Tensorflow_GAN

<Excerpt in index | 首页摘要>
使用Tensorflow搭建GAN,熟悉一些常见的应用。
自动编码器　　GAM
Github：Tensorflow
Resource:深度学习理论与实战（基于TensorFlow实现）

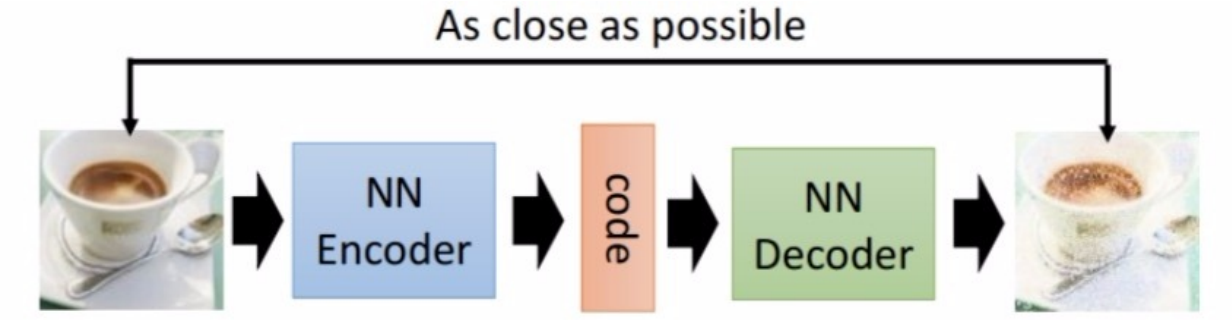
<The rest of contents | 余下全文>

自动编码器

1、编码器介绍

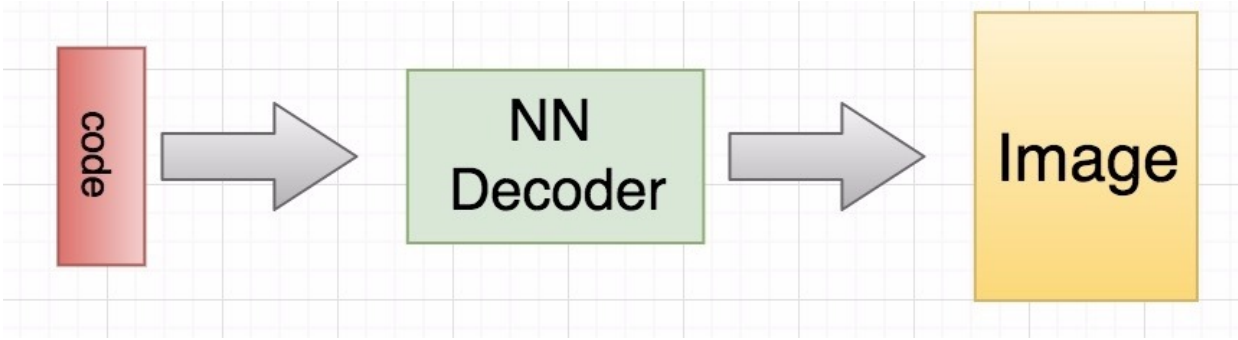
自动编码器最开始是作为一种数据压缩方法，同时还可以在卷积网络中进行逐层预训练，但是随后更多结构复杂的网络，比如 resnet 的出现使得我们能够训练任意深度的网络，自动编码器就不再使用在这个方面，下面我们讲一讲自动编码器的一个新的应用，这是随着**生成对抗模型**而出现的，就是使用自动编码器生成数据。

自动编码器的一般结构如下



由上面的图片，我们能够看到，**第一部分是编码器(encoder)**，**第二部分是解码器(decoder)**，编码器和解码器都可以是任意的模型，通常我们可以使用神经网络作为我们的编码器和解码器，输入的数据经过神经网络降维到一个编码，然后又通过另外一个神经网络解码得到一个与原始数据一模一样的生成数据，通过比较原始数据和生成数据，希望他们尽可能接近，所以最小化他们之间的差异来训练网络中编码器和解码器的参数。

当训练完成之后，我们如何生成数据呢？非常简单，我们只需要拿出解码器的部分，然后随机传入 code，就可以通过解码器生成各种各样的数据



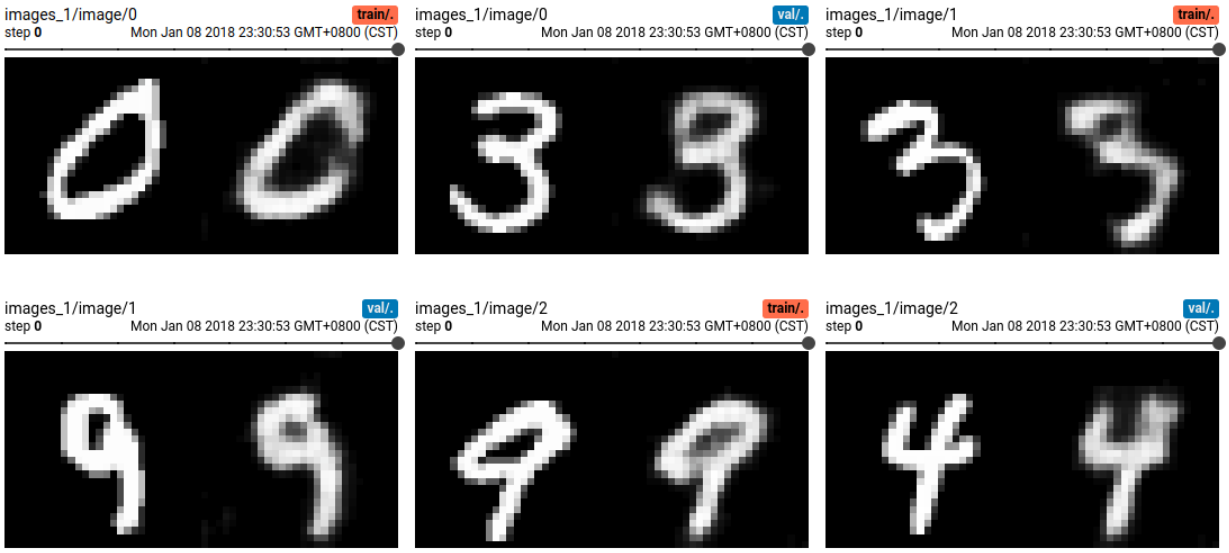
2、简构建自动编码器

使用卷积神经网络搭建编码器

```
def conv_autoencoder(inputs, scope='conv_autoencoder', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        with slim.arg_scope([slim.conv2d, slim.conv2d_transpose], activat
ion_fn=tf.nn.relu, padding='SAME'):
            with tf.variable_scope('encoder'):
                encode = slim.conv2d(inputs, 16, 3, stride=3, scope='conv
1') # (b, 10, 10, 16)
                encode = slim.max_pool2d(encode, 2, stride=2, padding='SA
ME', scope='pool1') # (b, 5, 5, 16)
                encode = slim.conv2d(encode, 8, 3, stride=2, scope='conv
2') # (b, 3, 3, 8)
                encode = slim.max_pool2d(encode, 2, stride=2, padding='SA
ME', scope='pool2') # (b, 2, 2, 8)
            with tf.variable_scope('decoder'):
                decode = slim.conv2d_transpose(encode, 16, 3, stride=2, p
adding='VALID', scope='trans_conv1') # (b, 5, 5, 16)
                decode = slim.conv2d_transpose(decode, 8, 5, stride=3, sc
ope='trans_conv2') # (b, 15, 15, 8)
                decode = slim.conv2d_transpose(decode, 1, 3, stride=2, ac
tivation_fn=None, scope='trans_conv3') # (b, 30, 30, 1)
                decode = tf.image.resize_bilinear(decode, (28, 28)) # (b,
 28, 28, 1) 这里由于转置卷积的不可逆，采用线性插值回到原图大小
                decode = tf.tanh(decode)

            return encode, decode
```

结果



变分自动编码器

1、Introduction

变分编码器是自动编码器的升级版本，其结构跟自动编码器是类似的，也由编码器和解码器构成。

自动编码器有个问题，就是并**不能任意生成图片**，因为我们没有办法自己去**构造隐藏向量**，需要通过一张图片输入编码我们才知道得到的隐含向量是什么，这时我们就可以通过变分自动编码器来解决这个问题。

其实原理特别简单，只需要在编码过程给它增加一些限制，迫使其**生成的隐含向量能够粗略的遵循一个标准正态分布，这就是其与一般的自动编码器最大的不同。**

这样我们生成一张新图片就很简单了，我们只需要给它一个标准正态分布的随机隐含向量，这样通过解码器就能够生成我们想要的图片，而不需要给它一张原始图片先编码。

一般来讲，我们通过 encoder 得到的隐含向量并不是一个标准的正态分布，为了衡量两种分布的相似程度，我们使用 KL divergence，利用其来**表示隐含向量与标准正态分布之间差异的 loss**，另外一个 loss 仍然使用生成图片与原图片的均方误差来表示。

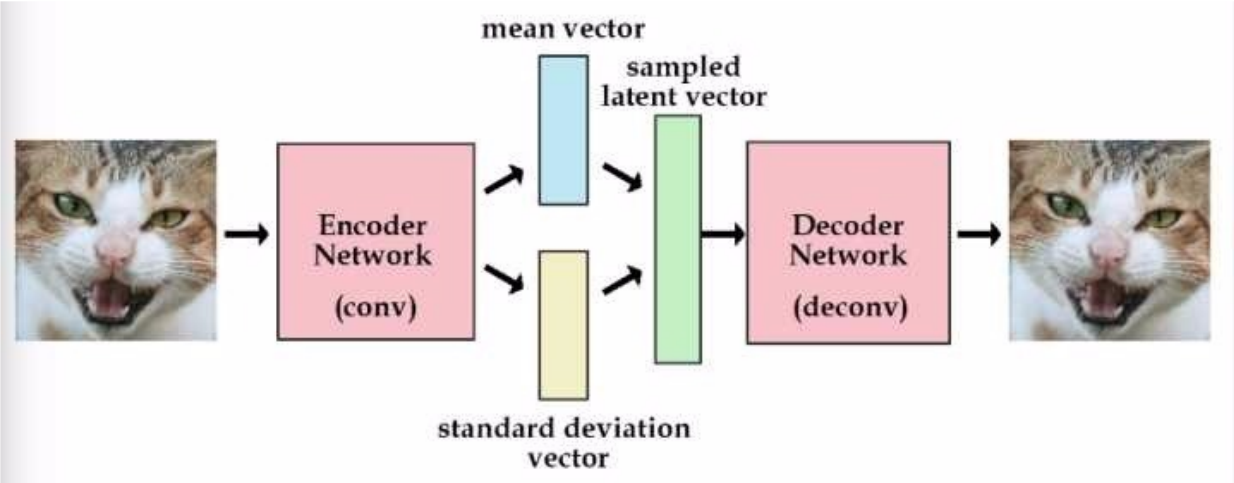
KL divergence 的公式如下

$$DKL(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

重参数

为了避免计算 KL divergence 中的积分，我们使用重参数的技巧，不是每次产生一个隐含向量，而是生成两个向量，**一个表示均值，一个表示标准差**，这里我们默认编码之后的隐含向量服从一个正态分布的之后，就可以用**一个标准正态分布先乘上标准差再加上均值来合成这个正态分布**，最后 loss 就是希望这个生成的正态分布能够符合一个标准正态分布，也就是希望均值为 0，方差为 1

所以标准的变分自动编码器如下



2、Variational Auto-Encoder(VAE)

loss

由均方误差和 **KL divergence** 求和得到一个总的 **loss**：

```
def loss_fun(recon_x, x, mean, std, eps=1e-8):
    """
    recon_x: generating images
    x: original images
    mean: latent mean
    var: latent var
    """
    mse = tf.reduce_sum(tf.square(x - recon_x))
    # 0.5 * sum(mu^2 + std^2 - 2log(std) - 1)
    kld_element = tf.square(mean) + tf.square(std) - 2.0 * tf.log(std + e
ps) - 1
    kld = 0.5 * tf.reduce_sum(kld_element)

    return mse + kld
```

简单构建一个变分自动编码器

```
def vae(inputs, scope='vae', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        with slim.arg_scope([slim.fully_connected], activation_fn = tf.n
n.relu):
            # 编码
            with tf.variable_scope('encoder'):
                encode = slim.fully_connected(inputs, 400, scope='fc1')
                mean = slim.fully_connected(encode, 20, activation_fn=Non
e, scope='fc2_mean')
                logvar = slim.fully_connected(encode, 20, activation_fn=N
one, scope='fc2_var')
```

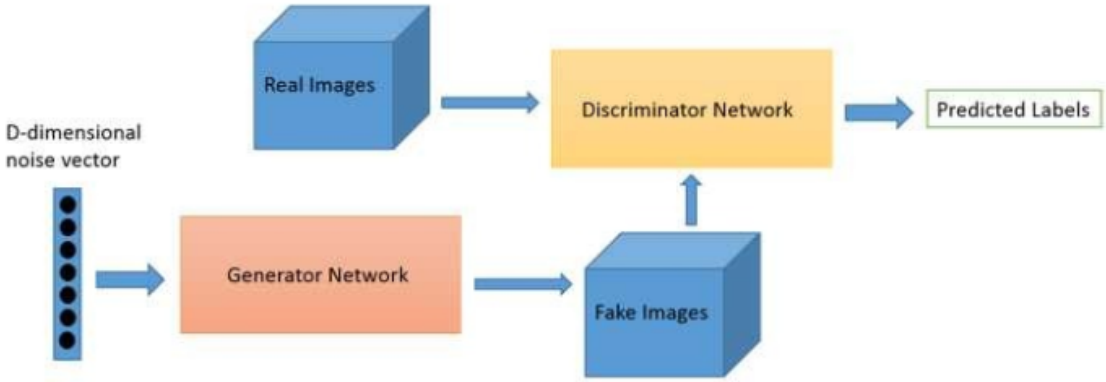
```
            # 重新参数化成正态分布
            with tf.variable_scope('reparametrize'):
                std = tf.sqrt(tf.exp(logvar))
                eps = tf.random_normal([20,], name='epsilon')
                rep = eps * std + mean

            # 解码
            with tf.variable_scope('decoder'):
                decode = slim.fully_connected(rep, 400, scope='fc3')
                decode = slim.fully_connected(decode, 784, activation_fn=
tf.tanh, scope='fc4')

            return decode, mean, std
```

GANs（生成对抗网络）

根据这个名字就可以知道这个网络是由两部分组成的，**第一部分是生成，第二部分是对抗。**简单来说，就是有一个生成网络和一个判别网络，通过训练让两个网络相互竞争，生成网络来生成假的数据，对抗网络通过判别器去判别真伪，最后希望生成器生成的数据能够以假乱真。



对抗过程简单来说就是一个判断真假的判别器，相当于一个二分类问题，我们输入一张真的图片希望判别器输出的结果是1，输入一张假的图片希望判别器输出的结果是0。**这其实已经和原图片的 label 没有关系了，不管原图片到底是一个多少类别的图片，他们都统一称为真的图片**，label 是 1 表示真实的；而生成的假的图片的 label 是 0 表示假的。

Generator Network

生成网络如何生成**一张假的图片。**首先给出一个简单的高维的正态分布的噪声向量，如上图所示的 D-dimensional noise vector，这个时候我们可以通过仿射变换，也就是 xw+b 将其映射到一个**更高的维度**，然后将他重新排列成一个矩形，这样看着更像一张图片，接着进行一些**卷积、转置卷积、池化、激活函数等进行处理**，最后得到了一个与我们输入图片大小一模一样的噪音矩阵，这就是我们所说的假的图片。

这个时候我们如何去**训练这个生成器呢？**这就需要通过对抗学习，**增大判别器判别这个结果为真的概率**，通过这个步骤不断调整生成器的参数，希望生成的图片越来越像真的，而在这一步中我

们不会更新判别器的参数，因为如果判别器不断被优化，可能生成器无论生成什么样的图片都无法骗过判别器。

基于mnist简单举例

判别网络

判别网络的结构非常简单，就是一个**二分类器**，结构如下:

- 全连接(784 -> 256)
- leakyrelu, α 是 0.2
- 全连接(256 -> 256)
- leakyrelu, α 是 0.2
- 全连接(256 -> 1)

其中 leakyrelu 是指 f(x) = max(α x, x)

```
def discriminator(inputs, scope='discriminator', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        with slim.arg_scope([slim.fully_connected], activation_fn=None):
            net = slim.fully_connected(inputs, 256, scope='fc1')
            net = tf.nn.leaky_relu(net, alpha=0.2, name='act1')
            net = slim.fully_connected(net, 256, scope='fc2')
            net = tf.nn.leaky_relu(net, alpha=0.2, name='act2')
            net = slim.fully_connected(net, 1, scope='fc3')

            return net
```

生成网络

接下来我们看看生成网络，生成网络的结构也很简单，就是根据一个**随机噪声生成一个和数据维度一样的张量**，结构如下:

- 全连接(噪音维度 -> 1024)
- relu
- 全连接(1024 -> 1024)
- relu
- 全连接(1024 -> 784)
- tanh 将数据裁剪到 -1 ~ 1 之间

```
def generator(noise, scope='generator', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        with slim.arg_scope([slim.fully_connected], activation_fn=tf.nn.r
elu):
            net = slim.fully_connected(noise, 1024, scope='fc1')
            net = slim.fully_connected(net, 1024, scope='fc2')
            net = slim.fully_connected(net, 784, activation_fn=tf.tanh, s
cope='fc3')

            return net
```

Loss

接下来我们需要定义生成**对抗网络的 loss**，通过前面的讲解我们知道，对于对抗网络，相当于二分类问题，将真的判别为真的，假的判别为假的，作为辅助，可以参考一下论文中公式

$$\ell_D = \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

而对于生成网络，需要去骗过对抗网络，也就是将假的也判断为真的，作为辅助，可以参考一下论文中公式

$$\ell_G = \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

如果你还记得前面的二分类 loss，那么你就会发现上面这两个公式就是二分类 loss

$$bce(s, y) = y * \log(s) + (1 - y) * \log(1 - s)$$

```
def discriminator_loss(logits_real, logits_fake, scope='D_loss'): # 判别网
络的`loss`
    with tf.variable_scope(scope):
        # 在`tensorflow`中我们使用`log_loss`, 并且需要将判别网络的输出
        # 用`sigmoid`函数转化为概率
        loss = tf.losses.log_loss(true_labels, tf.sigmoid(logits_real)) +
 tf.losses.log_loss(fake_labels, tf.sigmoid(logits_fake))
        return loss

def generator_loss(logits_fake, scope='G_loss'): # 生成网络的`loss`
    with tf.variable_scope(scope):
        loss = tf.losses.log_loss(true_labels, tf.sigmoid(logits_fake))
    return loss
```