大堆没有标签的 Auto-Encoder 来预训练一个网络，尤其是当有标签的分类数据本身比较少时，这种方法尤其有效。比如我们想训练一个如下的网络来进行五分类任务：

$$input: \quad 100 \tag{1.2.1}$$
$$first\ layer: \quad 216 \tag{1.2.2}$$
$$second\ layer: \quad 288 \tag{1.2.3}$$
$$third\ layer: \quad 120 \tag{1.2.4}$$
$$output: \quad 5 \tag{1.2.5}$$

我们首先可以训练第一层权重，让下面网络的输入和输出尽可能接近：

$$input: \quad 100 \tag{1.2.6}$$
$$first\ layer: \quad 216 \tag{1.2.7}$$
$$output: \quad 100 \tag{1.2.8}$$

然后固定第一层权重，训练下一层权重，令第一层和 output 尽可能接近：

$$input: \quad 100 \tag{1.2.9}$$
$$first\ layer\ (fixed): \quad 216 \tag{1.2.10}$$
$$second\ layer: \quad 288 \tag{1.2.11}$$
$$output: \quad 216 \tag{1.2.12}$$

然后固定第一和第二层权重，训练第三层权重，令第二层和 output 尽可能接近：

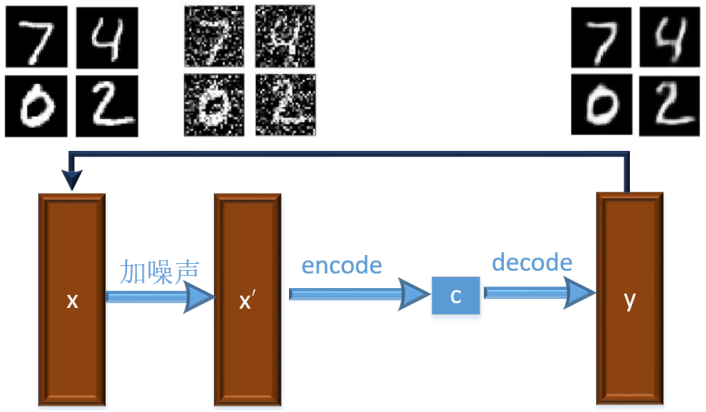
$$input: \quad 100 \tag{1.2.13}$$
$$first\ layer\ (fixed): \quad 216 \tag{1.2.14}$$
$$second\ layer\ (fixed): \quad 288 \tag{1.2.15}$$
$$third\ layer: \quad 120 \tag{1.2.16}$$
$$output: \quad 288 \tag{1.2.17}$$

这样就把中间层的权重都预训练好了。最终通过反向传播来 fine-tune（微调）网络权重即可。

1.3 自编码器用来去噪

首先提一句，我第一次用到自编码器就是为了做图像去噪，训练一个 Auto-encoder，就可以将蒙特卡洛光线追踪得到的噪声图进行有效地去噪。

去噪的网络如下 [3]：

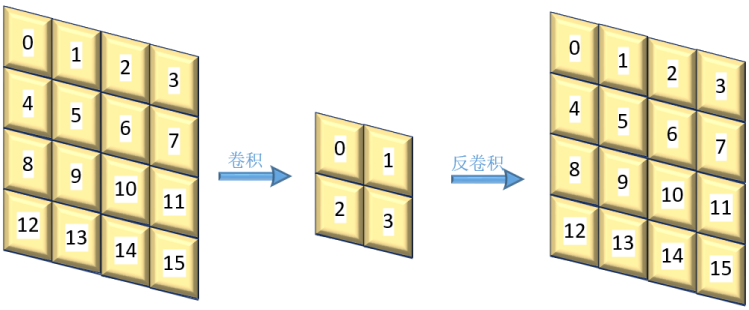


训练一个网络，能够让输出尽可能与无噪声输入相同，这样有噪图像输入以后，就能得到大致无噪声的原图。

1.4 卷积神经网络自编码器

对于卷积网络自编码器来说，编码器前面的几层是卷积和池化层，我们也会在解码器最后的几层设置逆卷积和反池化层（deconvolution 和 unpooling）。

不同的深度学习框架里，unpooling 的方法是不同的，这里不深入介绍，最简单的方法就是直接用重复值来填充扩大。deconvolution 其实就是 convolution，为了简单，我们以 4×4 像素的图为例，卷积核为 3×3 大小，不进行 padding，移动步长为 1，下图中的数字表示每个像素的编号，设第 i 个像素为 x_i。



卷积以后，得到 2×2 的图像，设第 i 个像素为 y_i。我们得到卷积的结果以 y_0 为例，设权重为 w。由于卷积层其实就是某些权重为 0 的全连接层，设前一层的第 i 个神经元连到后一层的第 j 个神经元的权重为 $w_{i,j}$

$$y_0 = w_{0,0}x_0 + w_{1,0}x_1 + w_{2,0}x_2 + w_{4,0}x_4 + w_{5,0}x_5 + w_{6,0}x_6 + w_{8,0}x_8 + w_{9,0}x_9 + w_{10,0}x_{10} \tag{1.4.1}$$

逆卷积则是一个逆过程，设反卷积结果为 z，权重为 t

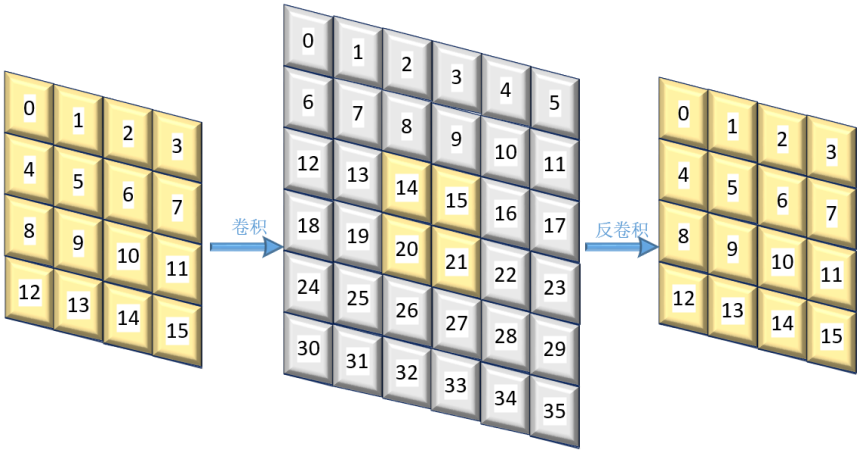
$$z_0 = t_{0,0}y_0 \tag{1.4.2}$$

$$z_{1,0} = t_{0,1}y_0 \quad z_{1,1} = t_{1,1}y_1 \Longrightarrow z_1 = z_{1,0} + z_{1,1} \tag{1.4.3}$$

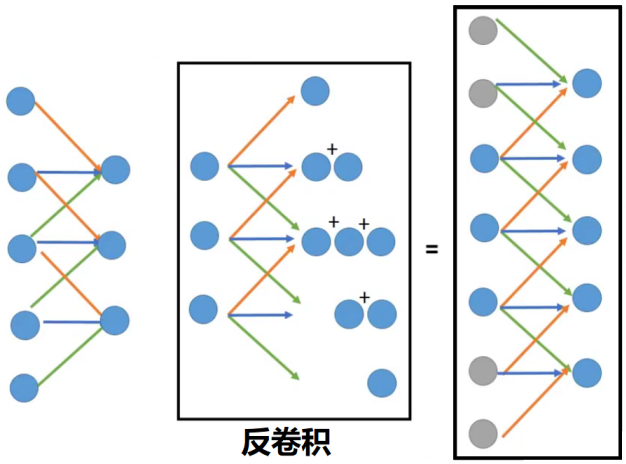
$$z_{2,0} = t_{0,2}y_0 \quad z_{2,1} = t_{1,2}y_1 \Longrightarrow z_2 = z_{2,0} + z_{2,1} \tag{1.4.4}$$

$$\ldots\ldots \tag{1.4.5}$$

如果在 y 周边补两圈 0，就可以看做是一个卷积层了：



如上式，因为逆卷积会对输出的多个像素值有影响，所以要把有影响的像素相加，得到最后的逆卷积像素值。借用 [4] 的一张图可以更好地说明情况：



1.5 其他训练方式

我们前面介绍的自编码器主要目的是为了降低重建误差。我们可以先训练出一个 encoder，然后将类别 A 输入得到的输出向量 A' 组合在一起，得到一个新的输入 AS'；类别 B 得到的输出向量 B' 组合在一起，得到 BB'；A 和 B' 组合在一起得到 AB'；B 和 A' 组合在一起得到 BA'。

我们设计一个二分类网络，当输入为 AA' 和 BB' 这种时，输出 1；输入为 AB' 和 BA' 时，输出 0。如果最后得到的分类正确率很高，说明这个自编码器得到的结果很有代表性——这个自编码器很好；否则这个自编码器就没有那么好。

这个思路来自论文 [5]，pytorch 源码可以从 [6] 中找到。

1.6 更具解释性的 encoder

我们希望得到的编码更具有代表性和可解释性，例如我们输入一段音频讲话的信号，编码以后，前 30 维是讲话的内容，后 30 维是说话者的内容（语气、声纹等）。或者干脆使用两个 encoder，一个编码讲话内容，另一个编码说话者的语气。

变声就可以这么来操作：将男生声音说的话编码后的后 30 维与女生声音编码后的前 30 维拼在一起，通入 decoder 以后就能得到同样的说话内容变声女生后的语音。

更极端点，可以将 encoder 输出的向量中最大值索引处设为 1，其他值设为 0，这样就得到了一个类似于分类的效果，这样就可以更好地理解聚类了，至于包含了 argmax 的函数怎么去微分，可以参考 [3]。

矢量量化变分自动编码器 VQVAE 是 NLP 中常用的自编码器 [8]，首先定义一个 codebook，里面有多个向量，我们用 encoder 输出一个向量以后，与 codebook 里面的向量比较，得到最相近的一个 v，然后 v 的解码就是我们最终的输出。这种方法可以比较容易的将输入数据进行归类，详细内容可以参考 [8]。

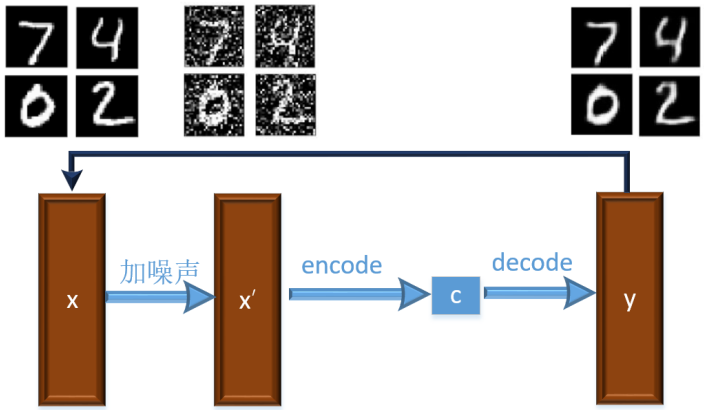
2. 代码实战——卷积 AutoEncoder 结构

本章介绍如何实现一个自编码器，并将此自编码器用于图像去噪。为了简单，我们这里使用手写数字辨识的 *MNIST* 数据集。本章最后一节会重点详细介绍 *Conv2d* 和 *ConvTranspose2d* 输入输出大小的计算原理。

2.1 我们的任务与数据

本节代码见 chapter2-1。

我们的结构类似于下图，我们首先用无噪声数据训练一个 Auto-encoder，然后测试输入有噪声数据以后得到的效果：



我们使用书写数字辨识数据集：

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
#下载数据集      28*28=784
```

```
training_data = datasets.MNIST(root='./dataset_method_1', train=True, transform=ToTensor(), download=True)
test_data = datasets.MNIST(root='./dataset_method_1', train=False, transform=ToTensor(), download=False)
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

我们可以用下面的代码测试一下能否正确显示：

```
import matplotlib.pyplot as plt
figure = plt.figure()
img, label = training_data[100]
plt.title(label)
#squeeze函数把为1的维度去掉
plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

2.2 网络结构

本节代码见 chapter2-2。

MNIST 数据是 28×28 大小的，而且是单通道黑白图，因此卷积层输入可以设为：

```
  nn.Conv2d(1, 32, 3, stride=1, padding=1)
```

这样卷积后输出为 [32,28,28]。当输入第一层卷积的 stride=3, padding=1 时，则卷积后输出为 [32,10,10]，大家可以在笔记本上动手画一画。

我们现在思考一下 ConvTranspose2d 逆卷积怎么操作。假如输入到逆卷积层的数据维度是 [8,2,2]，逆卷积层为：

```
nn.ConvTranspose2d(8, 16, 3, stride=2),
nn.ReLU(True),
nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1),
nn.ReLU(True),
nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),
```

我们看一下该函数的原型：

```
class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True, dilation=1)
```

outputpadding 表示输出边是否需要补充 0，一般不需要。根据官方文档，计算式如下：

$$output = (input - 1) \cdot stride + outputpadding - 2 \cdot padding + kernelsize \tag{2.2.1}$$

整个网络结构如下，注意最后一层用 Tanh 作为激活函数，这并不是说用 ReLU 不行，只是别人都在用，于是我也跟着一起用它了，其实实际测试中 Tanh 效果并不如 ReLU。

```
import torch.nn as nn
class AutoEncoder(nn.Module):
  def __init__(self):
    super(AutoEncoder, self).__init__()
    self.encoderConv = nn.Sequential(
      nn.Conv2d(1, 16, 3, stride=3, padding=1), # [16, 10, 10]
      nn.ReLU(True),
      nn.MaxPool2d(2), # [16, 5, 5]
      nn.Conv2d(16, 8, 3, stride=1, padding=1), # [8, 5, 5]
      nn.ReLU(True),
      nn.MaxPool2d(2), # [8, 2, 2]
    )
    self.decoderConv = nn.Sequential(
      nn.ConvTranspose2d(8, 16, 3, stride=2),  # [16, 5, 5]
      nn.ReLU(True),
      nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1),  # [8, 15,
          15]
      nn.ReLU(True),
      nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),  # [1, 28, 28]
      nn.ReLU(True)
    )
  def forward(self, x):
    x1 = self.encoderConv(x)
    x = self.decoderConv(x1)
    return x1, x
```

2.3 开始训练网络

本节代码见 chapter2-2。

首先定义一些网络的基本信息：

```
import torch
import torch.nn as nn
model = AutoEncoder().cuda()
```

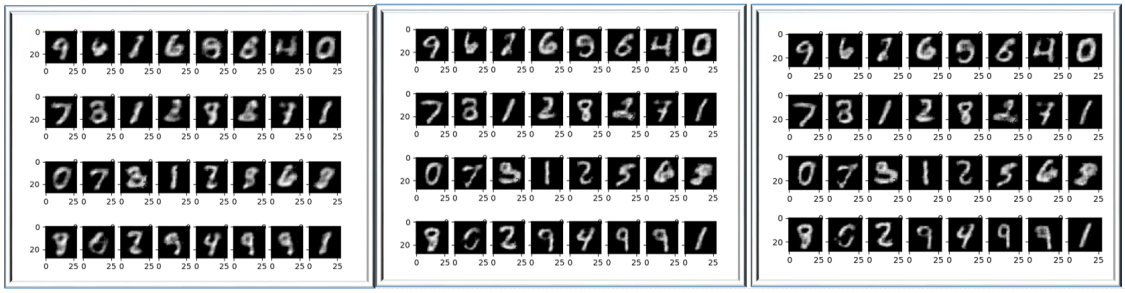
```
criterion = nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
    weight_decay=1e-5)
epochs_num = 100
import numpy as np
```

用来训练的程序如下。我们把训练集中随便找一组输入来当做测试可视化集，最终输出时只可视化一半的结果。

```
#用来抽取测试数据的标志位
testdataGot = 0
from torch.autograd import Variable
import matplotlib.pyplot as plt
for epoch in range(epochs_num):
    model.train()
    for i, data in enumerate(train_dataloader):
        #=====================forward=====================
        output1, output = model(data[0].cuda())
        loss = criterion(output, data[0].cuda())
        #=====================backward====================
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # 抽取第5个batch做测试
        if i == 5 & testdataGot==0:
            testdataGot = 1
            testdata = data[0].clone().cuda()
            print(testdata.shape)
    print('epoch [{}/{}], loss:{:.4f}'.format(epoch+1, epochs_num,
        loss.data))
    model.eval()
    if epoch % 10 == 0:
        with torch.no_grad():
            figure = plt.figure()
            outp2, outp = model(testdata)
            outp = outp.cpu()
            for i in range(8):
                for j in range(4):
                    adisplay = outp[i+j*8]
                    adisplay = adisplay.squeeze().detach().numpy()
```

```
                    plt.subplot(4, 8, i+j*8+1)
                    plt.imshow(adisplay, cmap="gray")
            plt.show()
```

可以看到，随着训练次数变多，生成的图像越来越清晰了：



2.4 去噪测试

写这一节感觉挺麻烦的。因为 MNIST 数据是直接读入为 tensor 的，因此加噪声还得转成 numpy 再进行操作。我也不想再建立自己的有噪数据集了，所以还是从已读入的 dataSet 入手吧。

首先写一个加噪声的程序：

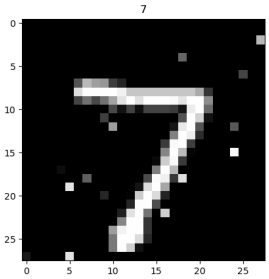
```
def add_noise(image,prob):
    noise_out = np.zeros(image.shape,np.float)
    thres = 1 - prob
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            rdn = random.random()
            if rdn < prob:
                noise_out[i][j] = random.random()
            elif rdn > thres:
                noise_out[i][j] = random.random()
            else:
                noise_out[i][j] = image[i][j]#其他情况像素点不变
    return noise_out #返回椒盐噪声和加噪图像
```

然后抽取一张图像并给其加入某种噪声：

```
# 构建一个用于加噪声然后测试网络去噪效果的图像
trans1 = ToTensor()
img, label = test_data[0]
img = img.squeeze().numpy()
print(img.shape)
img2 = sp_noise(img, 0.05)
import matplotlib.pyplot as plt
```

```
figure = plt.figure()
plt.title(label)
#squeeze函数把为1的维度去掉
plt.imshow(img2, cmap="gray")
plt.show()
img2 = trans1(img2)
img2 = img2.unsqueeze(dim=0).type(torch.FloatTensor).cuda()
```

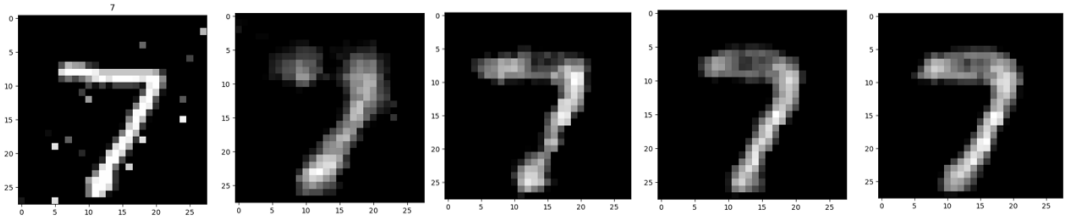
输出的有噪声图像显示如下：



在训练循环中就可以输入 img2 来作为测试：

```
outp2, outp = model(img2)
```

可以看到，训练 1 轮、10 轮、50 轮和 100 轮以以后，网络输入有噪图像得到的输出不断变好：



一个训练并且保存模型然后测试的代码可以参考 chapter2-4.py。

2.5 Conv2d 和 ConvTranspose2d 的计算原理

讲完实战部分，这一部分讲解图像通过 Conv2d 和 ConvTranspose2d 以后的长宽应该怎么计算。

Conv2d

函数原型：

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

输入和输出通道就不用说了。

groups 表示分组卷积，就是把输入输出分组，这里不再深入介绍。

dilation 表示扩张卷积，也叫空洞卷积，即卷积核之间会有空缺，这里也不再深入介绍，我们默认为 1，即卷积核之间间距为 1。

bias 为 True 表示输出中加入一个 bias，这个 bias 也可以来训练学习。

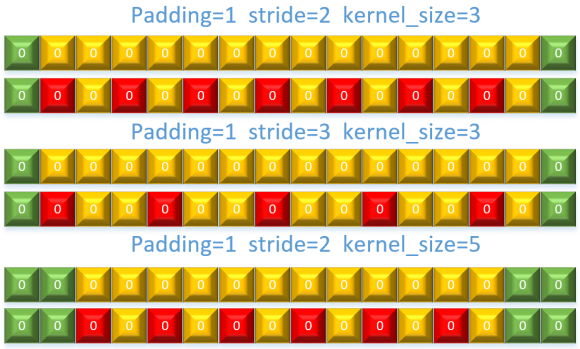
输出图像长宽取决于 kernel_size，stride 和 padding 这几个参数。举个特例，当 stride 为 1，padding 为 1，kernel_size 为 3 时，可以看出卷积后尺寸不变。令 dilation=0，根据官网的公式（我们会在后面重点讲解计算原理），得到：

$$W_{out} = \frac{W_{in} + 2 \times padding - (kernel\_size - 1) - 1}{stride} + 1 \tag{2.5.1}$$

这个式子第一个比较难理解的地方是 $(kernel\_size - 1)$ 这部分，需要注意的是卷积核如果变大，比如原图像是 100×100 的，padding=2 以后就变成了 104×104 大小的。如果 $kernel\_size = 3$，卷积核中心可以移动的范围就是 102×102 大小；如果 $kernel\_size = 5$，卷积核中心可以移动的范围就是 100×100 大小。当 stride=1 时，计算式就是：

$$W_{out}^{'} = W_{in} + 2 \times padding - (kernel\_size - 1) \tag{2.5.2}$$

当 stride 不为 1 时，我们需要判断卷积核中心在可以移动的范围内能移动多少次。当 stride=2，可以移动的范围是 100×100 时，stride 可以移动 50 次。当 stride=3，stride 可以移动的次数是 33 次。当 stride=2，可移动范围是 5×5 时，可移动次数为 3. 当 stride=2，可移动范围是 101×101 时，可移动范围是 51 次。当我们列举了这么多可能，我们就比较容易找到规律。下图中绿色表示 padding 后的大小，黄色表示可以移动的范围，红色表示可以移动的次数。



当 $W_{out}^{'}$ 恰好是 stride 的倍数时，卷积后图像大小就是 $\frac{W_{out}^{'}}{stride}$。当 $W_{out}^{'}$ 等于 $k \times stride + 1$ 时，卷积以后的图像大小就是 $k + 1$。因此，使用下式就可以得到计算方法：

$$\frac{W_{out}^{'} - 1}{stride} + 1 \tag{2.5.3}$$

把前面的 $W_{out}^{'}$ 代入进去，就能得到最初的计算式。

ConvTranspose2d

函数原型：

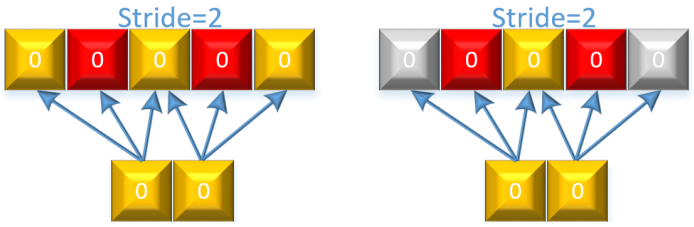
```
torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
    stride=1, padding=0, output_padding=0, groups=1, bias=True,
    dilation=1, padding_mode='zeros')
```

官网计算式为：

$$W_{out} = (W_{in} - 1) \cdot stride + outputpadding - 2 \cdot padding + kernel\_size \tag{2.5.4}$$

一般我们默认输出不再进行外扩，所以 outputpadding=0。

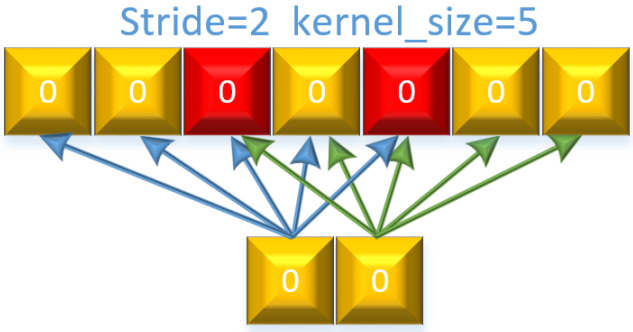
在前面，2×2 的图像输入到 kernel_size=3，stride=2 的逆卷积层后得到的结果是 5×5，如下图左；当其他条件不变，padding=1 时，就需要减去扩充的两边，如下图右。



当 padding=0 时，计算式就是：

$$W_{out} = (W_{in} - 1) \cdot stride + kernel\_size \tag{2.5.5}$$

我们可以根据逆过程来思考这个计算式。



当输入是 1×1 时，输出就是 $kernel\_size \times kernel\_size$ 大小。当输入是 2×2 时，输出在每一维度上就是 $kernel\_size + stride$ 大小。当输入是 $m \times m$ 时，输出在每一维度上就是 $kernel\_size + stride \times (m-1)$ 大小。

因此就得到了上述官网的计算式。

Bibliography

[1] https://www.astroml.org/book_figures/chapter7/fig_S_manifold_PCA.html

[2] Hinton, G. E , Salakhutdinov, et al. Reducing the Dimensionality of Data with Neural Networks.[J]. Science, 2006.

[3] Vincent P , Larochelle H , Bengio Y , et al. Extracting and Composing Robust Features with Denoising Autoencoders[C]// Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008. 2008.

[4] https://www.bilibili.com/video/BV1JE411g7XF?p=58

[5] Hjelm R D , Fedorov A , Lavoie-Marchildon S , et al. Learning deep representations by mutual information estimation and maximization[J]. 2018.

[6] https://github.com/DuaneNielsen/DeepInfomaxPytorch

[7] https://arxiv.org/abs/1611.01144

[8] https://arxiv.org/abs/1711.00937