机器学习实验报告

强基数学 *002* 吴天阳 *2204210460*

本报告包含 K-均值聚类、混合高斯 (GMM)、去偏变分自动编码机 (DB-VAE) 三个实验部分，每个部分的完整算法代码均在代码文件夹 `./code` 下给出.

1 K-均值聚类

设数据集为 $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N\}$, $\boldsymbol{x}_i \in \mathbb{R}^D$, 由 N 个在 $\mathbb{R}^D$ 中的观测值构成. K-均值聚类 (K-means Clustering) 的目标是将数据集划分为 K 簇 (Cluster), 假设 $\boldsymbol{\mu}_k \in \mathbb{R}^D$, $(k = 1, \cdots, K)$ 代表簇的中心，我们的目标是最小化每个数据点到最近 $\boldsymbol{\mu}_k$ 的距离平方和.

为方便描述每个数据点的分类，引入二进制指标集 $r_{nk} \in \{0, 1\}$，如果数据点 $\boldsymbol{x}_n$ 分配到簇 k，那么 $r_{nk} = 1$ 且 $r_{nj} = 0$, $(j \neq k)$. 根据该编码方法，可以定义以下最小化目标**失真度量 (distortion measure)**：

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2 \tag{1}$$

K-均值聚类也是用了 EM 算法，首先给出如何划分为 EM 算法：

- E 步：固定 μ_k，求使得 J 最小化的 r_{nk}（求出期望）.
- M 步：固定 r_{nk}，求使得 J 最小化的 μ_k（最大化）.

E 步 当固定 μ_k 时，由于 J 关于 r_{nk} 是线性的，即不同的数据 $\boldsymbol{x}_n$ 之间相互独立，所以可以对于每个样本单独进行优化，对于样本 $\boldsymbol{x}_n$ 的 r_{nk} 满足求解以下最优化问题：

$$\min_{r_{nk} \in \{0,1\}} \sum_{k=1}^{K} r_{nk} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2 \quad s.t. \sum_{k=1}^{K} r_{nk} = 1 \Rightarrow r_{nk} = \begin{cases} 1, & \text{当 } k = \arg\min\limits_{1 \leqslant j \leqslant K} ||\boldsymbol{x}_n - \boldsymbol{\mu}_j||^2, \\ 0, & \text{否则.} \end{cases}$$

不难发现，最优化结果正好就表明只需将每个 $\boldsymbol{x}_n$ 分配到最近的簇中心 $\boldsymbol{\mu}_k$ 上.

M 步 当固定 r_{nk} 时，由于 J 是关于 μ_k 的二次函数，所以可以通过导数为零确定最小化点：

$$2 \sum_{n=1}^{N} r_{nk} (\boldsymbol{x}_n - \boldsymbol{\mu}_k) = 0 \Rightarrow \boldsymbol{\mu}_k = \frac{\sum\limits_{1 \leqslant n \leqslant N} r_{nk} \boldsymbol{x}_n}{\sum\limits_{1 \leqslant n \leqslant N} r_{nk}}$$

表达式中分母是簇 k 分配到的数据的个数，所以 μ_k 的更新就是所有簇 k 分配到的数据点的平均值，因此被称为 K-均值，算法总共两步：将每个数据点分配到最近的簇，重新计算簇均值以替代新的簇. 由于该方法每一步都会使得 J 单调递减，所以算法收敛性显然，但是它只能收敛到 J 的局部最小值.

2 混合高斯模型

混合高斯模型 (Gaussian Mixture Model, GMM) 是一种基于概率的聚类模型，首先我们可以将所有的变量均视为随机变量，包括观测变量 x 和隐变量 θ, ω，它们都服从某个概率分布，于是可以将模型参数求解转化为求解 $p(\theta|D)$，即根据数据集 D 求出模型参数 θ 的后验分布，所以可以用最大似然 (MLE) 方法求解.

用上述方法理解重新聚类问题：混合概率模型的隐变量就是 y 表示数据的类别种类，服从分布 $p(y=k)=\pi_k$（表示全部的 $\boldsymbol{x}$ 来自类别 k 的概率大小），从**数据生成**的角度理解，第 k 个类别的数据 $\boldsymbol{x}$ 应来自与 y 相关的某个分布 $p(\boldsymbol{x}|y=k)$ 中（不妨令该分布为多维正态分布），于是二者的联合分布为

$$p(\boldsymbol{x},y)=p(y)p(\boldsymbol{x}|y)\overset{y=k}{=\!=\!=}\pi_k\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\Sigma_k)$$

通过联合分布我们又可以求出**数据预测**的结果：

$$p(y|\boldsymbol{x})=\frac{p(\boldsymbol{x},y)}{p(\boldsymbol{x})}\overset{y=k}{=\!=\!=}\frac{\pi_k\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\Sigma_k)}{\sum_{k=1}^{K}\pi_k\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\Sigma_k)}$$

将全部的参数简记为 $\theta=(\mu_1,\cdots,\mu_K,\sigma^2,\pi_1,\cdots,\pi_K)$，于是关于 θ 的 MLE 为

$$\max_{\theta}\prod_{i=1}^{N}p(x_i|\theta)=\prod_{i=1}^{N}\sum_{k=1}^{K}p(x_i,y_i=k|\theta)=\prod_{i=1}^{N}\sum_{k=1}^{K}p(y_i=k|\theta)p(x_i|y_i=k,\theta)\tag{2}$$

2.1 由 GMM 导出 K-均值

我们考虑一个 GMM 的特殊情况，假设所有的方差均相同，即 $\Sigma=\Sigma_k,\ (k=1,\cdots,K)$，并令 $\pi_k=\dfrac{1}{n}\sum_{i=1}^{n}r_{nk}$，则 $\boldsymbol{\mu}_k$ 似然函数为

$$\begin{aligned}L&=\prod_{i=1}^{N}\sum_{k=1}^{K}\pi_k p(x_i)=\prod_{i=1}^{N}\sum_{k=1}^{K}\pi_k\mathcal{N}(x_i|\boldsymbol{\mu}_k,\Sigma)\\&=\prod_{i=1}^{N}\sum_{k=1}^{K}\pi_k(2\pi)^{-\frac{k}{2}}|\Sigma|^{-\frac{1}{2}}\exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)\right\}\end{aligned}$$

取对数后得到 MLE 为

$$\max_{\boldsymbol{\mu}_k}-\sum_{i=1}^{N}\sum_{k=1}^{K}\pi_k||x_i-\boldsymbol{\mu}_k||^2=\min_{\boldsymbol{\mu}_k}\sum_{i=1}^{N}\sum_{k=1}^{K}\pi_k||x_i-\boldsymbol{\mu}_k||^2$$

结果与 K-均值 (1) 式中的失真度量 J 的区别仅需将 π_k 换为 r_{nk}，而这个转换就会将聚类方法从 GMM 的**软分类**变为 K-均值的**硬分类**.

其次我们可以利用 GMM 的预测方法证明：当方差相同时（K-均值的分类边界），分类边界是线性的. 假设数据 $\boldsymbol{x}$ 分到 i,j 类别具有相同的可能性时，也就是 $p(y=i|\boldsymbol{x})=$

$p(y=j|\boldsymbol{x})$，于是：

$$
\begin{aligned}
0 &= \log\frac{p(y=i|\boldsymbol{x})}{p(y=j|\boldsymbol{x})} = \log\frac{p(\boldsymbol{x}|y=i)\frac{p(y=i)}{p(\boldsymbol{x})}}{p(\boldsymbol{x}|y=j)\frac{p(y=j)}{p(\boldsymbol{x})}} = \log\frac{p(\boldsymbol{x}|y=i)\pi_i}{p(\boldsymbol{x}|y=j)\pi_j} \\
&= \log\frac{\pi_i}{\pi_j} + ||\boldsymbol{x}-\boldsymbol{\mu}_i||^2 - ||\boldsymbol{x}-\boldsymbol{\mu}_j||^2 = \log\frac{\pi_i}{\pi_j} + 2(\boldsymbol{\mu}_j^T - \boldsymbol{\mu}_i^T)\boldsymbol{x} + \boldsymbol{\mu}_i^T\boldsymbol{\mu}_i - \boldsymbol{\mu}_i^T\boldsymbol{\mu}_j \\
&\Rightarrow \boldsymbol{w}^T\boldsymbol{x} = \boldsymbol{b}
\end{aligned}
$$

说明如果 $\boldsymbol{x}$ 分为类别 i,j 的可能性相同时，则 $\boldsymbol{x}$ 一定处于直线 $\boldsymbol{w}^T\boldsymbol{x}=\boldsymbol{b}$ 上. 同理，对于一般情况 Σ_k，计算得到的分类边界为 $\boldsymbol{x}^TW\boldsymbol{x}+\boldsymbol{w}^T\boldsymbol{x}+\boldsymbol{c}$，说明 GMM 的分类边界就是二次函数.

2.2 使用 EM 算法进行参数求解

观察 (2) 式，对其取对数仍然无法将内部的求和符号展开成线性表示，所以难以求出极值，考虑基于 $p(x,y)$ 求 θ 的 MLE：

$$
\max_\theta \prod_{i=1}^N p(\boldsymbol{x}_i,y_i|\theta) \propto \sum_{i=1}^N \log p(\boldsymbol{x}_i,y_i|\theta) = \sum_{i=1}^N\sum_{k=1}^K p(y_i=k|\boldsymbol{x}_i,\theta)\log p(\boldsymbol{x}_i,y_i|\theta)
$$

M 步：假设我们在 $t-1$ 步已经得到了参数估计值 θ^{t-1}，于是可以在第 t 步建立 Q 函数，然后最大化该函数得到 θ^t

$$
\max_{\theta^t} Q(\theta^t|\theta^{t-1}) = \sum_{i=1}^n\sum_{k=1}^K p(y_i=k|\boldsymbol{x}_i,\theta^{t-1})\log p(\boldsymbol{x}_i,y_i=k|\theta^t) \tag{3}
$$

E 步：就是在 $t-1$ 步时，基于 θ^{t-1} 计算数据 $\boldsymbol{x}_i$ 从属于每个类别的概率：

$$
R_{i,k}^{t-1} = p(y_i=k|\boldsymbol{x}_i,\theta^{t-1}) = \frac{\pi_k^{t-1}\mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_k^{t-1},\Sigma_k^{t-1})}{\sum_{k=1}^K \pi_k^{t-1}\mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_k^{t-1},\Sigma_k^{t-1})}
$$

而 E 步的计算结果，就是 M 步中 Q 函数中 $\log p(\boldsymbol{x}_i,y_i=k|\theta^t)$ 前的加权系数.

我们先不探讨上述方法的收敛性，通过 Lagrange 乘子法和令导数为零可以求解求解 (3) 式：

$$
\text{由 Lagrange 乘子法：}\ \nabla_{\pi_k^t}Q + \lambda\nabla_{\pi_k^t}\left(\sum_{k=1}^K\pi_k - 1\right) = 0 \Rightarrow \frac{\sum_{i=1}^N R_{i,1}^{t-1}}{\pi_1^t} = \cdots = \frac{\sum_{i=1}^N R_{i,k}^{t-1}}{\pi_k^t}
$$

$$
\xrightarrow{\sum_{k=1}^K R_{i,k}^{t-1}=1} \pi_k^t = \frac{\sum_{i=1}^N R_{i,k}^{t-1}}{N}
$$

$$
\nabla_{\boldsymbol{\mu}_k^t}Q = 0 \Rightarrow \sum_{i=1}^N R_{i,k}^{t-1}(\boldsymbol{x}_i - \boldsymbol{\mu}_k^t) = 0 \Rightarrow \boldsymbol{\mu}_k^t = \sum_{i=1}^N w_{ik}\boldsymbol{x}_i \tag{4}
$$

$$
\nabla_{\Sigma_k^t}Q = 0 \Rightarrow \sum_{i=1}^N R_{i,k}^{t-1}(-\Sigma_k^t + (\boldsymbol{x}_i-\boldsymbol{\mu}_k^t)^T(\boldsymbol{x}_i-\boldsymbol{\mu}_k^t)) = 0 \Rightarrow \Sigma_k^t = \sum_{i=1}^N w_{ik}(\boldsymbol{x}_i-\boldsymbol{\mu}_k^t)^T(\boldsymbol{x}_i-\boldsymbol{\mu}_k^t)
$$

其中 $w_{ik} = \dfrac{R_{i,k}^{t-1}}{\sum_{i=1}^N R_{i,k}^{t-1}}$. 通过上式中的标红的部分，就可以得到第 t 步下的新参数值.

2.3 EM 算法收敛性证明

首先引入一个求解函数极值的技术：若要求解 $f(x)$ 的极小值，可以先取其定义域上任意一点 x_0，再找到一个在 $(x_0, f(x_0))$ 处与 f 相切的函数 $g(x)$，并且要求 $g(x)$ 是 $f(x)$ 的上界，令 $x_1 \leftarrow \arg\min_x g(x)$（求极大值反之亦然），根据该方法进行迭代即可得到 $f(x)$ 的极小值点.

下面推到一个关于 θ 的 MLE 重要结论，该方程成为**变分方程**：

$$\begin{aligned}\log p(x|\theta) &= \int_Y q(y)\log p(x|\theta)\,\mathrm{d}y = \int_Y q(y)\log\frac{p(x,y|\theta)}{p(y|x,\theta)}\frac{q(y)}{q(y)}\,\mathrm{d}y \\ &= \underbrace{\int_Y q(y)\log p(x,y|\theta)\,\mathrm{d}y}_{\text{核函数}} - \underbrace{\int_Y q(y)\log q(y)\,\mathrm{d}y}_{\text{与}\theta\text{无关}} + \underbrace{\int_Y q(y)\log\frac{q(y)}{p(y|x,\theta)}\,\mathrm{d}y}_{\mathrm{KL}(q||p)}\end{aligned} \tag{5}$$

注意到右边第三项正好是 p, q 的 KL 散度，于是有 $\mathrm{KL}(q||p) \geqslant 0$，于是前两项构成 $\log p(x|\theta)$ 的下界，要求极大似然的极大值，第二项与 θ 无关，所以只需对第一项求即可. 注意上式只讨论了一个数据，极大似然是对所有数据的对数似然求和得到：

$$\max_\theta \sum_{i=1}^N \int_Y q(y)\log p(\boldsymbol{x}_i, y|\theta)\,\mathrm{d}y$$

于是当我们取 $q(y) = p(y|\boldsymbol{x}, \theta)$ 时，KL 散度正好为 0，极大似然对应的 θ^* 可以通过以下迭代式求解：

$$\theta^* \leftarrow \arg\max_{\theta^*} \sum_{i=1}^N \int_Y p(y|\boldsymbol{x}_i, \theta)\log p(\boldsymbol{x}_i, y|\theta^*)\,\mathrm{d}y$$

将积分号换为求和符号就可以得到 EM 算法中 M 步的 (3) 式. 由上面引入的函数极值求解技术，可以证明每次 EM 都可以使得 MLE 下降，从而达到极小值点，说明 EM 算法具有收敛性.

3 变分自动编码机

3.1 基本思想

变分自动编码机（Variational AutoEncoder, VAE），是一种通过完全无监督的方式学习数据中的潜在特征编码. 参考论文Auto-Encoding Variational Bayes.

设隐参数维数为 K，记数据 $\boldsymbol{x}$ 对应的真实隐参数为 $\boldsymbol{z} \in \mathbb{R}^K$. 基于所有变量都服从某个参数分布的思路，我们可以定义以下两个分布函数：

- $q_\phi(\boldsymbol{z}) = q(\boldsymbol{z}|\boldsymbol{x}, \phi)$ 称为**编码分布**，该网络的输入为数据 $\boldsymbol{x}$，通过网络参数 ϕ，估计出隐参数 $\boldsymbol{z}$ 的分布.
- $p_\theta(\boldsymbol{x}) = p(\boldsymbol{x}|\boldsymbol{z}, \theta)$ 称为**解码分布**，该网络的输入为隐参数 $\boldsymbol{z}$，通过网络参数 θ，得到重建数据 $\hat{\boldsymbol{x}}$ 的分布.

注：上面两个函数均为概率密度函数，输入为随机变量，输出为对应的概率值；而非神经网络函数，如果要表示编码神经网络，记法不变，则**编码网络**为 $q_\phi(\boldsymbol{x})$，**解码网络**为 $p_\theta(\boldsymbol{z})$. 还需注意，由于我们无法通过神经网络得到随机变量 $\boldsymbol{z}$ 的分布，所以我们只能假

设其服从正态分布 $\mathcal{N}(\boldsymbol{\mu}, \sigma^2 I)$，并将编码网络的输出设置为 $K+1$ 维，其中前 K 维表示 $\boldsymbol{z}$ 的均值 $\boldsymbol{\mu}$，最后一维表示方差 σ（或者 $\log\sigma$）.

为了能够优化上述参数 ϕ, θ，我们需要找到与之相关的损失函数，由于是概率参数优化，不难想到使用 MLE，于是仍然可以从变分方程 (5) 式出发，可以得到：

$$\log p(\boldsymbol{x}|\theta) = \underbrace{\mathrm{KL}(q_\phi(z)||p_\theta(z))}_{\text{KL 散度}} + \underbrace{\int q_\phi(z)\log\frac{p_\theta(x,z)}{q_\phi(z)}\,\mathrm{d}z}_{\text{变分下界}L}$$

由于KL散度恒大于零,所以第二项 L 就是 $\log p(\boldsymbol{x}|\theta)$ 的下界,如果要最大化似然 $\log p(\boldsymbol{x}|\theta)$,也就是最大化 L，于是

$$\begin{aligned}\max_{\phi,\theta} L &= \int q_\phi(z)\log\frac{p(z)p_\theta(x)}{q_\phi(z)}\,\mathrm{d}z = -\int q_\phi(z)\log\frac{q_\phi(z)}{p(z)} + \int q_\phi(z)\log p_\theta(x)\,\mathrm{d}z \\ &= -\mathrm{KL}(q_\phi(z)||p(z)) + \mathbb{E}_{q_\phi(z)}\left[\log p_\theta(x|z)\right]\end{aligned}$$

于是 MLE 等价于：最小化 $q_\phi(z)$ 与 $\mathcal{N}(0, I)$ 的 KL 散度，我们可以直接计算出两个正态分布的 KL 散度，将这部分称为**隐变量损失**：

$$L_{KL} := \mathrm{KL}(q_\phi(z)||p(z)) = \frac{1}{2}\sum_{i=1}^{B}(\sigma_i + \mu_i^2 - 1 - \log\sigma_i)$$

最大化第二项 $\mathbb{E}_{q_\phi(z)}\left[\log p_\theta(x|z)\right]$ 等价于最小化输入的数据与重建结果的误差，可以使用最小化 ℓ^1 或 ℓ^2 范数代替，这部分称为**重建损失** $L_{\boldsymbol{x}}(\boldsymbol{x}, \hat{\boldsymbol{x}})$，下面我们用 ℓ^1 范数表示输入与输出的误差，则

$$L_{\boldsymbol{x}}(\boldsymbol{x}, \hat{\boldsymbol{x}}) := ||\boldsymbol{x} - \hat{\boldsymbol{x}}||_1$$

综上，VAE 的损失函数为

$$L_{\text{VAE}} = c \cdot L_{KL} + L_{\boldsymbol{x}}(\boldsymbol{x}, \hat{\boldsymbol{x}}) \tag{6}$$

如果将 KL 散度理解为正则化项，则其中的 c 为 KL 正则化系数.

3.1.1 参数重采样技巧

VAE 需要使用“参数重采样技巧”（Reparameterization Trick）对隐变量进行采样，这是因为我们在使用梯度下降对参数进行优化时，在解码部分出现了随机变量 $\boldsymbol{z}$，即 $p_\theta(\boldsymbol{x}|\boldsymbol{z})$，这样会导致梯度下降无法进行，考虑用观测值对其进行估计；由于 $\boldsymbol{z} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x})$，于是我们可以通过对 $\boldsymbol{z}$ 的采样结果进行梯度计算.

在 **VAE 基本思想**中注释部分说明了编码网络可以得到 $\boldsymbol{z}$ 的分布参数 $\boldsymbol{\mu}, \sigma$，所以可以通过标准正态分布的采样 $\sigma \sim \mathcal{N}(0, I)$ 进行缩放平移得到 $\boldsymbol{z}$ 的采样：

$$\boldsymbol{z} = \boldsymbol{\mu} + \sigma\varepsilon = \boldsymbol{\mu} + \exp\left\{\frac{1}{2}\log(\sigma^2 I)\right\}\varepsilon$$

写成 $\log\sigma$ 的形式只是因为在实际预测网络中，直接预测标准差可能误差较大，所以对标准差做对数变换.

3.2 去偏自动编码机

去偏变分自动编码机（Debiasing Variational AutoEncoder, DB-VAE）在 VAE 基础上对训练数据的选取上做了一点改进，从而可以一定程度上避免数据集的偏差导致模型的偏差.

具体来讲：它通过自适应重采样（自动选择数据，进行重复性训练）减轻训练集中的潜在偏差. 例如：面部识别训练集中，大多数图片的人脸都是正面图像，而侧脸的图像偏少，如果将它们均等地训练，训练出的模型可能对正脸识别效果优于侧脸的效果，这就是数据偏差（Debiasing）. 为了平衡这种偏差有两种方法，一是使用人工处理，提高数据集中偏差数据的训练数量，但操作十分复杂，而且人无法判断哪些数据是偏差数据；二是通过机器自动识别偏差数据，然后自我调整数据的训练数量，这就是 DB-VAE 的提升之处.

DB-VAE 的示意图如图所示，图片来源论文Uncovering and Mitigating Algorithmic Bias through Learned Latent Structure.

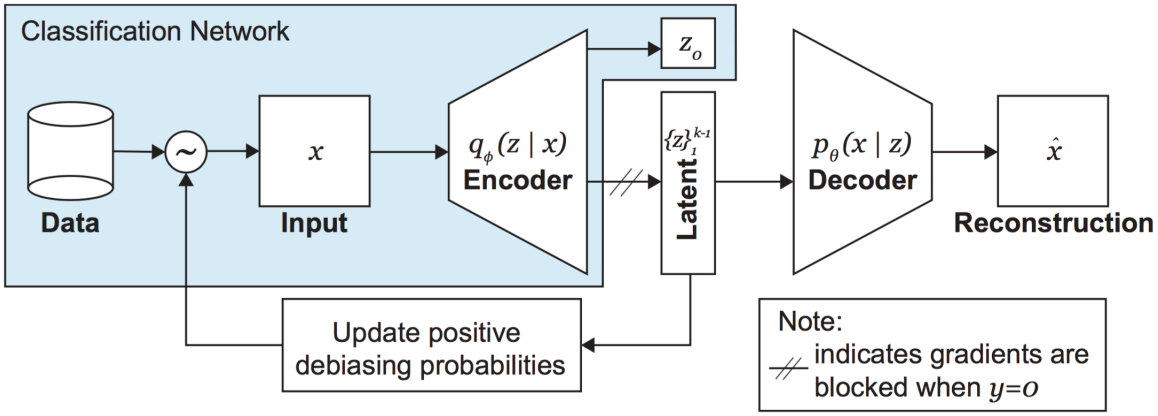

图 1: DB-VAE 原理图

并且 DB-VAE 在传统 VAE 的基础上加入了带有监督的变量 z_0，该变量可以用于判断做图像分类的任务. 需要注意由于数据集加入了分类任务，所以数据集中既有人脸图像也有非人脸图像，所以需要做的任务有以下三个：

- 仅对人脸相关的数据学习隐变量参数.
- 对人脸数据做去偏操作，避免模型偏向于某类数据.
- 做二分类问题，判断图像是否包含人脸.

在保证上面三点的条件下，我们可以基于 VAE 的损失函数 (6) 式，给出 DB-VAE 的损失函数：

$$L_{total} = \underbrace{\sum_{i\in\{0,1\}} y_i \log\left(\frac{1}{\hat{y}_i}\right)}_{L_y(y,\hat{y})} + \chi_{image}(y)\cdot L_{\text{VAE}}$$

其中 $L_y(y,\hat{y})$ 就是二分类问题的交叉熵损失函数，$\chi_{iamge}(y) = \begin{cases} 1, & \text{数据为人脸图片,} \\ 0, & \text{否则.} \end{cases}$

3.2.1 自适应重采样

我们可以利用图像隐变量的参数分布来增加对隐变量空间中代表性不足区域的采样，从而增加稀少数据的相对训练次数. 我们可以在训练网络前，先对每个数据预测得到对应的隐变量 $\boldsymbol{\mu}$ 的每一维频率直方图，从该直方图中我们可以得到每个数据对应的隐变量的出现频率占比，然后将出现频率取倒数（提高出现频率低的样本的重采样率），再归一化处理；对所有的数据在 K 维上的最大值采样概率作为该数据的重采样概率，并再次进行归一化，从而得到整个数据集的重采样概率分布.

从代码上可能更容易解释：

```python
### DB-VAE 在每个 epoch 开始前对数据集进行重采样 ###

# images 为图像数据集，dbvae 为编码网络，bins 为直方图中区间个数，smoothing_fac 平滑因子：用于降低重采样程度
def get_training_sample_probabilities(images, dbvae, bins=10, smoothing_fac=0.001):
    print("Recomputing the sampling probabilities")
    mu = get_latent_mu(images, dbvae)  # 获取每个 batch 对应潜变量的均值
    training_sample_p = np.zeros(mu.shape[0])  # 保存每个样本抽样概率分布
    for i in range(latent_dim):  # 考虑每一个潜变量分布
        latent_distribution = mu[:, i]

        # 生成一个潜变量分布的直方图
        hist_density, bin_edges = np.histogram(latent_distribution, density=True, bins=bins)

        # 获得每个潜变量分布的位置，np.digitize() 可以返回每个变量对应的分布位置
        bin_edges[0] = -float('inf')
        bin_edges[-1] = float('inf')
        bin_idx = np.digitize(latent_distribution, bin_edges)

        # 平滑密度函数
        hist_smoothed_density = hist_density + smoothing_fac
        hist_smoothed_density /= np.sum(hist_smoothed_density)

        # 反转密度函数
        p = 1 / (hist_smoothed_density[bin_idx-1])
        p /= np.sum(p)
        training_sample_p = np.maximum(p, training_sample_p)

    # 最后进行一次归一化操作
    training_sample_p /= np.sum(training_sample_p)
    return training_sample_p
```

4 实验步骤与结果分析

4.1 K-均值聚类

数据集下载：Kaggle - Old faithful. 数据集规模：$N = 272$，$D = 2$，N 行 D 列，使用 K-means 对其进行分类.

Algorithm 1 K-均值聚类

```
x = pd.read_csv("faithful.xls", index_col=0).to_numpy()
def normalize(x): return (x - np.mean(x)) / np.std(x)
x = np.apply_along_axis(normalize, 0, x)  # 按照列进行归一化
fig, axs = plt.subplots(2,3,figsize=(9,6))
def getdis(x, mu):
    dis = []
    for i in range(mu.shape[0]):
        dis.append(np.sqrt(np.sum(np.power(x-mu[i], 2))))
    return dis
mu = np.array([[-1, 1], [1, -1]])  # 初始化
for cnt, ax in enumerate(axs.reshape(-1)):
    # calculate the distance to each cluster
    dis = np.array([getdis(x[i], mu) for i in range(x.shape[0])])
    r = np.argmin(dis, axis=1)  # E 步
    plot(...)  # 绘制图像
    # M 步
    mu = np.array([np.mean(x[r==i], axis=0) for i in range(mu.shape[0])])
plt.show()
```

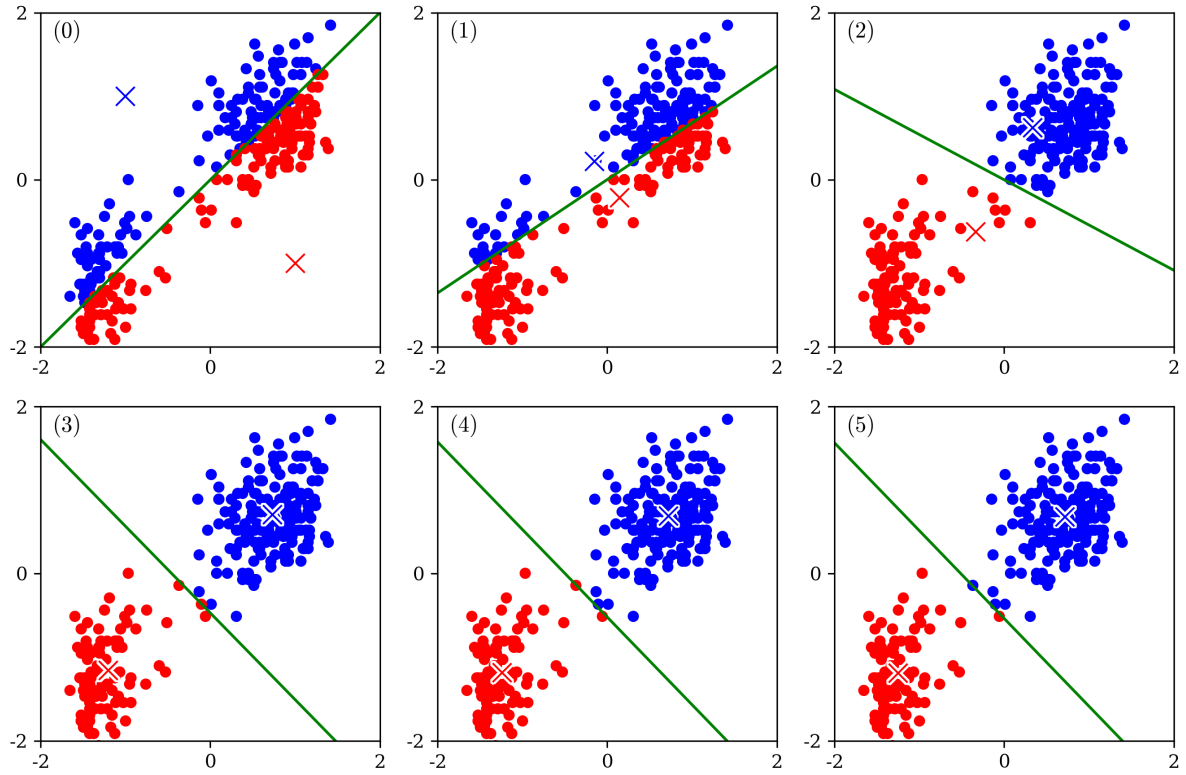

图 2: K-均值聚类

使用 K-均值做图像分割与图像压缩方法非常直接，假设图像是 $n \times m$ 的三通道像素，首先将图像拉直产生 $N = n \times m$ 个数据，每个数据的维数均为 $D = 3$，于是用 K-均值找到 K 个簇中心 $\boldsymbol{\mu}_k$，最后再用每个像素点从属的簇中心代替即可得到压缩后的图像. 这样我们只需存储原图像每个像素从属的簇编号，并记录下 $\boldsymbol{\mu}_k$，从而对图像大小进行压缩. 实现上使用的是 scikit-learn 库，因为图像较大，使用一般的线性算法速度太慢，在 scikit 中，K-均值使用了 KD 树进行加速，底层用 C++ 实现速度上有较大的提升.

Algorithm 2 K-均值聚类图像分割

```
from sklearn.cluster import KMeans
X = img.reshape(-1, 3)
fig, axs = plt.subplots(1, 4, figsize=(12, 4))
for k, ax in zip([2, 3, 10], axs):
    kmeans = KMeans(n_clusters=k).fit(X)  # 数据拟合
    pred = np.array([kmeans.cluster_centers_[i] for i in
    ↪  kmeans.labels_]).reshape(img.shape)  # 数据预测，转换为对应的聚类中心
    img_show(pred, ax, f"$K={k}$")
img_show(img, axs[-1], "Original Image")
plt.show()
```

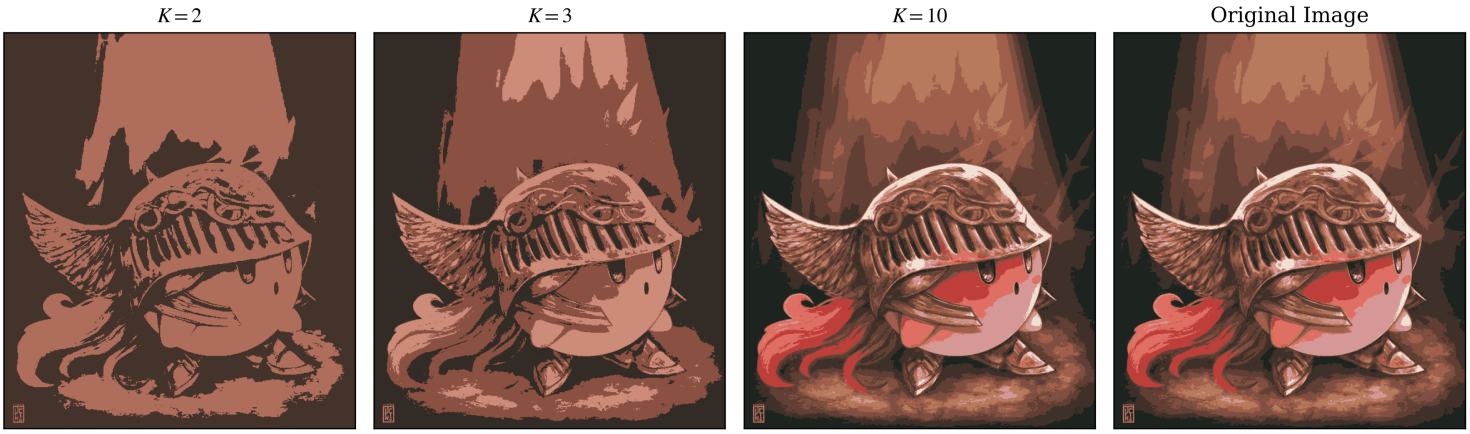

图 3: K-均值聚类图像分割

4.2 混合高斯模型 GMM

我对自己生成的数据使用了 K-均值和 GMM 进行聚类，并比较二者的区别. 数据生成方法：通过固定三个高斯分布，每个高斯下随机分布生成 1000 个数据

$$\begin{aligned}
&\mathcal{N}\left(\mu_1 = (1,1)^T, \Sigma = 0.3I + \varepsilon\right), \\
&\mathcal{N}\left(\mu_1 = (-1.5,0)^T, \Sigma = 0.2I + \varepsilon\right), \\
&\mathcal{N}\left(\mu_1 = (1,-1.5)^T, \Sigma = 0.1I + \varepsilon\right).
\end{aligned}$$

其中 ε 为 Gauss 噪声，即来自高斯分布的 2×2 随机数据作为方差的偏移量. 下面图4中展示了 K-均值的聚类效果，不难看到，最终聚类中心位置基本正确，但分类边界为硬分

类，无法很好的对边界进行处理. 而图5中展示了 GMM 的聚类效果，可以看出，由于 GMM 的软分类性质，所以可以很好得处理边界数据，但 GMM 算法对初值点的选取很重要，否则容易发生两个聚类中心重合的问题.

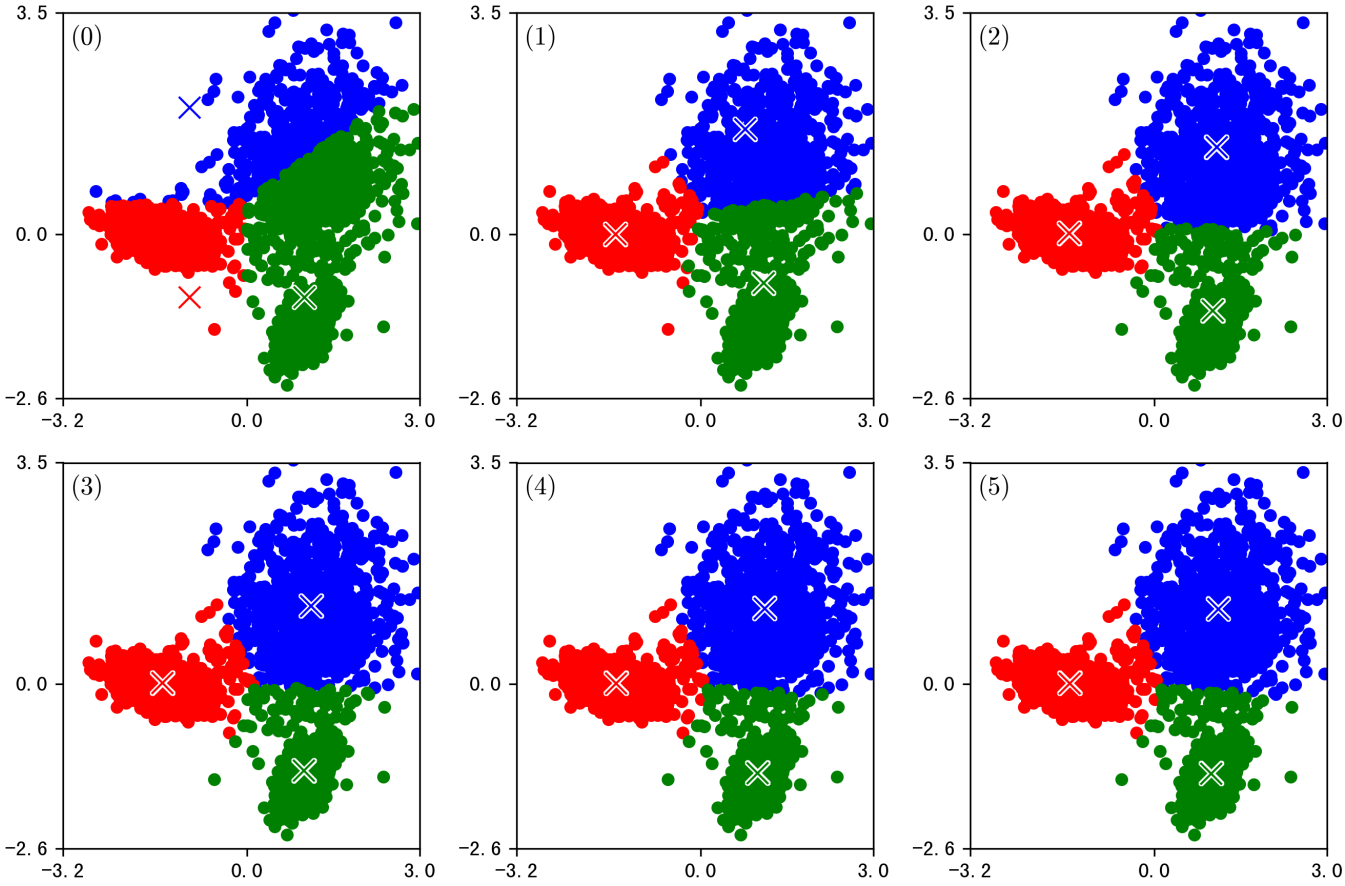

图 4: K-均值分类结果

Algorithm 3 GMM 算法

```
X = ...  # 初始化数据集
# 参数初始化
mu = np.array([[-1, 2], [-1, -1], [1, -1]])
sigma = np.array([0.1*np.eye(2) for _ in range(K)])
pi = np.full(3, 1/K)
def calc_normal(x, mu, sigma):  # 计算多维正态分布
    return np.power(2*np.pi, -K/2) * np.linalg.det(sigma) *
    ↪  np.exp(-0.5*np.dot(np.dot(x-mu, np.linalg.inv(sigma)), (x-mu).T))
for T in range(6):
    # E 步
    R = np.array([[calc_normal(X[i], mu[j], sigma[j]) for j in range(3)]
    ↪  for i in range(len(X))])
    plot(...)  # 绘制图像
    # M 步
    w = np.concatenate([R[:,j].reshape(-1, 1)/np.sum(R[:,j]) for j in
    ↪  range(K)], axis=1)
    pi = np.concatenate([R[:,j].reshape(-1, 1)/N for j in range(K)],
    ↪  axis=1)
```

```
15      sigma = np.array([np.sum([w[i,k] * np.dot((X[i]-mu[k]).reshape(-1,1),
        ↪  (X[i]-mu[k]).reshape(1,-1)) for i in range(N)], axis=0) for k in
        ↪  range(K)])
16      mu = np.array([[np.dot(X[:,i], w[:,k]) for i in range(2)] for k in
        ↪  range(K)])
17  plt.show()
```

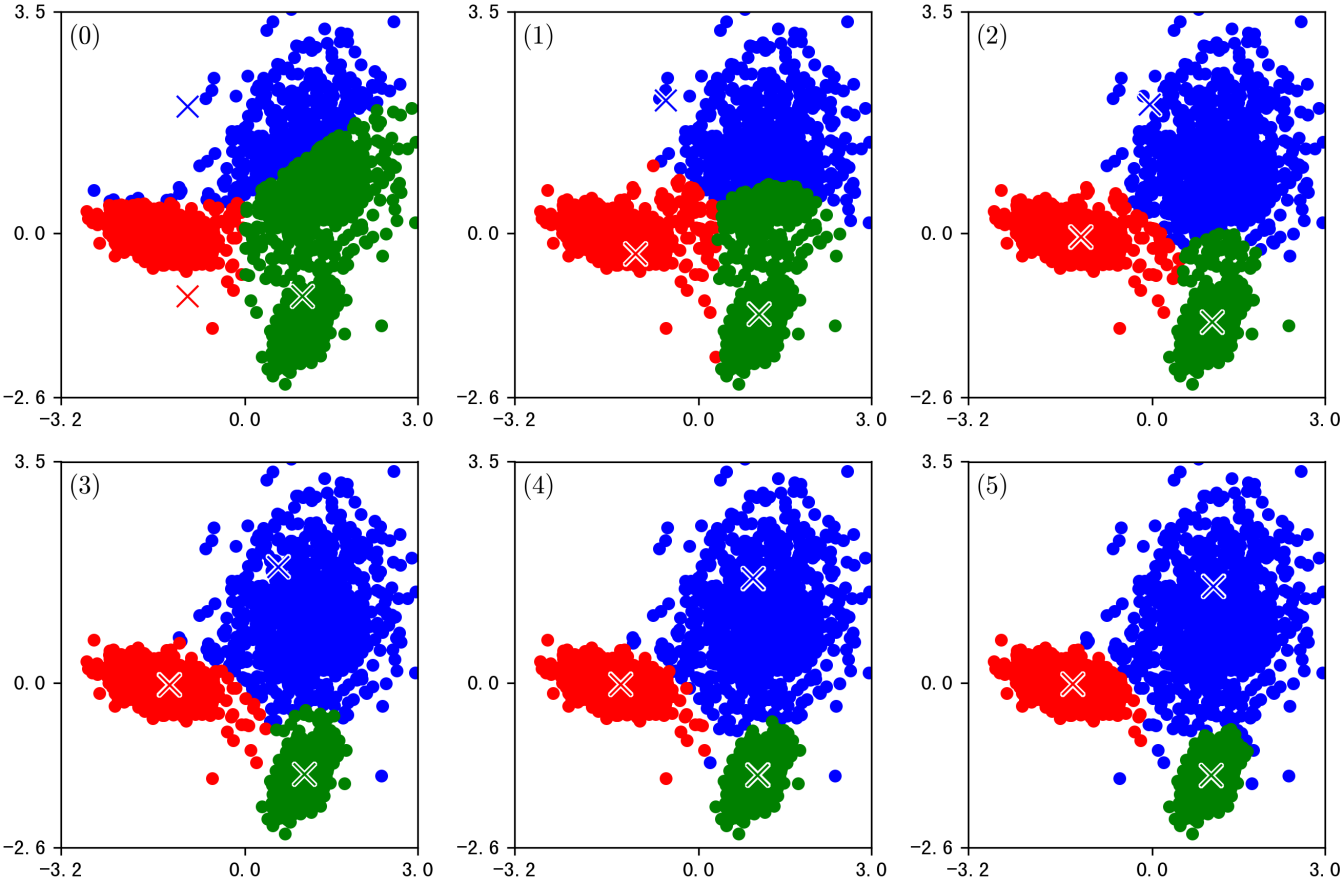

图 5: GMM 聚类结果

4.3 DB-VAE

代码参考MIT 公开课 6.S191 Lib2内容，该去偏算法也是由该课程主讲人给出的. 任务为人脸图像的分类问题，并根据图像的隐参数进行重建，我们使用了两个数据集：

1. 正训练集：CelebA Dataset，包含超过二十万张名人照片.
2. 负训练集：ImageNet，该网站上有非常多不同分类的图片，我们将从非人脸类别中选取负样本. 通过 Fitzpatrick 度量法 对肤色进行分类，将图片标记为”Lighter”或”Darker”.

然后我们使用了经典 CNN 和 DE-VAE 神经网络对图片进行识别，图6对两种方法的预测结果进行了比较，

图7展示了 VAE 的图像渐变转化功能（变脸效果），清晰的图像为输入的图片（左右两端），较为模糊的图像为 VAE 输出的重建结果.

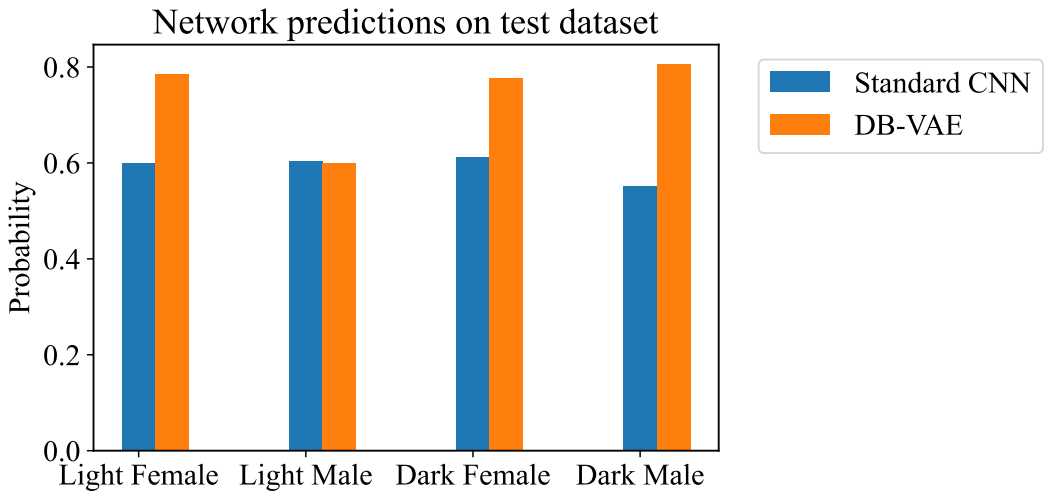

图 6: CNN 与 VAE 算法在带偏差的数据下分类效果比对

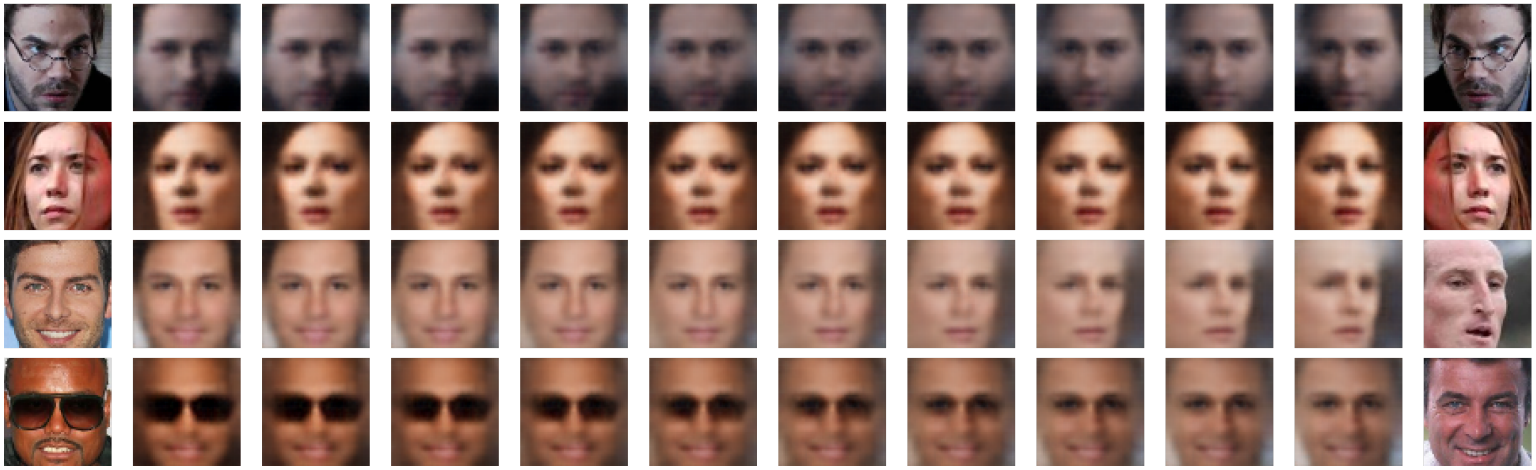

图 7: 图像渐变效果

下面代码 4-9 给出了 DB-VAE 的代码实现，主要分为模型搭建（编码网络，解码网络），损失函数（隐参数损失，重建损失，二分类交叉熵损失）及网络训练.

Algorithm 4 DB-VAE 模型

```
### VAE 参数重采样 ###

def sampling(z_mean, z_logsigma):
    batch, latent_dim = z_mean.shape
    epsilon = tf.random.normal(shape=(batch, latent_dim))
    z = z_mean + tf.math.exp(0.5 * z_logsigma) * epsilon
    return z

### 创建 DB-VAE 模型 ###

class DB_VAE(keras.Model):
    def __init__(self, latent_dim):
        super().__init__()
        self.latent_dim = latent_dim
        # 编码器输出大小，包含隐参数估计 mu,logsigma 和分类结果 y
        num_encoder_dims = 2*self.latent_dim + 1
```

```
        self.encoder = make_CNN_classifier(num_encoder_dims)
        self.decoder = make_face_decoder_network()

    def encode(self, x):  # 输入图像到编码器中，返回潜空间的参数和二分类的概率
        encoder_ouput = self.encoder(x)
        y_logit = tf.expand_dims(encoder_ouput[:, 0], -1)  # 取输出向量中第 0 维作为分类概率
        # 潜变量的均值和方差
        z_mean = encoder_ouput[:, 1:self.latent_dim+1]
        z_logsigma =  encoder_ouput[:, self.latent_dim+1:]
        return y_logit, z_mean, z_logsigma

    def reparameterize(self, z_mean, z_logsigma):  # VAE 重新参数化，从正态分布中取样
        return sampling(z_mean, z_logsigma)

    def decode(self, z):
        return self.decoder(z)

    def call(self, x):  # 定义该模型的计算过程
        y_logit, z_mean, z_logsigma = self.encode(x)
        z = self.reparameterize(z_mean, z_logsigma)
        recon = self.decode(z)
        return y_logit, z_mean, z_logsigma, recon

    def predict(self, x):  # 预测给定的输入 x 是否是人脸图片
        y_logit, z_mean, z_logsigma = self.encode(x)
        return y_logit

dbvae = DB_VAE(latent_dim)
```

Algorithm 5 编码部分-卷积网络

```
n_filters = 12 # 每个卷积层中卷积核的个数

def make_CNN_classifier(n_output=1):
    Conv2D = functools.partial(layers.Conv2D, padding='same', activation='relu')  # padding='same'表示进行零填充

    return keras.Sequential([
        Conv2D(filters=1*n_filters, kernel_size=5, strides=2, input_shape=(64, 64, 3)),  # strides 为卷积的步长
        layers.BatchNormalization(),
        Conv2D(filters=2*n_filters, kernel_size=5, strides=2),
        layers.BatchNormalization(),
        Conv2D(filters=4*n_filters, kernel_size=3, strides=2),  #  降低卷积核的大小，以提取更多的信息
        layers.BatchNormalization(),
        Conv2D(filters=6*n_filters, kernel_size=3, strides=2),
        layers.BatchNormalization(),
        layers.Flatten(),
```

```
        layers.Dense(512, activation='relu'),
        layers.Dense(n_output)
    ])
```

Algorithm 6 解码部分-转置卷积网络

```
n_filters = 12  # 卷积核个数，和编码部分 CNN 相同
latent_dim = 100  # 隐变量 mu 维度

def make_face_decoder_network():
    Conv2DTranspose = functools.partial(layers.Conv2DTranspose,
    ↪  padding='same', activation='relu')
    return keras.Sequential([
        # 和上文中的 CNN 建图过程部分正好相反
        layers.Dense(units=4*4*6*n_filters),  # 上文卷积最后得到图像大小正好是
        ↪  4*4，层数为 6N
        layers.Reshape(target_shape=(4, 4, 6*n_filters)),
        Conv2DTranspose(filters=4*n_filters, kernel_size=3, strides=2),
        Conv2DTranspose(filters=2*n_filters, kernel_size=3, strides=2),
        Conv2DTranspose(filters=1*n_filters, kernel_size=5, strides=2),
        Conv2DTranspose(filters=3, kernel_size=5, strides=2)
    ])
```

Algorithm 7 VAE 损失函数 $L_{\text{VAE}} = c \cdot \frac{1}{2} \sum_{j=0}^{k-1} (\sigma_j + \mu_j^2 - 1 - \log \sigma_j) + ||x - \hat{x}||_1$

```
def vae_loss_func(x, x_recon, mu, logsigma, kl_weight=0.0005):
    """
        x: 输入特征，
        x_recon: 重建输出，
        mu: 编码均值，
        logsigma: 标准差取 log 的结果，
        kl_weight: 隐参数损失的权系数（正则化系数）
    """
    latent_loss = 0.5 * tf.reduce_sum(tf.exp(logsigma) + tf.square(mu) -
    ↪  1.0 - logsigma, axis=1)
    reconstruction_loss = tf.reduce_mean(tf.abs(x - x_recon), axis=(1,2,3))
    vae_loss = kl_weight * latent_loss + reconstruction_loss
    return vae_loss
```

Algorithm 8 DB-VAE 损失函数 $L_{total} = L_y(y, \hat{y}) + \chi_{image}(y) \cdot L_{VAE}$

```
def debiasing_loss_func(x, x_pred, y, y_logit, mu, logsigma):
    vae_loss = vae_loss_func(x, x_pred, mu, logsigma)
    classification_loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=y,
    ↪  logits=y_logit)
    face_indicator = tf.cast(tf.equal(y, 1), tf.float32)
    total_loss = tf.reduce_mean(
        classification_loss +
```

```
        face_indicator * vae_loss
    )
    return total_loss, classification_loss
```

Algorithm 9 DB-VAE 训练

```
### Training the DB-VAE ###

# 超参数配置
batch_size = 32
learning_rate = 5e-4
latent_dim = 100
num_epochs = 15

dbvae = DB_VAE(latent_dim)
optimizer = keras.optimizers.Adam(learning_rate)

# 使用 @tf.function 将 Python 中的函数，转化为 TensorFlow 中的计算图，用于计算偏导数
@tf.function
def debiasing_train_step(x, y):
    with tf.GradientTape() as tape:  # 自动求导
        y_logit, z_mean, z_logsigma, x_recon = dbvae(x)
        loss, class_loss = debiasing_loss_func(x, x_recon, y, y_logit, z_mean, z_logsigma)
    grads = tape.gradient(loss, dbvae.trainable_variables)
    optimizer.apply_gradients(zip(grads, dbvae.trainable_variables))
    return loss

all_faces = loader.get_all_train_faces()

for i in range(num_epochs):
    # 重新计算重采样的概率
    p_faces = get_training_sample_probabilities(all_faces, dbvae)

    # 获取训练数据集，开始训练
    for j in tqdm(range(loader.get_train_size() // batch_size)):
        (x, y) = loader.get_batch(batch_size, p_pos=p_faces)
        loss = debiasing_train_step(x, y)
```

5 参考文献

1. 孟德宇. 西安交通大学《机器学习》课程 PPT, 2023.
2. Christopher, M. Bishop. Pattern Recognition and Machine Learning, Springer, 2006. 9. Mixture Models and EM.
3. Auto-Encoding Variational Bayes, Diederik P Kingma and Max Welling, 2013. 9.
4. Alexander Amini, Ava P. Soleimany, et al. Uncovering and Mitigating Algorithmic Bias through Learned Latent Structure, 2019. 1.
5. MIT 6.S191 Lib2, 2022.