# 自编码器（Autoencoder）

整理于 2018.11.21 by nana-li

**写在前面：**

因为时间原因本文有些图片自己没有画，来自网络的图片我尽量注出原链接，但是有的链接已经记不得了，如果有使用到您的图片，请联系我，必注释。

**本文展示的所有完整代码详见：**https://github.com/Nana0606/Autoencoder/tree/master/Autoencoder（目前只有 Keras 版本，有时间会写 Tensorflow 版本）

**自编码器主要涉及以下内容：**

① 自编码器（Autoencoder）
② 栈式自编码器（Stack Autoencoder）
③ 欠完备自编码器（Undercomplete Autoencoder）
④ 稀疏自编码器（Sparse Autoencoder）
⑤ 去噪自编码器（Denoising Autoencoder）
⑥ 收缩自编码器（Contractive Autoencoder）
⑦ 变分自编码器（Variational Autoencoder）
⑧ CNN 自编码器
⑨ RNN 自编码器
⑩ 随机编码器和解码器
⑪ 表示能力、层的大小和深度
⑫ 自编码器应用

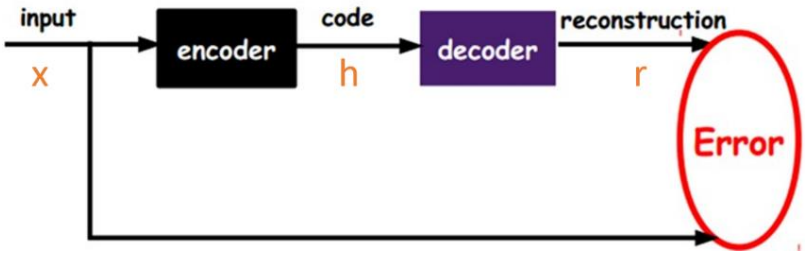
本文暂时主要基于普通自编码器、栈式自编码器、欠完备自编码器、稀疏自编码器和去噪自编码器（①-⑤的内容），会提供理论+实践（有的理论本人没有完全理解，就先没有写上，后更）。另外，关于收缩自编码器、变分自编码器、CNN 自编码器、RNN 自编码器及其自编码器的应用，后更。

## 一、自编码器（Autoencoder, AE）

### 1.1、 自编码器结构和思想

**自编码器是一种无监督的数据维度压缩和数据特征表达方法。**

自编码器是神经网络的一种，经过训练后能尝试将输入复制到输出。自编码器由编码器和解码器组成，如下图所示：




自编码器结构

（图形来源：https://www.cnblogs.com/caocan702/p/5665972.html）

$h = f(x)$表示编码器，$r = g(h) = g(f(x))$表示解码器，自编码器的目标便是优化损失函数$L(x, g(f(x)))$，也就是减小图中的 Error。

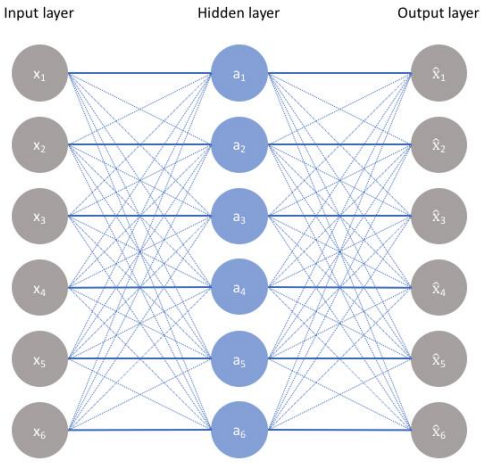
## 1.2、 自编码器和前馈神经网络的比较

（1）自编码器是前馈神经网络的一种，最开始主要用于数据的降维以及特征的抽取，随着技术的不断发展，现在也被用于生成模型中，可用来生成图片等。
（2）前馈神经网络是有监督学习，其需要大量的有标注数据。自编码器是无监督学习，数据不需要标注因此较容易收集。
（3）前馈神经网络在训练时主要关注的是输出层的数据以及错误率，而自编码的应用可能更多的关注中间隐层的结果。

## 1.3、 普通自编码器存在的问题

在普通的自编码器中，输入和输出是完全相同的，因此输出对我们来说没有什么应用价值，所以我们希望利用中间隐层的结果，比如，可以将其作为特征提取的结果、利用中间隐层获取最有用的特性等。

但是如果只使用普通的自编码器会面临什么问题呢？比如，输入层和输出层的维度都是 5，中间隐层的维度也是 5，那么我们使用相同的输入和输出来不断优化隐层参数，最终得到的参数可能是这样：$x_1 \to a_1, x_2 \to a_2, \ldots$的参数为 1，其余参数为 0，也就是说，**中间隐层的参数只是完全将输入记忆下来，并在输出时将其记忆的内容完全输出即可，神经网络在做恒等映射，产生数据过拟合**。如下图所示：




自编码器的完全记忆情况

（图片来源：https://www.jeremyjordan.me/autoencoders/）

上图是隐层单元数等于输入维度的情况，当然，如果是隐层单元数大于输入维度，也会发生类似的情况。**即，当隐层单元数大于等于输入维度时，网络可以采用完全记忆的方式，虽然这种方式在训练时精度很高，但是复制的输出对我们来说毫无意义。**

因此，我们会给隐层加一些约束，如限制隐藏单元数、添加正则化等，后面后介绍。

## 1.4、 普通自编码器实现与结果分析

**（1）实现框架：**Keras

**（2）数据集：**Mnist 手写数字识别

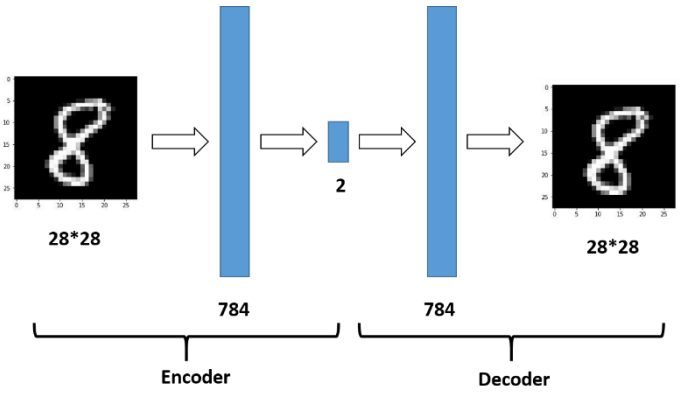
**（3）关键代码：**

```
def train(x_train):
    """
    build autoencoder.
    :param x_train:    the train data
    :return: encoder and decoder
    """
    # input placeholder
    input_image = Input(shape=(ENCODING_DIM_INPUT, ))
    # encoding layer
    hidden_layer = Dense(ENCODING_DIM_OUTPUT, activation='relu')(input_image)
    # decoding layer
    decode_output = Dense(ENCODING_DIM_INPUT, activation='relu')(hidden_layer)
    # build autoencoder, encoder, decoder
```

```
    autoencoder = Model(inputs=input_image, outputs=decode_output)
    encoder = Model(inputs=input_image, outputs=hidden_layer)
    # compile autoencoder
    autoencoder.compile(optimizer='adam', loss='mse')
    # training
    autoencoder.fit(x_train, x_train, epochs=EPOCHS, batch_size=BATCH_SIZE,
shuffle=True)
    return encoder, autoencoder
```

**（4）代码分析：**

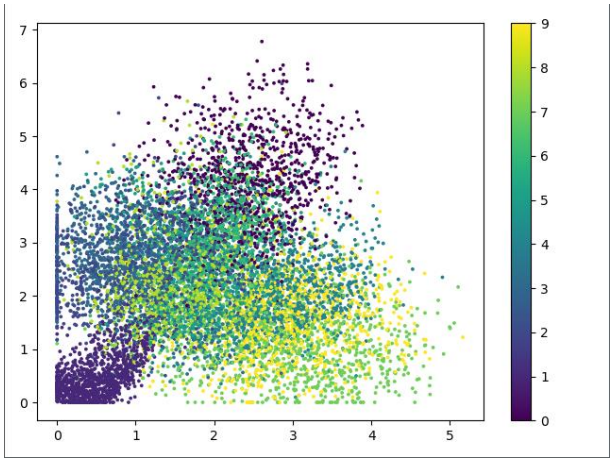
Keras 封装的比较厉害，所以傻瓜式编程，这里是最简单的自编码器，其输入维度是 28*28=784，中间单隐层的维度是 2，使用的激活函数是 Relu，返回 encoder 和 autoencoder。encoder 部分可以用于降维后的可视化，或者降维之后接分类等，autoencoder 可以用来生成图片等（这部分代码 git 上都有）。结构见图如下：




自编码器代码结构图

**（5）结果展示：**

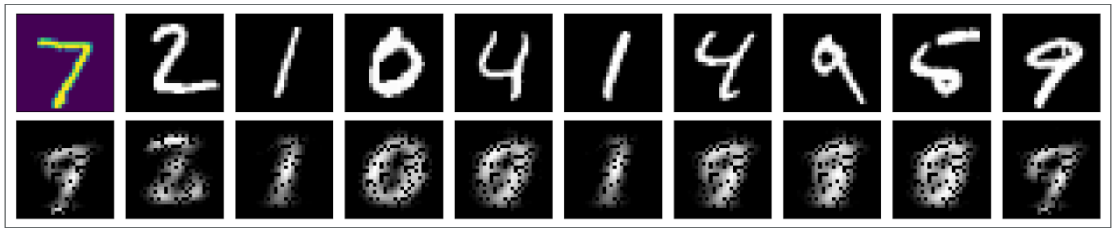
**（i）Eencoder 结果的可视化如图：**



自编码器 Encoder 输出可视化

上图中不同表示表示不同的数字，由图可知，自编码器降维之后的结果并不能很好地表示 10 个数字。

**（ii）autoencoder 还原之后的图片和原图片对比如下：**



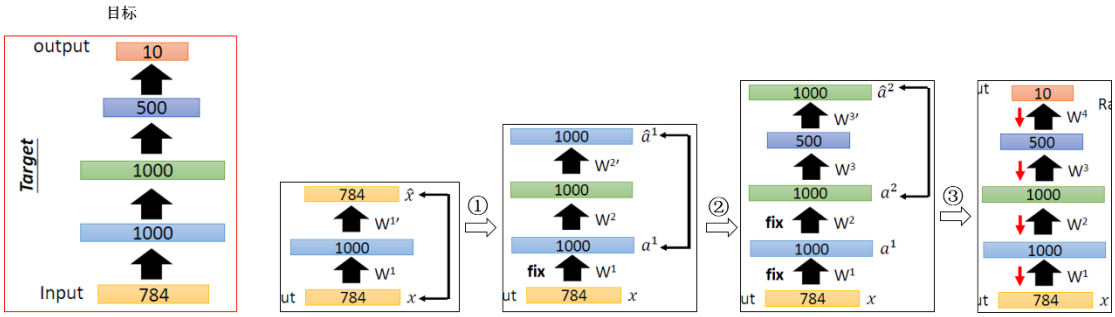
自编码器原图片和生成的图片对比

上图说明，autoencoder 的生成结果不是很清晰。

# 二、栈式自编码器（Stack Autoencoder）

## 2.1、栈式自编码器思想

栈式自编码器又称为深度自编码器，其训练过程和深度神经网络有所区别，下面是基于栈式自编码器的分类问题的训练过程：




栈式自编码器的训练过程

（图片来源：台大李宏毅老师的 PPT）

训练过程如下：

① 训练 784->1000->784 的自编码器，而后已经固定已经训练好的参数和 1000 维的结果，开始训练第二个自编码器。

② 训练 1000->1000->1000 的自编码器，而后固定已经训练好的参数和训练的中间层结果，开始训练第三个自编码器。

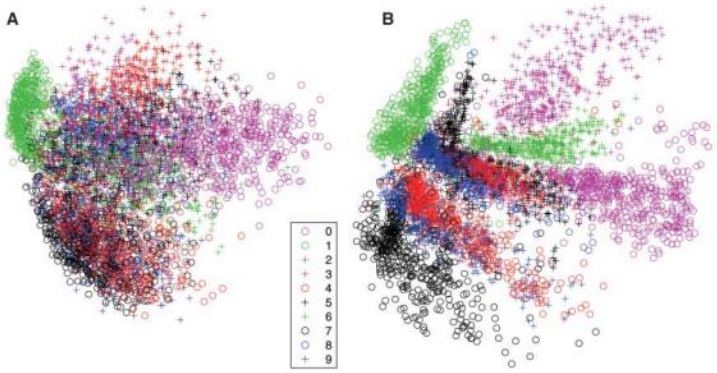
③ 训练 1000->500->1000 的自编码器，固定参数和中间隐层的结果。此时，前 3 层的参数已经训练完毕。

④ 最后一层接一个分类器，将整体网络使用反向传播进行训练，对参数进行微调。

这便是使用栈式自编码器进行分类的整体过程。

**注：**encoder 和 decoder 的参数可以是对称的，也可以是非对称的。

## 2.2、栈式自编码器效果




PCA 和栈式自编码器效果对比

上图左边是 PCA 从 784->2 维时数字的可视化结果，右图是使用栈式自编码器 784->1000->500->250->2 时的结果。（图片来源： Hinton G E, Salakhutdinov R R. Reducing the dimensionality of data with neural networks[J]. science, 2006, 313(5786): 504-507.），从图中可以看出，栈式自编码器的效果比 PCA 要好很多。

## 2.3、栈式自编码器实现与结果分析

**（1）实现框架：**Keras

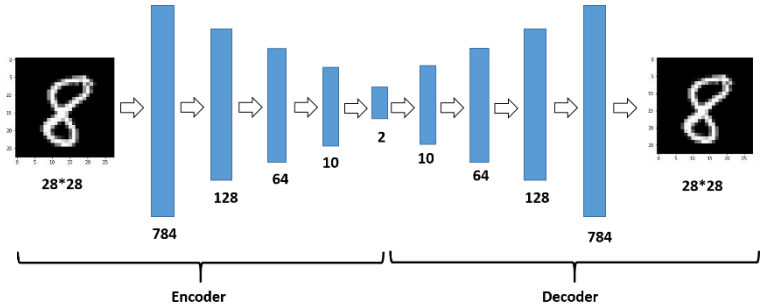
**（2）数据集：**Mnist 手写数字识别

**（3）关键代码：**

```
def train(x_train):
    # input placeholder
    input_image = Input(shape=(ENCODING_DIM_INPUT, ))
    # encoding layer
    encode_layer1 = Dense(ENCODING_DIM_LAYER1, activation='relu')(input_image)
    encode_layer2 = Dense(ENCODING_DIM_LAYER2, activation='relu')(encode_layer1)
    encode_layer3 = Dense(ENCODING_DIM_LAYER3, activation='relu')(encode_layer2)
    encode_output = Dense(ENCODING_DIM_OUTPUT)(encode_layer3)
    # decoding layer
    decode_layer1 = Dense(ENCODING_DIM_LAYER3, activation='relu')(encode_output)
    decode_layer2 = Dense(ENCODING_DIM_LAYER2, activation='relu')(decode_layer1)
    decode_layer3 = Dense(ENCODING_DIM_LAYER1, activation='relu')(decode_layer2)
    decode_output = Dense(ENCODING_DIM_INPUT, activation='tanh')(decode_layer3)
    # build autoencoder, encoder
    autoencoder = Model(inputs=input_image, outputs=decode_output)
```

```
        encoder = Model(inputs=input_image, outputs=encode_output)
        # compile autoencoder
        autoencoder.compile(optimizer='adam', loss='mse')
        # training
        autoencoder.fit(x_train, x_train, epochs=EPOCHS, batch_size=BATCH_SIZE,
shuffle=True)
        return encoder, autoencoder
```

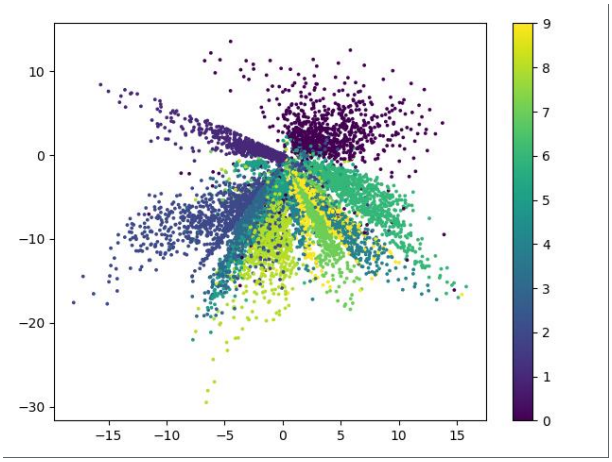
栈式自编码器相当于深度网络的过程，主要注意维度对应即可，另外，这里设置的 encoder 和 decoder 的维度是对称的。其架构图如下：




栈式自编码器代码架构

**（4）结果展示：**

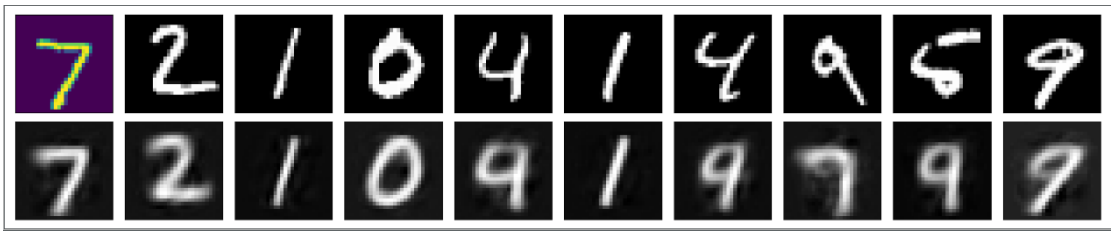
**（i）Eencoder 结果的可视化如图：**




栈式自编码器 Encoder 输出可视化

上图中不同表示表示不同的数字，由图可知，栈式自编码器的效果相比较普通自编码器好很多，这里基本能将 10 个分类全部分开。

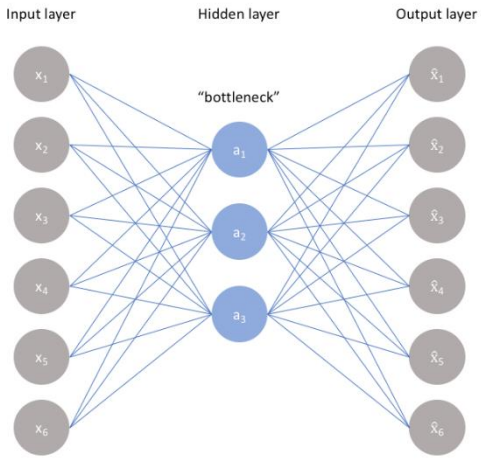
**（ii）autoencoder 还原之后的图片和原图片对比如下：**



# 三、欠完备自编码器（Undercomplete Autoencoder）

## 3.1、欠完备自编码器思想

由上述自编码器的原理可知，当隐层单元数大于等于输入维度时，网络会发生完全记忆的情况，为了避免这种情况，**我们限制隐层的维度一定要比输入维度小，这就是欠完备自编码器**，如下图所示。学习欠完备的表示将强制自编码器捕捉训练数据中**最显著**的特征。




欠完备自编码器的直观展示

（图片来源：https://www.jeremyjordan.me/autoencoders/）

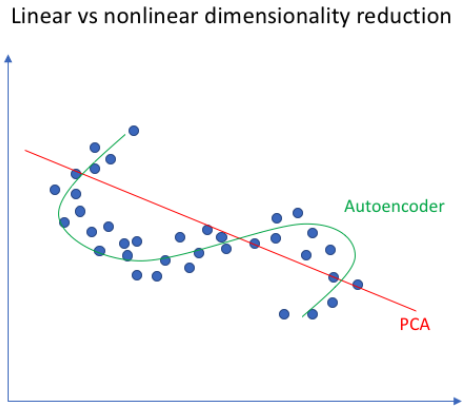
## 3.2、欠完备自编码器和主成分分析（PCA）的比较

实际上，若同时满足下列条件，欠完备自编码器的网络等同于 PCA，其会学习出于 PCA 相同的生成子空间：

（1） 每两层之间的变换均为线性变换。

（2） 目标函数$L(x, g(f(x))$为均方误差。

因此，拥有非线性编码器函数 $f$ 和非线性解码器函数 $g$ 的自编码器鞯狗学习出比 PCA 更强大的知识，其是 PCA 的非线性推广。如，下图是在二维空间中 PCA 算法和自编码器同时作用在二维点上做映射的结果，从图中可以看出，自编码器具有更好的表达能力，其可以映射到非线性函数。




二维空间点降维后在 PCA 和 Autoencoder 上的结果

（图片来源：https://www.jeremyjordan.me/autoencoders/）

### 3.3、欠完备自编码器特点

（1）防止过拟合，并且因为隐层编码维数小于输入维数，因此可以学习数据分布中最显著的特征。

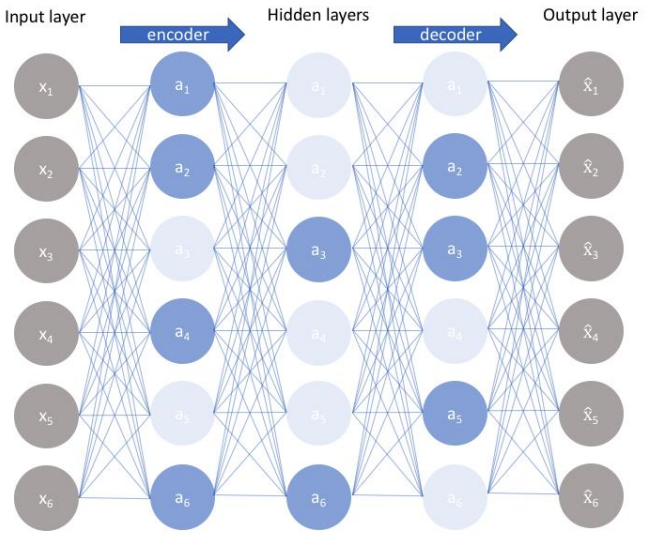
（2）若中间隐层单元数特别少，则其表达信息有限，会导致重构过程比较困难。

## 四、稀疏自编码器（Sparse Autoencoder）

### 4.1、稀疏自编码器思想

稀疏自编码器是加入正则化的自编码器，其未限制网络接收数据的能力，即不限制隐藏层的单元数。

**所谓稀疏性限制是指：**若激活函数是 sigmoid，则当神经元的输出接近于 1 的时候认为神经元被激活，输出接近于 0 的时候认为神经元被抑制。使得大部分神经元别抑制的限制叫做稀疏性限制。若激活函数是 tanh，则当神经元的输出接近于-1 的时候认为神经元是被抑制的。




稀疏自编码实例图

（图片来源：https://www.jeremyjordan.me/autoencoders/）

如上图所示，浅色的神经元表示被抑制的神经元，深色的神经元表示被激活的神经元。通过稀疏自编码器，我们没有限制隐藏层的单元数，但是防止了网络过度记忆的情况。

稀疏自编码器损失函数的基本表示形式如下：

$$L_{sparse}(x, g(f(x))) = L(x, g(f(x))) + \Omega(h)$$

其中$g(h)$是解码器的输出，通常$h$是编码器的输出，即$h = f(x)$。

## 4.2、损失函数和 BP 函数推导

损失函数可以加入 L1 正则化，也可以加入 KL 散度，下面是对加入 KL 散度的损失函数的分析。

**损失函数的分析如下：**

假设$a_j^{(2)}(x)$表示在给定输入$x$的情况下，自编码网络隐层神经元$j$的激活度，则神经元在所有训练样本上的平均激活度为：$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}[a_j^{(2)}(x^{(i)})]$。注意，$a_j^{(2)} = f(w_j^{(1)}x^{(i)} + b_j^{(1)})$。我们的目的使得网络激活神经元稀疏，所以可以引入一个稀疏性参数ρ，通常ρ是一个接近于 0 的值（即表示隐藏神经元中激活神经元的占比）。则若可以使得$\hat{\rho}_j = \rho$，则神经元在所有训练样本上的平均激活度$\hat{\rho}_j$便是稀疏的，这就是我们的目标。为了使得$\hat{\rho}_j = \rho$ ，我们使用 KL 散度衡量二者的距离，两者相差越大，KL 散度的值越大，KL 散度的公式如下：

$$\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j) = \sum_{j=1}^{s_2}[\rho log\frac{\rho}{\hat{\rho}_j} + (1-\rho)log\frac{1-\rho}{1-\hat{\rho}_j}]$$

$\rho$表示平均激活度的目标值。因此损失函数如下：

$$J_{sparse}(W,b) = J(W,b) + \beta\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j)$$

其中$J(W,b)$便是 NN 网络中的普通的代价函数，可以使用均方误差等。

**反向传播的分析如下：**

上式代价函数，左部分就是之前 BP 的结果，结果如下：

$$\frac{\partial J(W,b)}{\partial z_i^{(2)}} = \sum_{j=1}^{s_2} W_{ji}^{(2)}\frac{\partial J(W,b)}{\partial z_i^{(3)}} f'(z_i^{(2)})$$

可参考地址：http://ufldl.stanford.edu/wiki/index.php/反向传导算法。

右部分的求导如下：

$$\begin{aligned}\frac{\partial\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j)}{\partial z_i^{(2)}} &= \frac{\partial KL(\rho||\hat{\rho}_i)}{\partial z_i^{(2)}} \\ &= \frac{\partial KL(\rho||\hat{\rho}_i)}{\partial\hat{\rho}_i}\cdot\frac{\partial\hat{\rho}_i}{\partial z_i^{(2)}} \\ &= \frac{\partial(\rho\log\frac{\rho}{\hat{\rho}_i} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_i})}{\partial\hat{\rho}_i}\cdot\frac{\partial\hat{\rho}_i}{\partial z_i^{(2)}} \\ &= (-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})\cdot f'(z_i^{(2)})\end{aligned}$$

将二者结合，因此，稀疏自编码的反向传播结果如下：

$$\frac{\partial J_{sparse}(W,b)}{\partial z_i^{(2)}} = ((\left(W_{ji}^{(2)}\frac{\partial J(W,b)}{\partial z_i^{(3)}}\right) + \beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}))f'(z_i^{(2)})$$

根据此 BP 方程进行参数$W$和$b$的更新。

## 4.3、Mini-Batch 的情况

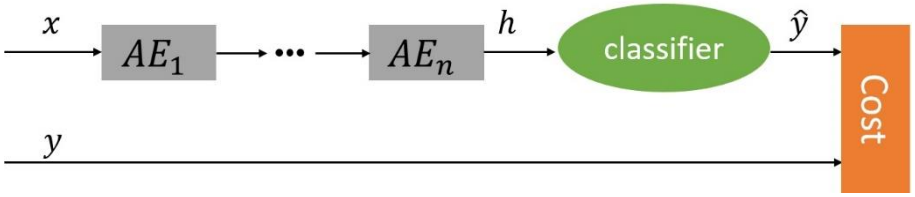
平均激活度是根据所有样本计算出来的，所以在计算任何单元的平均激活度之前，需要对所有样本计算一下正向传播，从而获得平均激活度，所以使用小批量时计算效率很低。要解决这个问题，可采取的方法是**只计算 Mini-Batch 中包含的训练样本的平均激活度，然后在 Mini-Batch 之间计算加权值：**

$$\hat{\rho}_j^t = \lambda\hat{\rho}_j^{t-1} + (1-\lambda)\hat{\rho}_j^t$$

其中，$\hat{\rho}_j^t$是时刻$t$的 Mini-Batch 的平均激活度，$\hat{\rho}_j^{t-1}$是时刻$t-1$的 Mini-Batch 的平均激活度。若λ大，则时刻$t-1$的 Mini-Batch 的平均激活度所占比重大，否则，时刻$t$的 Mini-Batch 的平均激活度所占比重大。

## 4.4、稀疏自编码器应用

稀疏自编码器一般用来学习特征，以便用于像分类这样的任务。




自编码器在分类上的应用

（图片来源：https://www.zhihu.com/question/41490383）

上述过程不是一次训练的，可以看到上面只有编码器没有解码器，因此其训练过程是自编码器先使用数据训练参数，然后保留编码器，将解码器删除并在后面接一个分类器，并使用损失函数来训练参数已达到最后效果。

## 4.5、稀疏自编码器实现与结果分析

**（1）实现框架：**Keras

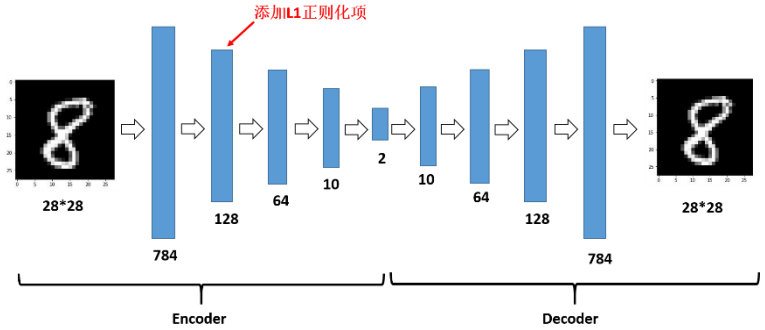
**（2）数据集：**Mnist 手写数字识别

**（3）关键代码：**

```
def train(x_train):
    # input placeholder
    input_image = Input(shape=(ENCODING_DIM_INPUT, ))
    # encoding layer
    # *****this code is changed compared with Autoencoder, adding the activity_regularizer to make the input sparse.
    encode_layer1 = Dense(ENCODING_DIM_LAYER1, activation='relu', activity_regularizer=regularizers.l1(10e-6))(input_image)
    # ********************************
    encode_layer2 = Dense(ENCODING_DIM_LAYER2, activation='relu')(encode_layer1)
    encode_layer3 = Dense(ENCODING_DIM_LAYER3, activation='relu')(encode_layer2)
    encode_output = Dense(ENCODING_DIM_OUTPUT)(encode_layer3)
    # decoding layer
    decode_layer1 = Dense(ENCODING_DIM_LAYER3, activation='relu')(encode_output)
    decode_layer2 = Dense(ENCODING_DIM_LAYER2, activation='relu')(decode_layer1)
    decode_layer3 = Dense(ENCODING_DIM_LAYER1, activation='relu')(decode_layer2)
    decode_output = Dense(ENCODING_DIM_INPUT, activation='tanh')(decode_layer3)
    # build autoencoder, encoder
    autoencoder = Model(inputs=input_image, outputs=decode_output)
    encoder = Model(inputs=input_image, outputs=encode_output)
    # compile autoencoder
    autoencoder.compile(optimizer='adam', loss='mse')
```

```
    # training
    autoencoder.fit(x_train, x_train, epochs=EPOCHS, batch_size=BATCH_SIZE,
shuffle=True)
    return encoder, autoencoder
```

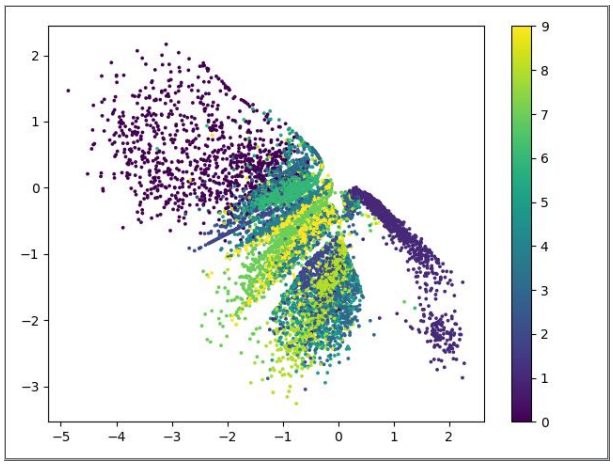
这里是以多层的自编码器举例，单隐层的同样适用，主要是在第一层加一个正则化项，activity_regularizer=regularizers.l1(10e-6)说明加入的是 L1 正则化项，10e-6 是正则化项系数，完整代码可参见最开始的 git。其架构如下：




稀疏自编码器代码架构

**（4）结果展示：**

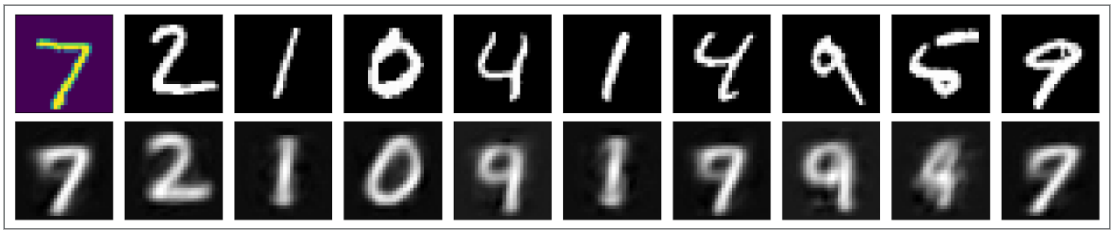
**（i）Encoder 结果的可视化如图：**



稀疏自编码器 Encoder 输出可视化

上图中不同颜色表示不同的数字，由图可知，这个编码器的分类效果还可以，比自编码器好很多，但是看起来还是作用不大，因为大部分作用需要归功于栈式自编码器。

**（ii）autoencoder 还原之后的图片和原图片对比如下：**



# 五、去噪自编码器（Denoising Autoencoder）

## 5.1、去噪自编码器思想

**去噪自编码器是一类接受损失数据作为输入，并训练来预测原始未被损坏的数据作为输出的自编码器。**如下图所示：

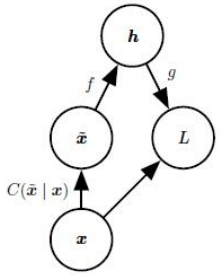



图 14.3: 去噪自编码器代价函数的计算图。去噪自编码器被训练为从损坏的版本 $\tilde{\boldsymbol{x}}$ 重构干净数据点 $\boldsymbol{x}$。这可以通过最小化损失 $L = -\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}}))$ 实现，其中 $\tilde{\boldsymbol{x}}$ 是样本 $\boldsymbol{x}$ 经过损坏过程 $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ 后得到的损坏版本。通常，分布 $p_{\text{decoder}}$ 是因子的分布（平均参数由前馈网络 $g$ 给出）。

去噪自编码器代价函数计算图

（图片来源：花书）

**其训练过程如下：**

引入一个损坏过程$C(\tilde{x}|x)$，这个条件分布代表给定数据样本$x$产生损坏样本$\tilde{x}$的概率。自编码器学习重构分布$p_{reconstruct}(x|\tilde{x})$：

- 从训练数据中采一个训练样本$x$
- 从$C(\tilde{x}|X = x)$中采一个损坏样本$\tilde{x}$
- 将$(\tilde{x}, x)$作为训练样本来估计自编码器的重构分布$p_{reconstruct}(x|\tilde{x}) = p_{decoder}(x|h)$，其中$h$是编码器$f(\tilde{x})$的输出，$p_{decoder}$是根据解码函数$g(h)$定义。

去噪自编码器中作者给出的直观解释是：类似于人体的感官系统，比如人的眼睛看物体时，如果物体的某一小部分被遮住了，人依然能够将其识别出来，所以去噪自编码器就是破坏输入后，使得算法学习到的参数仍然可以还原图片。

**注：噪声可以是添加到输入的纯高斯噪声，也可以是随机丢弃输入层的某个特性。**

## 5.2、去噪自编码器和 Dropout

噪声可以是添加到输入的纯高斯噪声，也可以是随机丢弃输入层的某个特性，即如果$C(\tilde{x}|x)$是一个二项分布，则其表现为下图所示内容，即经过处理的$x$的结果是保留或者舍掉，也就是说$C(\tilde{x}|x)$会舍去一部分内容，保留一部分内容：

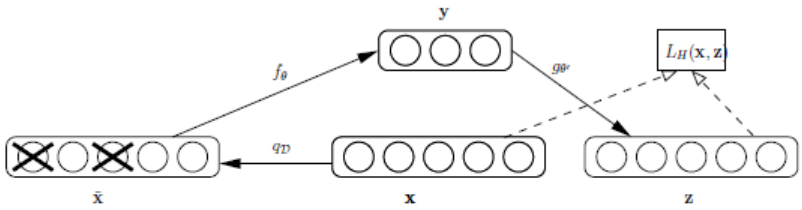



*Figure 1.* An example $\mathbf{x}$ is corrupted to $\tilde{\mathbf{x}}$. The autoencoder then maps it to $\mathbf{y}$ and attempts to reconstruct $\mathbf{x}$.

去噪自编码器随机舍去输入示意图

这个思想类似于 Dropout，但是二者还有一些区别：

（1） 去噪自编码器操作的是输入数据，相当于对输入数据去掉一部分内容；而 Dropout 操作的是网络隐藏层，相当于去掉隐藏层的一部分单元。

（2） Dropout 在分层预训练权值的过程中是不参与的，只是后面的微调部分会加入；而去噪自编码器是在每层预训练的过程中作为输入层被引入，在进行微调时不参与。

## 5.3、去噪自编码器和 PCA

去噪自编码器来源于论文[Vincent P, Larochelle H, Bengio Y, et al. Extracting and composing robust features with denoising autoencoders[C]//Proceedings of the 25th international conference on Machine learning. ACM, 2008: 1096-1103.]

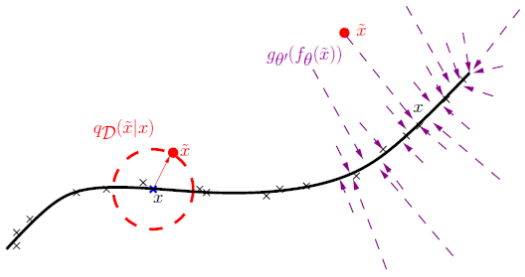



*Figure 2.* **Manifold learning perspective.** Suppose training data (×) concentrate near a low-dimensional manifold. Corrupted examples (•) obtained by applying corruption process $q_\mathcal{D}(\tilde{X}|X)$ will lie farther from the manifold. The model learns with $p(X|\tilde{X})$ to "project them back" onto the manifold. Intermediate representation $Y$ can be interpreted as a coordinate system for points on the manifold.

上图是去噪自编码器从流形角度的原理，图中黑色的曲线表示原始的局部流型，我们通过$C(\tilde{x}|x)$将其映射到一个圆的某一点，$\tilde{x}$表示添加噪声之后数据点。我们的目标是使得添加噪声之后的点能够映射到开始点，这样损失值为最小，所以可以理解为：图中红色箭头是噪声添加的向量场，而紫色部分是重构过程中需要不断查找的向量场。

因此，可以理解为，去噪自编码器的局部就是简化的 PCA 原理，其是对 PCA 的非线性扩展。

除此之外，文章还从信息论文、随机算子等角度理论上证明了去噪自编码器的可行性，有兴趣的读者可以参考上面提到的论文。

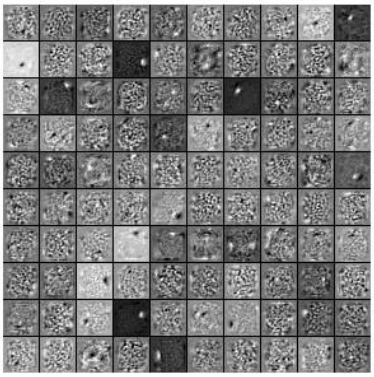
## 5.4、去噪自编码器结果分析

论文中的实验基于 Minst 数据集，其实验结果如下：

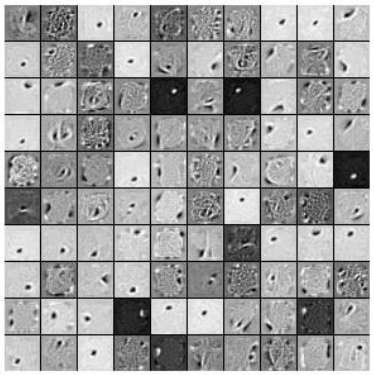
| Dataset | $SVM_{rbf}$ | $SVM_{poly}$ | DBN-1 | SAA-3 | DBN-3 | SdA-3 ($\nu$) |
|---|---|---|---|---|---|---|
| *basic* | **3.03±0.15** | 3.69±0.17 | 3.94±0.17 | 3.46±0.16 | 3.11±0.15 | **2.80±0.14** (10%) |
| *rot* | 11.11±0.28 | 15.42±0.32 | 14.69±0.31 | **10.30±0.27** | **10.30±0.27** | **10.29±0.27** (10%) |
| *bg-rand* | 14.58±0.31 | 16.62±0.33 | 9.80±0.26 | 11.28±0.28 | **6.73±0.22** | 10.38±0.27 (40%) |
| *bg-img* | 22.61±0.37 | 24.01±0.37 | **16.15±0.32** | 23.00±0.37 | **16.31±0.32** | **16.68±0.33** (25%) |
| *rot-bg-img* | 55.18±0.44 | 56.41±0.43 | 52.21±0.44 | 51.93±0.44 | 47.39±0.44 | **44.49±0.44** (25%) |
| *rect* | **2.15±0.13** | **2.15±0.13** | 4.71±0.19 | 2.41±0.13 | 2.60±0.14 | **1.99±0.12** (10%) |
| *rect-img* | 24.04±0.37 | 24.05±0.37 | 23.69±0.37 | 24.05±0.37 | 22.50±0.37 | **21.59±0.36** (25%) |
| *convex* | 19.13±0.34 | 19.82±0.35 | 19.92±0.35 | **18.41±0.34** | **18.63±0.34** | **19.06±0.34** (10%) |

其中，第一个和第二个 baseline 是使用高斯核和多项式核的 SVM，DBN-1 是 1 层隐层单元的深度信念网络，DBN-3 是 3 层隐层单元的深度信念网络，SAA-3 是使用栈式自编码器初始化之后的 3 层深度网络，最后的 Sda-3 是 3 层的栈式去噪自编码器。从表中可以看出，使用去噪自编码器的结果优于其他网络。

除此之外，网站 http://deeplearning.net/tutorial/dA.html 有去噪自编码器的代码和实验，其去噪前和去噪后过滤器的对比结果如下：



去噪前的过滤器



去噪后的过滤器

左图是去噪之前的过滤器数据，右图是去噪之后的过滤器数据，从图中可以看出，去噪自编码器学习到的特征更具代表性。

## 5.5、去噪自编码器特点

（1）普通的自编码器的本质是学一个相等函数，即输入和输出是同一个内容，这种相等函数的缺点便是当测试样本和训练样本不符合同一个分布时，在测试集上效果不好，而去噪自编码器可以很好地解决这个问题。

（2）欠完备自编码器限制学习绒里，而去噪自编码器允许学习容量很高，同时防止在编码器和解码器学习一个无用的恒等函数。

（3）经过了加入噪声并进行降噪的训练过程，能够强迫网络学习到更加鲁棒的不变性特征，获得输入的更有效的表达。

## 5.6、去噪自编码器实现与结果分析

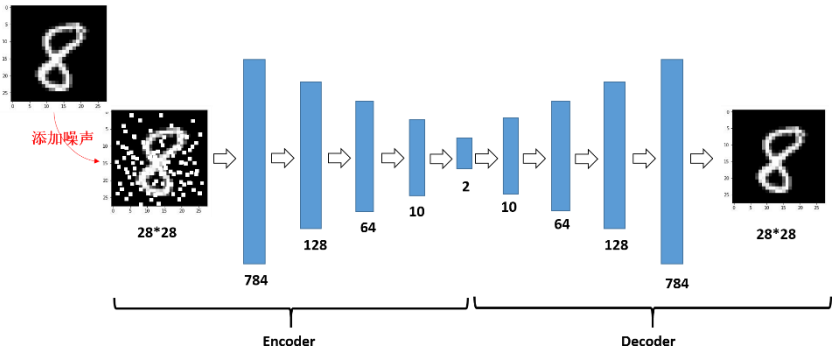
（1）**实现框架：**Keras

（2）**数据集：**Mnist 手写数字识别

（3）**关键代码：**

```
def addNoise(x_train, x_test):
    """
    add noise.
    :return:
    """
    x_train_noisy = x_train + NOISE_FACTOR * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
    x_test_noisy = x_test + NOISE_FACTOR * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
    x_train_noisy = np.clip(x_train_noisy, 0., 1.)         # limit into [0, 1]
    x_test_noisy = np.clip(x_test_noisy, 0., 1.)      # limit into [0, 1]
    return x_train_noisy, x_test_noisy
```

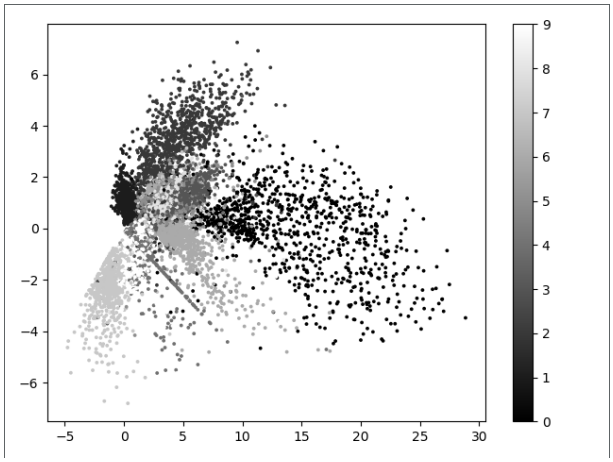
去噪自编码器主要是对输入添加噪声，所以训练过程是不需要改变的，只需要改变输入和输出。上述便是对输入添加噪声的过程，NOISE_FACTOR * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)便是添加的噪声。 np.clip()是截取函数，将数值限制在 0~1 之间。其架构如下：




去噪自编码器代码架构

（4）**结果展示：**

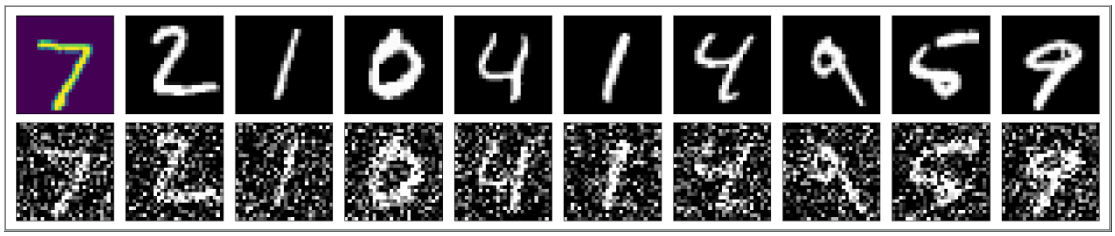
（i）Encoder 结果的可视化如图：



去噪自编码器 Encoder 输出可视化

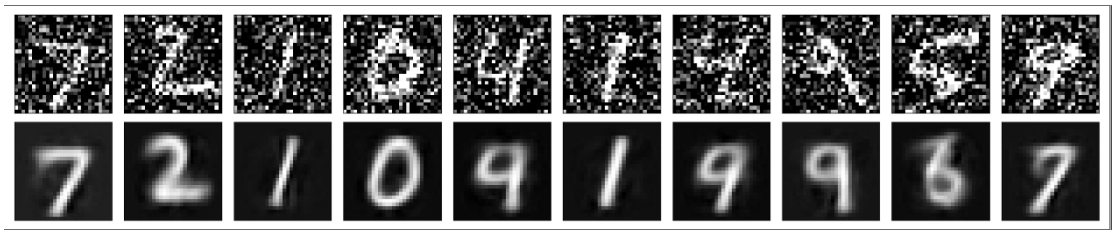
上图中不同表示表示不同的数字，这里不是很直观，看下面的图片对比

**（ii）autoencoder 还原之后的图片和原图片对比如下：**



添加噪声前后对比图

上图是添加噪声的效果对比，第一行表示原数据，第二行表示噪声处理过后的数据。



噪声数据和生成图片对比

上图根据噪声数据还原图片的对比，第一行表示噪声处理过后的数据，第二行表示去噪自编码器 decoder 还原之后的结果，上图可以看出，去噪自编码器的效果还是可以的。

# 六、其他参考文章：


[1] Kandeng. 自编码算法与稀疏性[EB/OL]. (2018/11/24)[ 2018/11/24] http://ufldl.stanford.edu/wiki/index.php/自编码算法与稀疏性

[2] Goodfellow I, Bengio Y, Courville A, et al. Deep learning[M]. Cambridge: MIT press, 2016.

[3] 山下隆义. 图解深度学习[M]. 人民邮电出版社，2018：68-78

[4] Francois Chollet. Building Autoencoders in Keras[EB/OL]. (2018/11/24)[ 2018/11/25] https://blog.keras.io/building-autoencoders-in-keras.html