非监督学习

布衣之莠

November 23, 2018

1 几种算法:

1.1 RBM:

基于能量(Energy-based)的模型
Restricted Bolzmann Machines(RBM,1986,Hiton)
Deep Brief Network(DBN,2006,Hinton)

1.2 AE:

Auto-Encoder
Denoising AE
Stacked AE
Sparsing AE
Variational AE(VAE,生成样本)

1.3 Cluster:

K-means
Mean-shift
Spectral clustering(谱聚类)
Hieraechical clustering
Density-Based Spatial Clustering of Applications with Noise(DBSCAN)
Blanced Iterative Reducing and Clustering(Birch)

1.4 Dimensionality reduction:

Singular value decomposition(SVD)
Principal component analysis(PCA),KernelPCA,SparsePCA
Independent component analysis(ICA),Kernel ICA
Linear Discriminant Analysis(LDA)
Factor Analysis
Non-negative matrix factorization(NMF)
Stochastic neighbor embedding(SNE,2002,Hinton),对称SNE，t-SNE,LargeVis(唐

建),注:用于低维空间可视化
Dictionary Learning(Sparse Representation)
Manifold Learning:*MDS(Multidimensional Scaling),ISOMAP(Isometric feature mapping),LLE(Locally linear embedding),LE(Laplacian Eigenmap)*

1.5 GAN:

生成样本
DCGAN
WGAN
InfoGAN

2 受限波尔兹曼机

2.1 RBM

RBM结构:

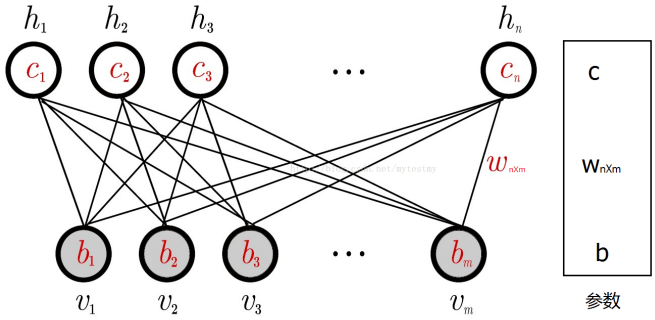

Figure 1: RBM

RBM网络结构有m个可视节点和n个隐藏节点，可视层和隐藏层之间的权重矩阵为$W_{m\times n}$,可视节点的bias为$b = (b_1, b_2, \ldots, b_m)$,隐藏节点的bias为$c = (c_1, c_2, \ldots, c_n)$。 RBM可以用于降维，用于初始化神经网络参数。
基于能量的模型。 能量函数定义如下:

$$E(v,h) = -\sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij}h_i v_j - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \tag{2.1.1}$$

定义可视节点和隐藏节点的联合概率为:

$$P(v,h) = \frac{e^{-E(v,h)}}{\sum_{v,h} e^{-E(v,h)}} \tag{2.1.2}$$

这样定义的概率符合Gibbs分布。

条件概率和边缘概率是:

$$P(v) = \frac{\sum_h e^{-E(v,h)}}{\sum_{v,h} e^{-E(v,h)}} \tag{2.1.3}$$

$$P(h) = \frac{\sum_v e^{-E(v,h)}}{\sum_{v,h} e^{-E(v,h)}} \tag{2.1.4}$$

$$P(v|h) = \frac{e^{-E(v,h)}}{\sum_h e^{-E(v,h)}} \tag{2.1.5}$$

$$P(h|v) = \frac{e^{-E(v,h)}}{\sum_v e^{-E(v,h)}} \tag{2.1.6}$$

从概率到最大似然,最大似然定义:

$$L(\theta) = \frac{1}{N} \sum_{n=1}^{N} \ln P_\theta(v_n) \tag{2.1.7}$$

可以通过梯度下降来最大化$L(\theta)$
对于每个$\ln p(v)$

$$\begin{aligned} &\frac{\partial \ln p(v)}{\partial W_{ij}} \\ &\quad = \sum_h p(h|v) h_i v_j - \sum_{v,h} p(v,h) h_i v_j \\ &\quad = \sum_h p(h|v) h_i v_j - \sum_v p(v) \sum_h p(h|v) h_i v_j \\ &\quad = p(h_i = 1|v) v_j - \sum_v p(v) p(h_i = 1|v) v_j \end{aligned} \tag{2.1.8}$$

其中:

$$p(h_i = 1|v) = \frac{\sum_{h_k \neq i} p(h_i, h_{k \neq i}, v)}{\sum_h p(v,h)} = \sigma(\sum_{j=1}^{m} w_{ij} v_j + c_i) \tag{2.1.9}$$

可以轻易求解，但第二项就要遍布所有v值，这样很麻烦。

我们可以通过对训练样本抽样得到符合模型Gibbs分布的样本，然后直接用这些样本估算以上条件概率
对每个训练样本x，都用某种抽样方法抽取一个它对应的符合RBM网络表示的Gibbs分布的样本（对应的意思就是符合参数确定的Gibbs分布p(x)的），假如叫y；那么，对于整个的训练集$x_1, x_2, \ldots x_n$来说，就得到了一组符合RBM网

络表示的Gibbs分布的样本$y_1, y_2, \ldots, y_n$，然后拿这组样本去估算第二项，那么梯度就可以用下面的公式来近似了:

$$\begin{aligned}\frac{\partial \ln p(v)}{\partial W_{ij}} \\ &= p(h_i = 1|v)v - \sum_v p(v)p(h_i = 1|v)v \\ &= p(h_i = 1|v)v - \frac{1}{n}\sum_{j=1}^{n} p(h_i = 1|v_y)v_{yj}\end{aligned} \tag{2.1.10}$$

我们有了上面的条件概率，我们就可以交替地进行下面的采样:
$h_0 <p(h|v_0), v_1 <p(v|h_0)$
$h_1 <p(h|v_1)v_2 <p(v|h_1)$
$\cdots v_{k+1} <p(v|h_k)$

在抽样步数k足够大的情况下，就可以获得符合RBM模型Gibbs分布的样本，得到这些样本就可以计算梯度的第二项了。

通过Constrastive Divergence来学习

Algorithm 1. k-step contrastive divergence

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

1 init $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$
2 **forall the** $v \in S$ **do**
3 $v^{(0)} \leftarrow v$
4 **for** $t = 0, \ldots, k-1$ **do**
5 **for** $i = 1, \ldots, n$ **do** sample $h_i^{(t)} \sim p(h_i \mid v^{(t)})$
6 **for** $j = 1, \ldots, m$ **do** sample $v_j^{(t+1)} \sim p(v_j \mid h^{(t)})$
7 **for** $i = 1, \ldots, n$, $j = 1, \ldots, m$ **do**
8 $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 \mid v^{(0)}) \cdot v_j^{(0)} - p(H_i = 1 \mid v^{(k)}) \cdot v_j^{(k)}$
9 $\Delta b_j \leftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$
10 $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 \mid v^{(0)}) - p(H_i = 1 \mid v^{(k)})$

Figure 2: CD-K算法

参考:限制波尔兹曼机
参考:限制波尔兹曼机

2.2 DBN

Deep Brief Network

把原始数据输入到最下面的RBM可视层中，然后训练RBM1，训练完成之后把RBM1的隐含层作为RBM2的可视层，继续训练RBM2，接下来把RBM2的隐含层做为RBM3的可视层，直到训练完成为止。RBM的最后一隐含层层与输出层连接，对该层进行bp优化。 参考:深度信念网络

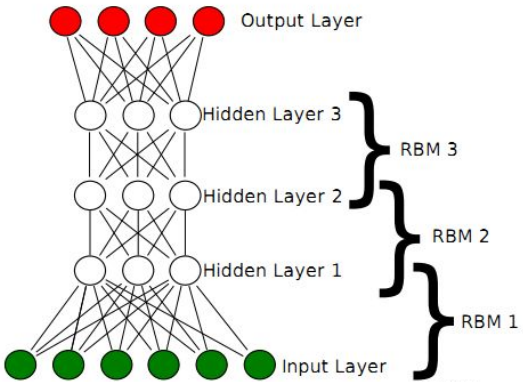

Figure 3: DBN

3 自编码器

3.1 AutoEncoder

AE结构

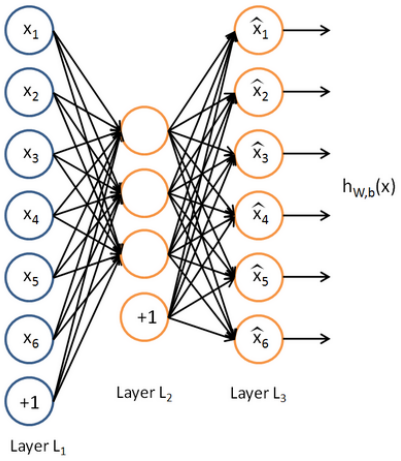

Figure 4: AutoEncoder

自编码神经网络尝试学习一个 $h_{W,b}(x) \approx x$ 的函数。它尝试逼近一个恒等函数，从而使得输出 $\hat{x}$ 接近于输入 x 。当我们为自编码神经网络加入某些限制，比如限定隐藏神经元的数量，自编码神经网络可以去学习输入数据的压缩表示。

3.2 Denoising AE

DAE(Denoising Autoencoder)的核心思想是，一个能够从中恢复出原始信号的神经网络表达未必是最好的，能够对“损坏”的原始数据编码、解码，然后还能恢复真正的原始数据，这样的特征才是好的。

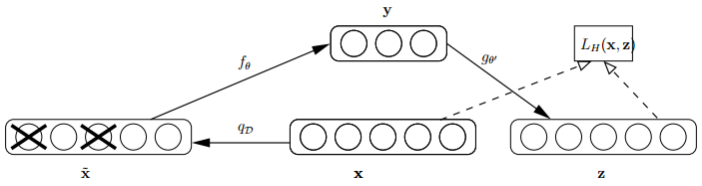

Figure 5: Denosing AE

参考:Extracting and Composing Robust Features with Denoising Autoencoders

3.3 Stacked AE

SAE:AE逐层堆叠, layer-wise unsuperwised pre-training

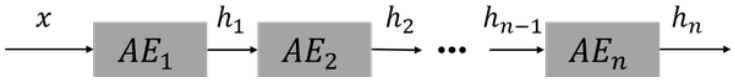

Figure 6: Stacked AE

多个稀疏自编码器与softmax分类器结合，构建一个包含多隐藏层与一个最终softmax分类器的SAE网络，其训练步骤如下
Step1：将原始数据作为 SAE 的输入，训练第一个隐藏层的网络参数，并用训练好的参数算出第一个隐藏层的输出；
Step2:把上一层网络的输出作为其下一层网络的输入，用同样的方法训练该层网络的参数；重复这一步骤，直到训练完最后一个隐藏层；
Step3:将 Step2 中的输出作为 softmax 分类器的输入，结合原始数据的标签来训练 softmax 分类器的网络参数；
Step4:计算整个网络(包括所有隐藏层和一个 softmax 分类器)的代价函数，以及该网络对每个参数的偏导函数值；
Step5:用 Step1，Step2 和 Step3 的网络参数作为整个深度网络的初始化参数值，然后用优化算法迭代求出代价函数最小值附近的参数值，并作为整个网络最后的最优参数值。

3.4 Sparse AE

AE隐藏神经元的数量较大，我们仍然通过给自编码神经网络施加一些其他的限制条件来发现输入数据中的结构。

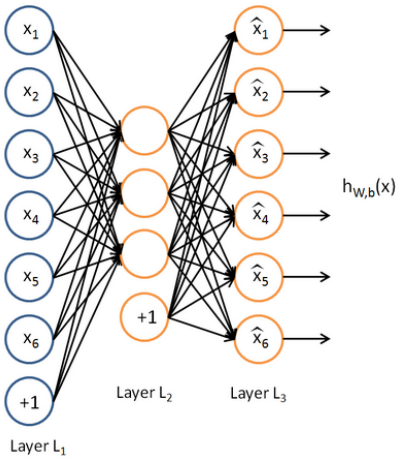

Figure 7: AutoEncoder

稀疏性限制:
神经元的输出接近于1的时候被激活，而输出接近于0的时候被抑制，那么使得神经元大部分的时间都是被抑制的限制则被称作稀疏性限制。假设的神经元的激活函数是sigmoid函数。tanh作为激活函数的话，当神经元输出为-1的时候，神经元是被抑制的。
我们将使用 $a_j^{(2)}(x)$ 来表示在给定输入为 x情况下，自编码神经网络隐藏神经元j的激活度。 进一步，让 $\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right]$ 表示隐藏神经元 j的平均活跃度（在训练集上取平均）。我们可以近似的加入一条限制 $\hat{\rho}_j = \rho$, 其中，ρ是稀疏性参数，通常是一个接近于0的较小的值（比如 $\rho = 0.05$ ）。
为了实现这一限制，我们将会在我们的优化目标函数中加入一个额外的惩罚因子，而这一惩罚因子将惩罚那些 $\hat{\rho}_j$和 ρ有显著不同的情况从而使得隐藏神经元的平均活跃度保持在较小范围内。惩罚因子 $\sum_{j=1}^{s_2}\rho\log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}$. 这里， s_2是隐藏层中隐藏神经元的数量，而索引 j 依次代表隐藏层中的每一个神经元。
参考:UFLDL Tutorial

3.5 Variational AE

变分自编码器 贝叶斯神经网络和变分推断
优化目标ELBO(Evidence Lower Bound)

$$ELBO = \log p(x) - KL[q(z|x) \parallel p(z|x)] \tag{3.5.1}$$

可以简化为:

$$ELBO = E_{z\sim q}[\log p(x|z)] - KL(q(z|x) \parallel p(z)) \tag{1}$$

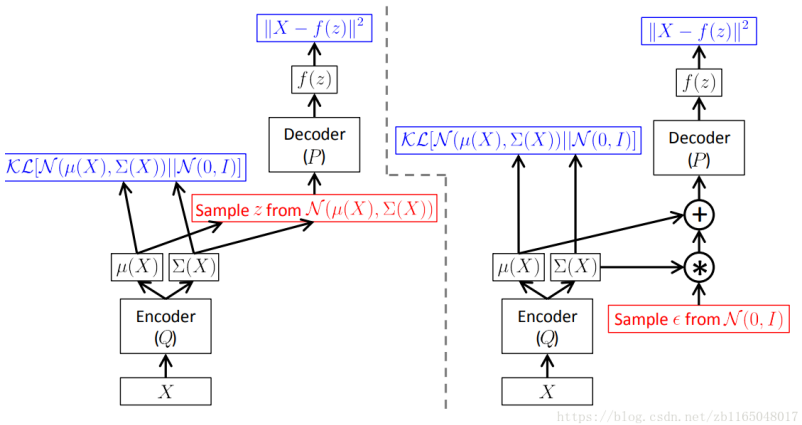

Figure 8: Variational AE

最终损失为:

$$L = \| X - f(z) \|^2 - \lambda * KL(q(z|x) \| p(z)) \tag{2}$$

重新参数化(reparameterization trick)

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparametrize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()#exp(0.5*var)
        eps = Variable(std.data.new(std.size()).normal_())
        return eps.mul(std).add_(mu)#eps*std+u

    def decode(self, z):
        h3 = self.relu(self.fc3(z))
        return self.sigmoid(self.fc4(h3))
```

```
    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))#[ size 32x20 (GPU
            0)],[size 32x20 (GPU 0)]
        z = self.reparametrize(mu, logvar)
        return self.decode(z), mu, logvar

reconstruction_function = nn.BCELoss()
def loss_function(recon_x, x, mu, logvar):
    BCE = reconstruction_function(recon_x, x.view(-1, 784))
    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD_element =
        mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)

    return BCE + KLD
```

参考:Auto-Encoding Variational Bayes
参考:Tutorial on Variational Autoencoders
参考:变分自编码器

4 聚类

Cluster

5 降维

Dimensionality reduction

6 对抗生成网络

GAN