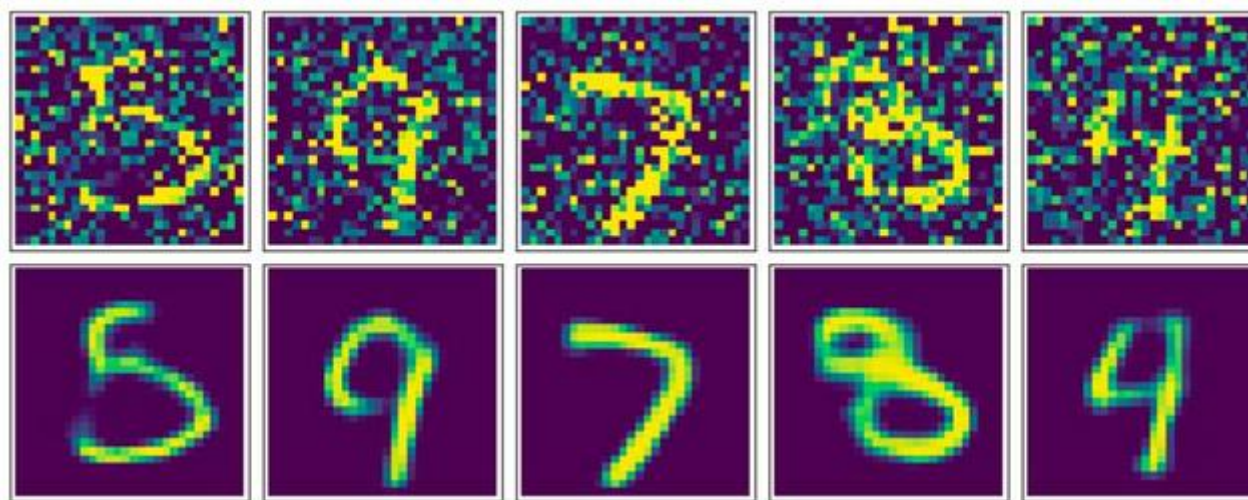
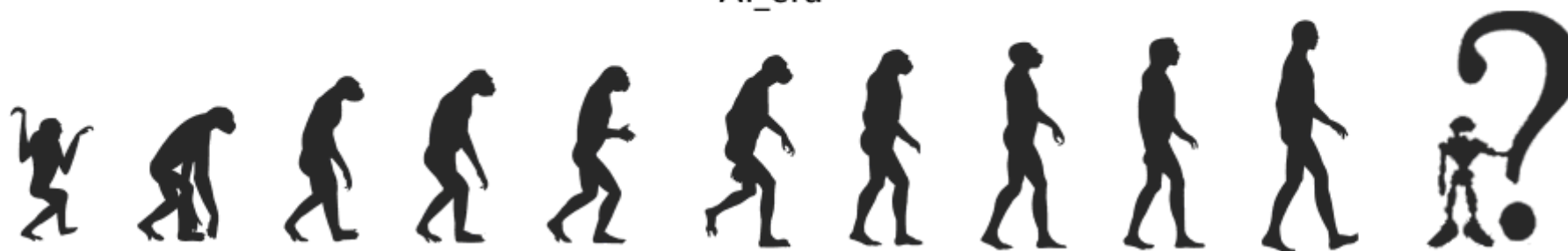


新智元 2017-07-17 17:04

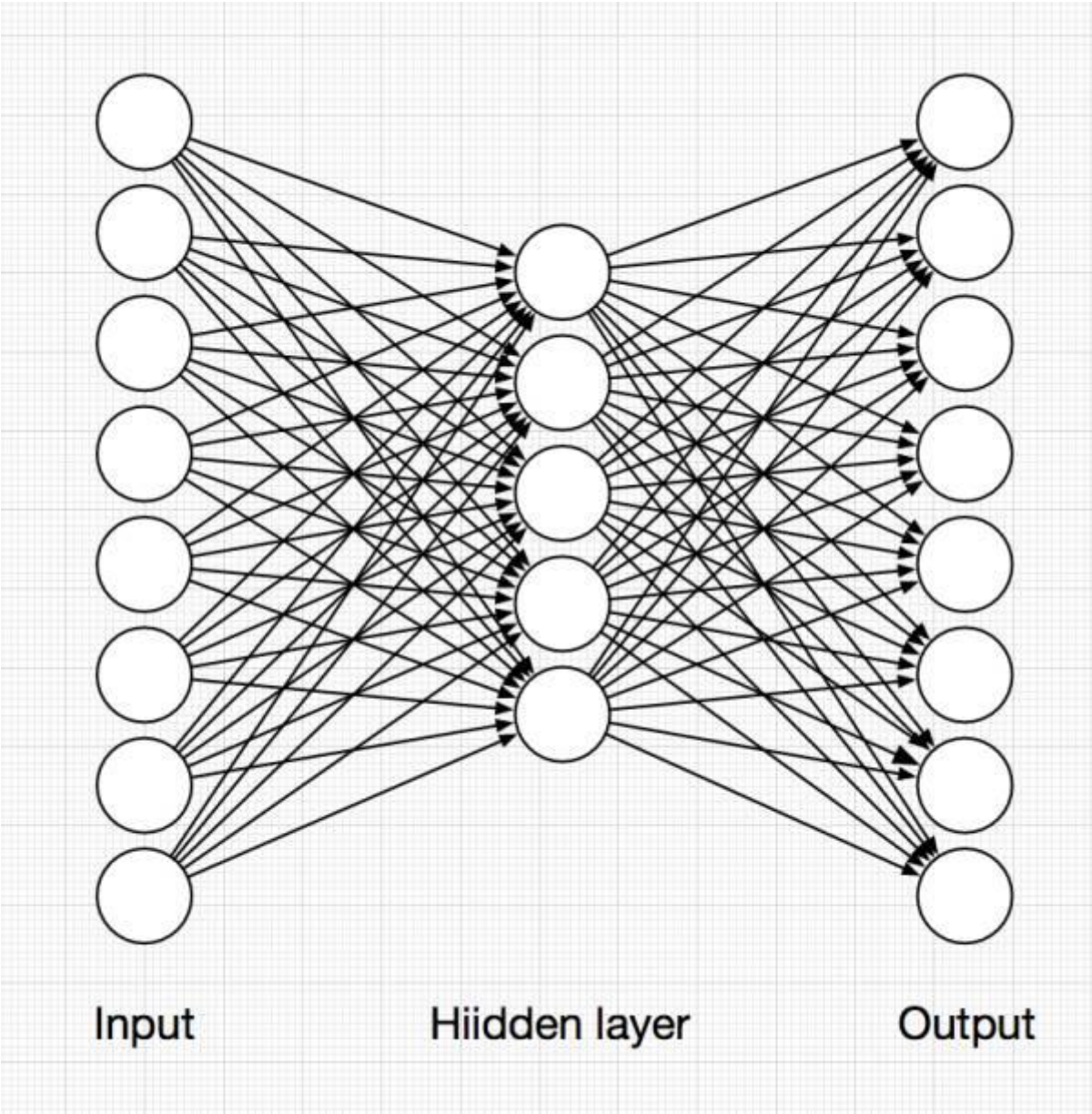


新智元
AI EQ

Al_era



这周工作太忙，本来想更把Attention tranlsation写出来，但一直抽不出时间，等后面有时间再来写。我们这周来看一个简单的自编码器实战代码，关于自编码器的理论介绍我就不详细介绍了，网上一搜一大把。最简单的自编码器就是通过一个encoder和decoder来对输入进行复现，例如我们将一个图片输入到一个网络中，自编码器的encoder对图片进行压缩，得到压缩后的信息，进而decoder再将这个信息进行解码从而复现原图。



作图工具：OmniGraffle

自编码器实际上是通过去最小化target和input的差别来进行优化，即让输出层尽可能地去复现原来的信息。由于自编码器的基础形式比较简单，对于它的一些变体也非常之多，包括DAE，SDAE，VAE等等，如果感兴趣的小伙伴可以去网上搜一下其他相关信息。

本篇文章将实现两个Demo，第一部分即实现一个简单的input-hidden-output结的自编码器，第二部分将在第一部分的基础上实现卷积自编码器来对图片进行降噪。

工具说明

- TensorFlow1.0
- jupyter notebook
- 数据：MNIST手写数据集
- 完整代码地址：NELSONZHAO/zhihu

第一部分

首先我们将实现一个如上图结构的最简单的AutoEncoder。

加载数据

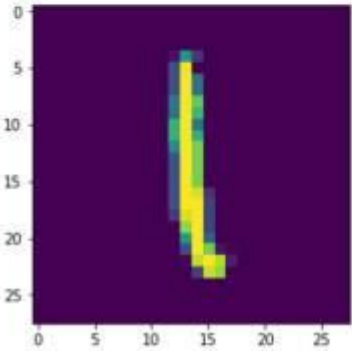
在这里，我们使用MNIST手写数据集来进行实验。首先我们需要导入数据，TensorFlow已经封装了这个实验数据集，所以我们使用起来也非常简单。

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', validation_size=0, one_hot=False)

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz

img = mnist.train.images[20]
plt.imshow(img.reshape((28,28)))

<matplotlib.image.AxesImage at 0x12d783780>
```



如果想让数据显示灰度图像，使用代码plt.imshow(img.reshape((28,28)), cmap='Greys_r')即可。

通过input_data就可以加载我们的数据集。如果小伙伴本地已经有了MNIST数据集（四个压缩包），可以把这四个压缩包放在目录MNIST_data下，这样TensorFlow就会直接Extract数据，而不用再重新下载。我们可以通过imshow来随便查看一个图像。由于我们加载进来的数据已经被处理成一个784维度的向量，因此重新显示的时候需要reshape一下。

构建模型

我们把数据加载进来以后就可以进行最简单的建模。在这之前，我们首先来获取一下input数据的大小，我们加载进来的图片是28x28的像素块，TensorFlow已经帮我们处理成了784维度的向量。同时我们还需要指定一下hidden layer的大小。

```
hidden_units = 64
input_units = mnist.train.images.shape[1]
```

在这里我指定了64，hidden_units越小，意味着信息损失的越多，小伙伴们也可以尝试一下其他的大小来看看结果。

AutoEncoder 中包含了 input，hidden 和 output 三层：

输入层

由于AutoEncoder是对源输入的复现，因此这里的输出层数据与输入层数据相同

inputs_ = tf.placeholder(tf.float32, (None, input_units), name='inputs_')
targets_ = tf.placeholder(tf.float32, (None, input_units), name='targets_')

隐层

hidden_ = tf.layers.dense(inputs_, hidden_units, activation=tf.nn.relu)

输出层

logits_ = tf.layers.dense(hidden_, input_units, activation=None)
outputs_ = tf.sigmoid(logits_, name='outputs_')

损失函数

loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=targets_, logits=logits_)
cost = tf.reduce_mean(loss)

优化函数

learning_rate = 0.01
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)

在隐层，我们采用了ReLU作为激活函数。

至此，一个简单的AutoEncoder就构造完成，接下来我们可以启动TensorFlow的graph来进行训练。


```
sess = tf.Session()
```

```
epochs = 20
batch_size = 128
sess.run(tf.global_variables_initializer())
for e in range(epochs):
    for idx in range(mnist.train.num_examples//batch_size):
        batch = mnist.train.next_batch(batch_size) # 获取下一个batch
        batch_cost, _ = sess.run([cost, optimizer],
                                feed_dict={inputs_: batch[0],
                                             targets_: batch[0]})

    print("Epoch: {}/{}".format(e+1, epochs),
          "Training loss: {:.4f}".format(batch_cost))
```

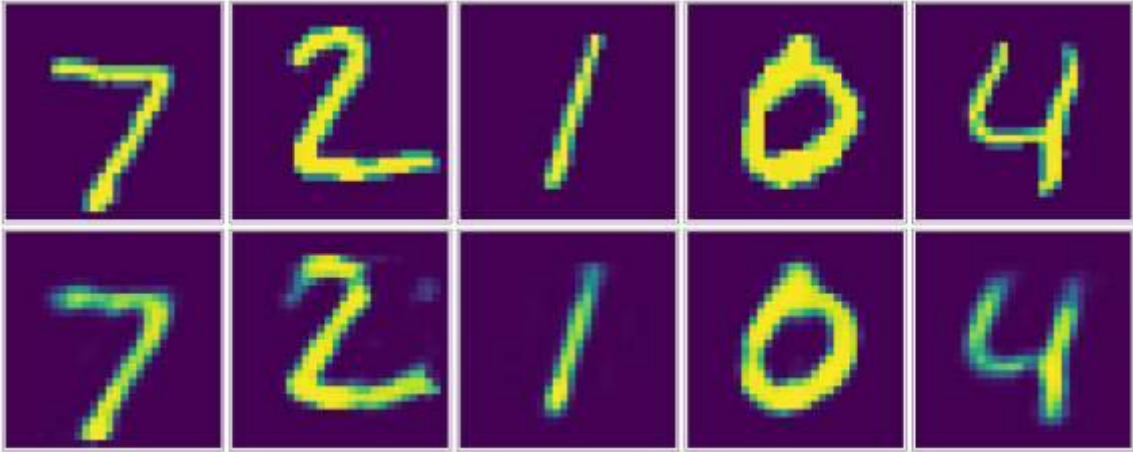
训练结果可视化

经过上面的步骤，我们构造了一个简单的AutoEncoder，下面我们将对结果进行可视化看一下它的表现。

```
# 绘图
fig, axes = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True, figsize=(20,8))
test_imgs = mnist.test.images[:5]
reconstructed, compressed = sess.run([outputs_, hidden_],
                                     feed_dict={inputs_: test_imgs})

for image, row in zip([test_imgs, reconstructed], axes):
    for img, ax in zip(image, row):
        ax.imshow(img.reshape((28, 28)))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

fig.tight_layout(pad=0.1)
```



这里，我挑选了测试数据集中的5个样本来进行可视化，同样的，如果想观察灰度图像，指定cmap参数为'Greys_r'即可。上面一行为test数据集中原始图片，第二行是经过AutoEncoder复现以后的图片，可以很明显的看到像素信息的损失。

同样，我们也可以把隐层压缩的数据拿出来可视化，结果如下：



这五张图分别对应了test中五张图片的隐层压缩后的图像。

通过上面一个简单的例子，我们了解了AutoEncoder的基本工作原理，下面我们将更进一步改进我们的模型，将隐层转换为卷积层来进行图像降噪。

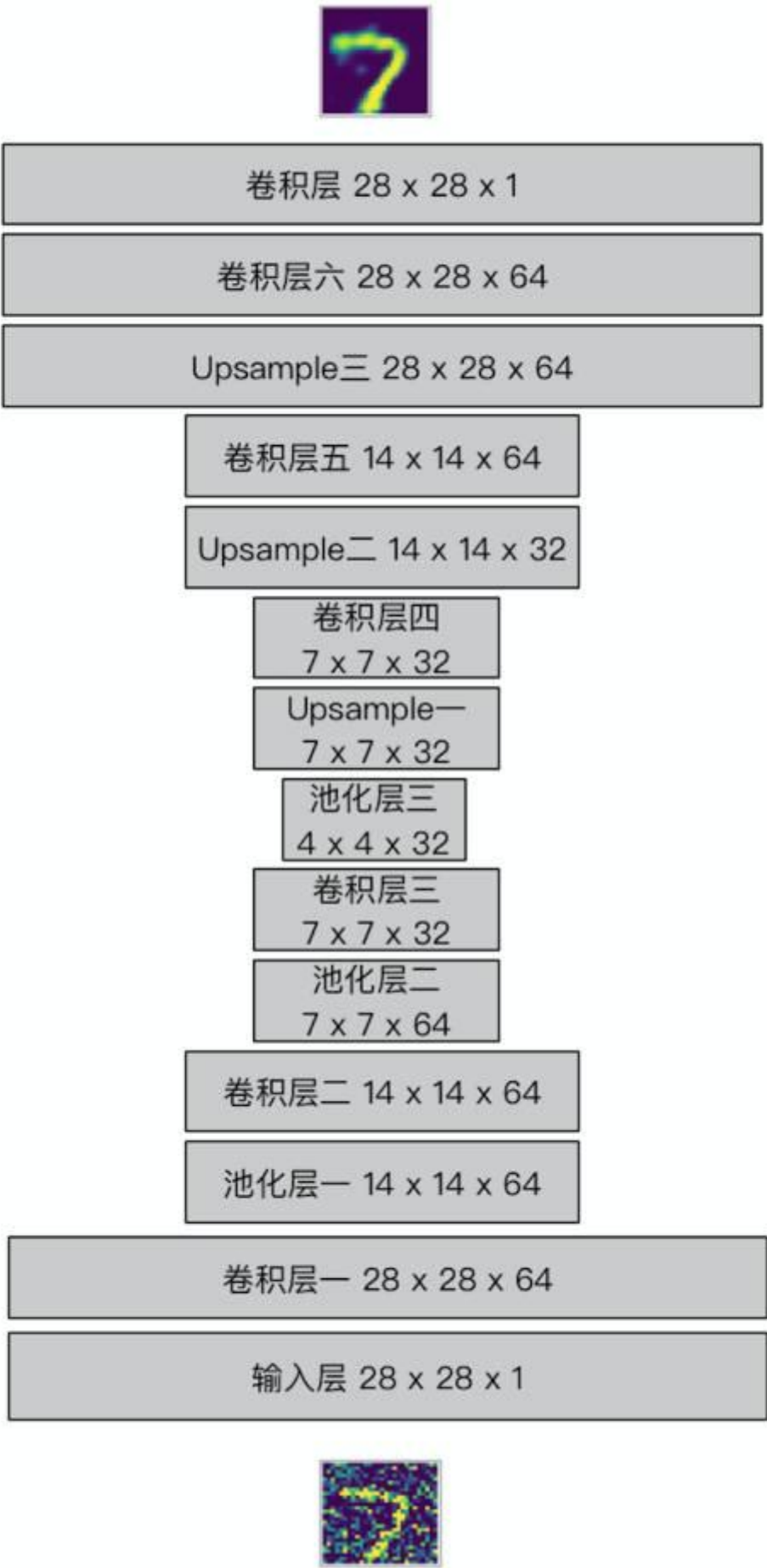
上面过程中省略了一部分代码，完整代码请去作者的GitHub上查看。

第二部分

在了解了上面AutoEncoder工作原理的基础上，我们在这一部分将对AutoEncoder加入多个卷积层来进行图片的降噪处理。

如果有小伙伴对卷积神经网络不清楚的话，可以去知乎看看大神们的文章来学习一下。

同样的我们还是使用MNIST数据集来进行实验，关于数据导入的步骤不再赘述，请下载代码查看。在开始之前，我们先通过一张图片来看一下我们的整个模型结构：



作图工具：OmniGraffle

我们通过向模型输入一个带有噪声的图片，在输出端给模型没有噪声的图片，让模型通过卷积自编码器去学习降噪的过程。

输入层

```
inputs_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='inputs_')
targets_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='targets_')
```

这里的输入层和我们上一部分的输入层已经不同，因为这里我们要使用卷积操作，因此，输入层应该是一个height x width x depth的一个图像，一般的图像depth是RGB格式三层，这里我们的MNIST数据集的depth只有1。

Encoder卷积层

Encoder卷积层设置了三层卷积加池化层，对图像进行处理。

```
conv1 = tf.layers.conv2d(inputs_, 64, (3,3), padding='same', activation=tf.nn.relu)
conv1 = tf.layers.max_pooling2d(conv1, (2,2), (2,2), padding='same')

conv2 = tf.layers.conv2d(conv1, 64, (3,3), padding='same', activation=tf.nn.relu)
conv2 = tf.layers.max_pooling2d(conv2, (2,2), (2,2), padding='same')

conv3 = tf.layers.conv2d(conv2, 32, (3,3), padding='same', activation=tf.nn.relu)
conv3 = tf.layers.max_pooling2d(conv3, (2,2), (2,2), padding='same')
```


第一层卷积中，我们使用了64个大小为3 x 3 的滤波器（ filter ），strides默认为1，padding设置为same后我们的height和width不会被改变，因此经过第一层卷积以后，我们得到的数据从最初的28 x 28 x 1 变为 28 x 28 x 64。

紧接着对卷积结果进行最大池化操作（ max pooling ），这里我设置了size和stride都是2 x 2，池化操作不改变卷积结果的深度，因此池化以后的大小为 14 x 14 x 64。

对于其他卷积层不再赘述。所有卷积层的激活函数都是用了ReLU。

经过三层的卷积和池化操作以后，我们得到的conv3实际上就相当于上一部分中AutoEncoder的隐层，这一层的数据已经被压缩为4 x 4 x 32的大小。

至此，我们就完成了Encoder端的卷积操作，数据维度从开始的28 x 28 x 1变成了4 x 4 x 32。

Decoder卷积层

接下来我们就要开始进行Decoder端的卷积。在这之前，可能有小伙伴要问了，既然Encoder中都已经把图片卷成了4 x 4 x 32，我们如果继续在Decoder进行卷积的话，那岂不是得到的数据size越来越小？所以，在Decoder端，我们并不是单纯进行卷积操作，而是使用了Upsample（中文翻译可以为上采样）+ 卷积的组合。

我们知道卷积操作是通过一个滤波器对图片中的每个patch进行扫描，进而对patch中的像素块加权求和后再进行非线性处理。举个例子，原图中我们的patch的大小假如是3 x 3（说的通俗点就是一张图片中我们取其中一个3 x 3大小的像素块出来），接着我们使用3 x 3的滤波器对这个patch进行处理，那么这个patch经过卷积以后就变成了1个像素块。在Deconvolution中（或者叫transposed convolution）这一过程是反过来的，1个像素块会被扩展成3 x 3的像素块。

但是Deconvolution有一些弊端，它会导致图片中出现checkerboard patterns，这是因为在Deconvolution的过程中，滤波器中会出现很多重叠。为了解决这个问题，有人提出了使用Upsample加卷积层来进行解决。

关于Upsample有两种常见的方式，一种是nearest neighbor interpolation，另一种是bilinear interpolation。

本文也会使用Upsample加卷积的方式来进行Decoder端的处理。

```
conv4 = tf.image.resize_nearest_neighbor(conv3, (7,7))
conv4 = tf.layers.conv2d(conv4, 32, (3,3), padding='same', activation=tf.nn.relu)

conv5 = tf.image.resize_nearest_neighbor(conv4, (14,14))
conv5 = tf.layers.conv2d(conv5, 64, (3,3), padding='same', activation=tf.nn.relu)

conv6 = tf.image.resize_nearest_neighbor(conv5, (28,28))
conv6 = tf.layers.conv2d(conv6, 64, (3,3), padding='same', activation=tf.nn.relu)
```

在TensorFlow中也封装了对Upsample的操作，我们使用resize_nearest_neighbor对Encoder卷积的结果resize，进而再进行卷积处理。经过三次Upsample的操作，我们得到了28 x 28 x 64的数据大小。最后，我们要将这个结果再进行一次卷积，处理成我们原始图像的大小。

```
logits_ = tf.layers.conv2d(conv6, 1, (3,3), padding='same', activation=None)
outputs_ = tf.nn.sigmoid(logits_, name='outputs_')
```

最后一步定义loss和optimizer。

```
loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=targets_, logits=logits_)
cost = tf.reduce_mean(loss)

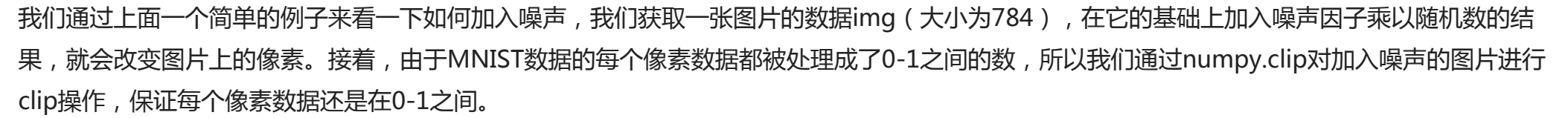
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)
```

loss函数我们使用了交叉熵进行计算，优化函数学习率为0.001。

构造噪声数据

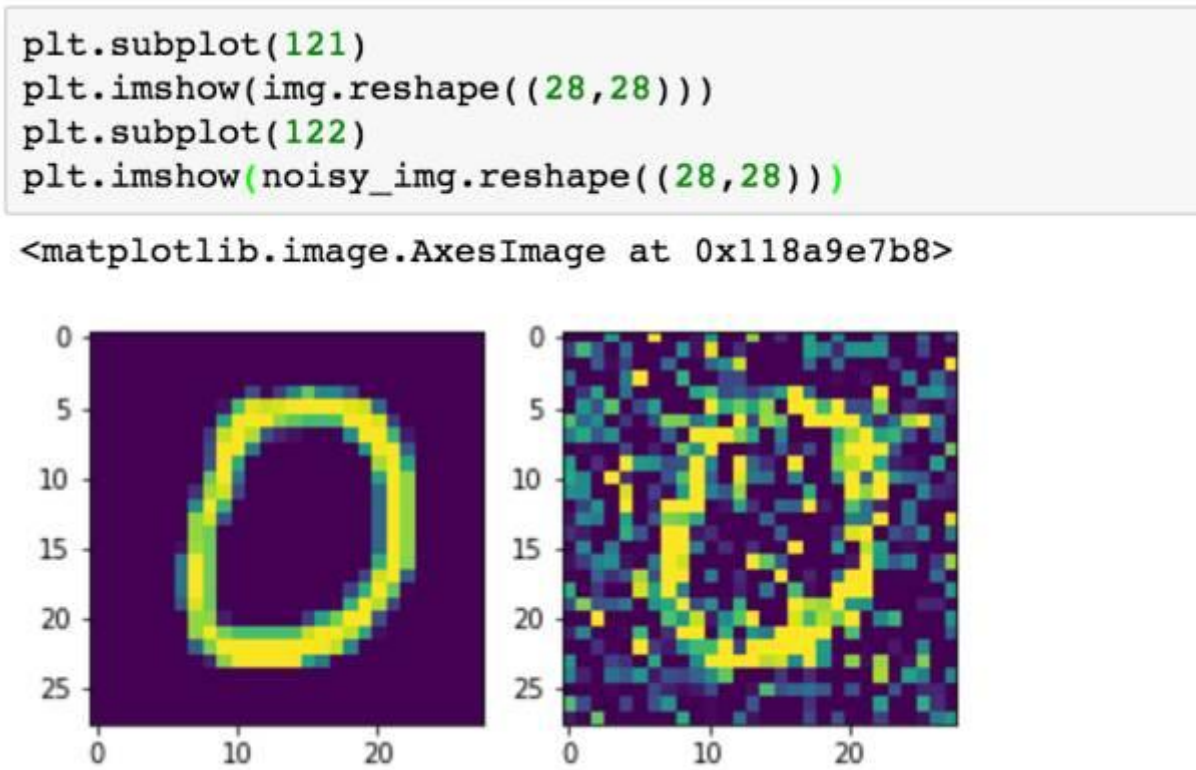
通过上面的步骤我们就构造完了整个卷积自编码器模型。由于我们想通过这个模型对图片进行降噪，因此在训练之前我们还需要在原始数据的基础上构造一下我们的噪声数据。

```
noise_factor = 0.5
img = mnist.test.images[10]
noisy_img = img + noise_factor * np.random.randn(*img.shape)
noisy_img = np.clip(noisy_img, 0.0, 1.0)
```

我们通过上面一个简单的例子来看一下如何加入噪声，我们获取一张图片的数据（大小为784），在它的基础上加入噪声因子乘以随机数的结果，就会改变图片上的像素。接着，由于MNIST数据的每个像素数据都被处理成了0-1之间的数，所以我们通过numpy.clip对加入噪声的图片进行clip操作，保证每个像素数据还是在0-1之间。

```
np.random.randn(*img.shape)的操作等于np.random.randn(img.shape[0], img.shape[1])
```

我们下来来看一下加入噪声前后的图像对比。



训练模型

介绍完模型构建和噪声处理，我们接下来就可以训练我们的模型了。

```
sess = tf.Session()

epochs = 10
batch_size = 128
sess.run(tf.global_variables_initializer())

for e in range(epochs):
    for idx in range(mnist.train.num_examples//batch_size):
        batch = mnist.train.next_batch(batch_size)
        imgs = batch[0].reshape((-1, 28, 28, 1))

        # 加入噪声
        noisy_imgs = imgs + noise_factor * np.random.randn(*imgs.shape)
        noisy_imgs = np.clip(noisy_imgs, 0., 1.)
        batch_cost, _ = sess.run([cost, optimizer],
                                feed_dict={inputs_: noisy_imgs,
                                             targets_: imgs})

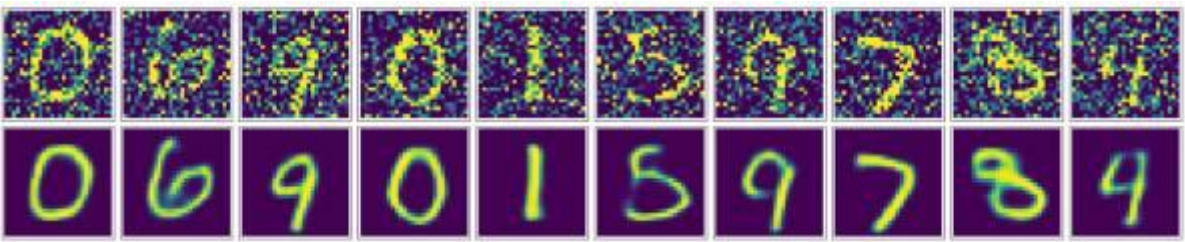
    print("Epoch: {}/{}".format(e+1, epochs),
          "Training loss: {:.4f}".format(batch_cost))
```

在训练模型时，我们的输入已经变成了加入噪声后的数据，而输出是我们的原始没有噪声的数据，主要要对原始数据进行reshape操作，变成与inputs_相同的格式。由于卷积操作的深度，所以模型训练时候有些慢，建议使用GPU跑。

```
记得最后关闭sess。
```

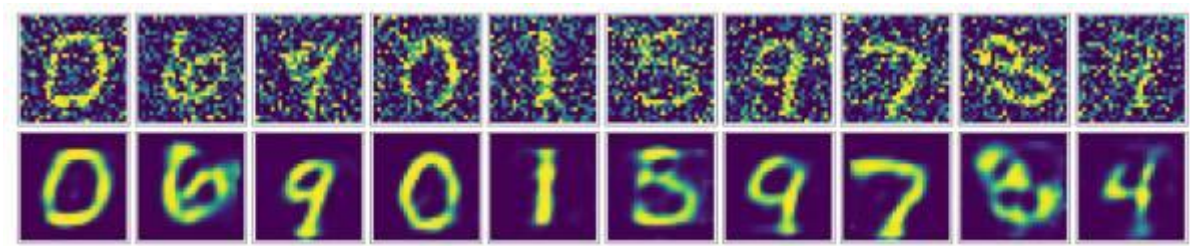
结果可视化

经过上面漫长的训练，我们的模型终于训练好了，接下来我们就通过可视化来看一看模型的效果如何。



可以看到通过卷积自编码器，我们的降噪效果还是非常好的，最终生成的图片看起来非常顺滑，噪声也几乎看不到了。

有些小伙伴可能就会想，我们也可以用基础版的input-hidden-output结构的AutoEncoder来实现降噪。因此我也实现了一版用最简单的input-hidden-output结构进行降噪训练的模型（ 代码在我的GitHub ）。我们来看看它的结果：



可以看出，跟卷积自编码器相比，它的降噪效果更差一些，在重塑的图像中还可以看到一些噪声的影子。

结尾

至此，我们完成了基础版本的AutoEncoder模型，还在此基础上加入卷积层来进行图片降噪。相信小伙伴对AntoEncoder也有了一个初步的认识。

完整的代码已经放在我的GitHub(NELSONZHAO/zhihu)上，其中包含了六个文件：

- BasicAE，基础版本的AutoEncoder（ 包含jupyter notebook和html两个文件 ）
- EasyDAE，基础版本的降噪AutoEncoder（ 包含jupyter notebook和html两个文件 ）
- ConvDAE，卷积降噪AutoEncoder（ 包含jupyter notebook和html两个文件 ）

如果觉得不错，可以给我的GitHub点个star就更好啦！