《数据科学与工程算法基础》实践报告

报告题目：基于主成分分析的图像压缩及其变体

姓名：龚敬洋

学号：10195501436

完成日期：2021/11/24

摘要

图像压缩一直是图像处理中一个重要的任务，一个好的图像压缩算法可以大大降低存储和传输代价。主成分分析（Principal Component Analysis）作为一个经典的降维方法，已经在图像压缩领域得到了极为广泛的运用。在本文中，我们从主成分分析的原理出发，导出并实现了基于特征分解的朴素 PCA 算法，并使用奇异值分解方法对其计算进行了优化。随后，我们实现了更适合图像处理的 2DPCA、其改进方法 2D-2DPCA。此外，我们还实现了基于核方法的 Kernel PCA 来进一步提升主成分的表达能力，并对比了多种不同的核函数下图像的压缩效果。进一步的，我们尝试使用了较为现代的基于神经网络的 GHA（Generalized Hebbian Algorithm）算法来迭代得到主成分。最后，我们将实现的结果与时下较为常用的基于离散余弦变换的 JPEG 图像压缩算法进行了对比。

关键字： 图像压缩，主成分分析，核方法，感知机，余弦变换

Abstract

Image compression has long be a fundamental task in image processing, a good image compression algorithm can massively reduce the cost of storage and transmission. As a classic dimension reduction algorithm, Principle Component Analysis has been widely used in the field of image compression. In this article, we derive and implement the naive PCA algorithm based on eigenvalue decomposition, and use the singular value decomposition method to optimize its calculation. Subsequently, we implemented 2DPCA and its improvement version 2D-2DPCA, which is more suitable for image processing. In addition, we implemented Kernel PCA based on the kernel method to further improve the expression ability of principal components, and compared the compression effects of images under a variety of different kernel functions. Further, we tried to use a more modern neural network-based GHA (Generalized Hebbian Algorithm) algorithm to iteratively obtain the principal components. Finally, we compared the achieved results with the contemporary mainstream JPEG image compression algorithm which is based on discrete cosine transform.

Keywords: Image compression, Principle component analysis, Perceptron, Cosine transformation

1 项目概述

主成分分析（Principal Component Analysis）可以用来减少矩阵（图像）的维度，并将这些新的维度投射到图像上，使其保留质量。本项目要求我们使用 PCA 方法及其变体，对 3 组图像（每组包含 100 张图像）进行压缩，并对图像压缩的性能进行分析。

2 问题描述

一张 8 位三通道（RGB）正方形彩色图片可视为三个 N 维矩阵 $\boldsymbol{X}$，其中 $x_{ij} \in \{0, 1, \cdots, 255\}$，其存储代价为 b。图像压缩的目标即为寻找一个映射 $\mathcal{Q}$，使得 $\mathcal{Q}(\boldsymbol{X}) \in \mathbb{R}^{k\times n}(k \ll n), \mathcal{Q}^{-1}(\mathcal{Q}(\boldsymbol{X})) \approx X$，且存储 $\mathcal{Q}(\boldsymbol{X})$ 和 $\mathcal{Q}^{-1}$ 所需的空间 $\tilde{b} \ll b$。

3 方法

3.1 朴素 PCA

PCA 的主要思想是通过将一个高维样本 $\boldsymbol{x} \in \mathbb{R}^n$ 左乘一个正交矩阵 $\boldsymbol{Q} \in \mathbb{R}^{k\times n}(k \ll n)$，使得其映射到一个较低维的超平面 $\boldsymbol{Qx} \in \mathbb{R}^k$ 上，同时又保证多个数据点映射后的统计性质保持不变。具体来说，这样的超平面要具有如下的性质：[周 16]

- **最近重构性**：样本点到这个超平面的距离足够近
- **最大可分性**：样本点在这个超平面上的投影尽可能分开

由此，我们就有两种角度来求解这一正交矩阵。事实上，在中心化条件下，这两者是等价的。这是由于有如下定理保证：

Theorem 3.1.1. 对于中心化数据集 $\{\boldsymbol{x}^{(i)}\}_{i=1}^N$，最小化重构距离等价于最大化投影方差 *[Gor18]*

Proof. 这里仅证明投影到一维时的情形，高维时的情况可自然推广

设投影直线的方向向量为 $\boldsymbol{v}$，其中 $||\boldsymbol{v}||^2 = 1$

则由勾股定理可知，$||\boldsymbol{x}^{(i)} - \boldsymbol{v^T x}^{(i)}\boldsymbol{v}||^2 = ||\boldsymbol{x}^{(i)}||^2 - \left(\boldsymbol{v}^T\boldsymbol{x}^{(i)}\right)^2$

于是 $\boldsymbol{v}$ 的最优解

$$\begin{aligned}
\boldsymbol{v}^* &= \arg\min_{\boldsymbol{v}:||\boldsymbol{v}||^2=1} \frac{1}{N}\sum_{i=1}^N ||\boldsymbol{x}^{(i)} - \boldsymbol{v^T x}^{(i)}\boldsymbol{v}||^2 \\
&= \arg\min_{\boldsymbol{v}:||\boldsymbol{v}||^2=1} \frac{1}{N}\sum_{i=1}^N \left(||\boldsymbol{x}^{(i)}||^2 - \left(\boldsymbol{v}^T\boldsymbol{x}^{(i)}\right)^2\right) \\
&= \arg\max_{\boldsymbol{v}:||\boldsymbol{v}||^2=1} \frac{1}{N}\sum_{i=1}^N \left(\boldsymbol{v}^T\boldsymbol{x}^{(i)}\right)^2
\end{aligned}$$

也即最小化重构距离与最大化投影方差等价 □

这里我们通过最大化投影方差的方法来求解。我们知道，对于一个中心化矩阵 $\boldsymbol{X}$（即 $E(\boldsymbol{X}) =$

$\mathbf{0}$），其协方差矩阵

$$\begin{aligned}\Sigma(\boldsymbol{X}) &= E\left[(\boldsymbol{X}-E(\boldsymbol{X}))(\boldsymbol{X}-E(\boldsymbol{X}))^T\right] \\ &= \begin{pmatrix} \mathrm{Cov}(\boldsymbol{X}_1,\boldsymbol{X}_1) & \mathrm{Cov}(\boldsymbol{X}_1,\boldsymbol{X}_2) & \cdots & \mathrm{Cov}(\boldsymbol{X}_1,\boldsymbol{X}_n) \\ \mathrm{Cov}(\boldsymbol{X}_2,\boldsymbol{X}_1) & \mathrm{Cov}(\boldsymbol{X}_2,\boldsymbol{X}_2) & \cdots & \mathrm{Cov}(\boldsymbol{X}_2,\boldsymbol{X}_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}(\boldsymbol{X}_n,\boldsymbol{X}_1) & \mathrm{Cov}(\boldsymbol{X}_n,\boldsymbol{X}_2) & \cdots & \mathrm{Cov}(\boldsymbol{X}_n,\boldsymbol{X}_n) \end{pmatrix} \\ &= \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T\end{aligned}$$

若将投影矩阵 $\boldsymbol{Q}$ 按行划分为向量组，也即设

$$\boldsymbol{Q} = \begin{pmatrix} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \vdots \\ \boldsymbol{q}_k \end{pmatrix}$$

，则要使得投影方差和最大，也即求解

$$\begin{aligned}\underset{\boldsymbol{Q}:\boldsymbol{Q}\boldsymbol{Q}^T=\boldsymbol{I}}{\arg\max} \sum_{i=1}^{k} ||\boldsymbol{q}_i\boldsymbol{X}||_2^2 &= \underset{\boldsymbol{Q}:\boldsymbol{Q}\boldsymbol{Q}^T=\boldsymbol{I}}{\arg\max} ||\boldsymbol{Q}\boldsymbol{X}||_F^2 \\ &= \underset{\boldsymbol{Q}:\boldsymbol{Q}\boldsymbol{Q}^T=\boldsymbol{I}}{\arg\max} \mathrm{tr}\left(\boldsymbol{X}^T\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{X}\right)\end{aligned}$$

由此可得优化问题

$$\begin{aligned}\min \quad & -\mathrm{tr}\left(\boldsymbol{X}^T\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{X}\right) \\ \mathbf{s.t} \quad & \boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I}_{k\times k}\end{aligned}$$

利用拉格朗日乘子法，我们可得当目标函数取到最小值时，有 [Woo09]

$$\boldsymbol{Q}\boldsymbol{X}\boldsymbol{X}^T = \boldsymbol{Q}\boldsymbol{\lambda}$$

也即 $\boldsymbol{Q}$ 中的第 i 行为 $\boldsymbol{X}\boldsymbol{X}^T$ 的第 i 个特征值 λ_i 对应的特征向量。进一步的，将结果代回原式我们可以发现，由于 $\boldsymbol{Q}$ 为一个 $k\times n$ 的矩阵，因此若要使得目标函数取到最小值，$\boldsymbol{Q}$ 中的行向量应取前 k 大的特征值所对应的特征向量。

在图像压缩任务中，当将数据集映射到低维空间后，我们还需要将其重构回原来的图像空间以保证图像的可用性。对于使用正交变换的 PCA 方法，这一重构任务是容易的。由于 $\boldsymbol{Q}$ 为一正交矩阵，其逆矩阵 $\boldsymbol{Q}^{-1} = \boldsymbol{Q}^T$。因此要重构压缩后的图像，进行我们只需要对降维数据进行逆变换，即左乘 $\boldsymbol{Q}^T$ 即可。

由此，我们导出了使用朴素 PCA 方法进行图像压缩的一般过程。需要注意的是，要使用基于特征分解的 PCA 方法对图像进行分析和处理，我们需要将图像矩阵划分为向量组并进行中心化操作，这里我们采用按列划分的方法。算法的具体流程如下：（Algorithm 1）

Algorithm 1 基于特征分解的 PCA 图像压缩算法

中心化图像点 $\tilde{x}_{ij} = x_{ij} - \bar{x}_i$

$\boldsymbol{C} = \tilde{\boldsymbol{X}}\tilde{\boldsymbol{X}}^T$

求解 $\boldsymbol{C}$ 的特征值及对应的特征向量 $(\lambda_i, \boldsymbol{\alpha}_i)$

按特征值从大到小将前 k 个特征向量按行排列成变换矩阵 $\boldsymbol{Q}_k$

对中心化图像点进行变换并作逆变换 $\tilde{\boldsymbol{Y}} = \boldsymbol{Q}_k^T\boldsymbol{Q}_k\tilde{\boldsymbol{X}}$

还原图像 $y_{ij} = \tilde{y}_{ij} + \bar{x}_i$

我们使用 Python 实现了该算法，实现代码见附录。

在选择了不同主成分个数时，使用朴素 PCA 算法进行图像压缩的效果结果如下图所示（Figure 1）。可以看到，仅使用前 10 个主成分已经能还原出整体的图像轮廓，当 $k = 50$ 时，图像的细节已基本得到恢复。

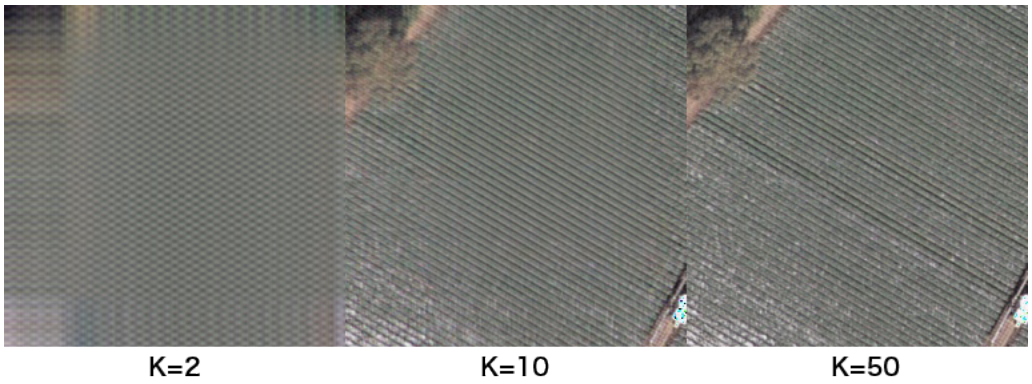

Figure 1: 不同主成分个数下使用 PCA 进行图像压缩的重构结果

3.2 基于奇异值分解的 PCA

可以看到，基于特征值分解的 PCA 算法中计算开销最大的部分为计算协方差矩阵 $\boldsymbol{X}\boldsymbol{X}^T$ 的特征值与特征向量。事实上，我们可以使用奇异值分解来避免这一高开销计算，

一个 $m \times n$ 的矩阵 $\boldsymbol{A}$ 的奇异值分解是指将其分解为三个特殊矩阵乘积的形式 $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$，其中 $\boldsymbol{U}$ 为 m 阶正交矩阵，$\boldsymbol{V}$ 为 n 阶正交矩阵，Σ 是由降序排列的非负的对角线元素组成的 $m \times n$ 对角矩阵。

对于任意实矩阵，我们都能找到它的奇异值分解。这是由于有如下定理保证：[Wik21b]

Theorem 3.2.1. 若 $\boldsymbol{A}$ 为一 $m \times n$ 实矩阵，$\boldsymbol{A} \in \mathbb{R}^{m\times n}$，则 $\boldsymbol{A}$ 的奇异值分解存在 $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ 其中 $\boldsymbol{U}$ 是 m 阶正交矩阵，$\boldsymbol{V}$ 是 n 阶正交矩阵，$\boldsymbol{\Sigma}$ 是 $m \times n$ 对角矩阵，其前 r 个对角元素 $(\sigma_1, \cdots, \sigma_r)$ 为正，且按降序排列，其余均为 0。

Proof. 由于 $\boldsymbol{A}^T\boldsymbol{A}$ 为对称半正定矩阵，因此可以对其进行特征分解 $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}_n\boldsymbol{V}^T$，其中 $V \in \mathbb{R}^{n\times n}$ 是正交矩阵，$\boldsymbol{\Lambda}_n$ 是对称矩阵，并且对角线元素是 $\boldsymbol{A}^T\boldsymbol{A}$ 的特征值 $\lambda_i \geq 0, i = 1, \cdots, n$，并且是按降序排列的。因为 $(\boldsymbol{A}) = (\boldsymbol{A}^T\boldsymbol{A}) = r$，所以前 r 个特征值是正的。

注意到 $\boldsymbol{A}\boldsymbol{A}^T$ 和 $\boldsymbol{A}^T\boldsymbol{A}$ 有相同的非零特征值，因此他们的秩是相等的。我们定义

$$\sigma_i = \sqrt{\lambda_i} > 0, i = 1, \cdots, r$$

，记 $\boldsymbol{v}_1,\cdots,\boldsymbol{v}_r$ 是 $\boldsymbol{V}$ 的前 r 列，它们同时也是 $\boldsymbol{A}^T\boldsymbol{A}$ 前 r 个特征值对应的特征向量。即有

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{v}_i=\lambda_i\boldsymbol{v}_i,i=1,\cdots,r$$

。因此同时在两边左乘上 $\boldsymbol{A}$ 就有

$$(\boldsymbol{A}\boldsymbol{A}^T)\boldsymbol{A}\boldsymbol{v}_i=\lambda_i\boldsymbol{A}\boldsymbol{v}_i,i=1,\cdots,r$$

。这就意味着 $\boldsymbol{A}\boldsymbol{v}_i$ 是 $\boldsymbol{A}\boldsymbol{A}^T$ 的特征向量，因为 $\boldsymbol{v}_i^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{v}_j=\lambda_j\boldsymbol{v}_i^T\boldsymbol{v}_j$ 所以这些特征向量也是正交的。所以将他们标准化则有

$$\boldsymbol{u}_i=\frac{\boldsymbol{A}\boldsymbol{v}_i}{\sqrt{\lambda_i}}=\frac{\boldsymbol{A}\boldsymbol{v}_i}{\sigma_i},i=1,\cdots,r$$

这些 $\boldsymbol{u}_1,\cdots,\boldsymbol{u}_r$ 是 r 个 $\boldsymbol{A}\boldsymbol{A}^T$ 关于非零特征值 $\lambda_1,\cdots,\lambda_r$ 的特征向量。因此

$$\boldsymbol{u}_i^T\boldsymbol{A}\boldsymbol{v}_j=\frac{1}{\sigma_i}\boldsymbol{v}_i^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{v}_j=\frac{\lambda_j}{\sigma_i}\boldsymbol{v}_i^T\boldsymbol{v}_j=\begin{cases}\sigma_i,i=j\\0,\end{cases}$$

以矩阵的方式重写即有

$$\begin{pmatrix}\boldsymbol{u}_1^T\\\vdots\\\boldsymbol{u}_r^T\end{pmatrix}\boldsymbol{A}(\boldsymbol{v}_1,\cdots,\boldsymbol{v}_r)=\mathrm{diag}(\sigma_1,\cdots,\sigma_r)=\boldsymbol{\Sigma}_r$$

注意到根据定义

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{v}_i=0,i=r+1,\cdots,n$$

即有

$$\boldsymbol{A}\boldsymbol{v}_i=0,i=r+1,\cdots,n$$

取相互正交的单位向量 $\boldsymbol{u}_{r+1},\cdots,\boldsymbol{u}_m$ 均与 $\boldsymbol{u}_1,\cdots,\boldsymbol{u}_r$ 正交，即有

$$\boldsymbol{u}_i^T\boldsymbol{A}\boldsymbol{v}_j=0.i=1,\cdots,m;j=r+1,\cdots,n$$

它们共同构成了 $\mathbb{R}^m$ 的一组标准正交基。因此，扩展前述奇异值分解式即有

$$\begin{pmatrix}\boldsymbol{u}_1^T\\\vdots\\\boldsymbol{u}_m^T\end{pmatrix}\boldsymbol{A}(\boldsymbol{v}_1,\cdots,\boldsymbol{v}_n)=\begin{pmatrix}\boldsymbol{\Sigma}_r&\boldsymbol{0}^T\\\boldsymbol{0}&\boldsymbol{O}\end{pmatrix}=\boldsymbol{\Sigma}$$

令 $\boldsymbol{U}=(\boldsymbol{u}_1,\cdots,\boldsymbol{u}_m)$，$\boldsymbol{V}=(\boldsymbol{v}_1,\cdots,\boldsymbol{v}_n)$，即有 $\boldsymbol{A}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$。由此可知，矩阵 $\boldsymbol{A}$ 必存在奇异值分解。 □

根据矩阵的奇异值分解定理，对于中心化图像矩阵 $\boldsymbol{X}$，我们有

$$\begin{aligned}\boldsymbol{X}^T\boldsymbol{X}&=\left(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\right)^T\left(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\right)\\&=\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{V}^T\end{aligned}$$

也即 $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{V}=\boldsymbol{V}\boldsymbol{\Sigma}$。由此我们得知协方差矩阵的第 i 个特征向量也即右奇异值矩阵 $\boldsymbol{V}$ 的第 i 列。

基于奇异值分解的 PCA 图像压缩算法具体流程如下：（Algorithm 2）

Algorithm 2 基于奇异值分解的 PCA 图像压缩算法

中心化图像点 $\tilde{x}_{ij} = x_{ij} - \bar{x}_i$

对 $\tilde{\boldsymbol{X}}$ 进行奇异值分解 $\tilde{\boldsymbol{X}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$

取 $\boldsymbol{V}$ 的前 k 列得到 $\boldsymbol{V}_k$

对中心化图像点进行变换并作逆变换 $\tilde{\boldsymbol{Y}} = \tilde{\boldsymbol{X}}\boldsymbol{V}_k\boldsymbol{V}_k^T$

还原图像 $y_{ij} = \tilde{y}_{ij} + \bar{x}_i$

同样的，我们使用 Python 实现了上述算法。对于奇异值分解操作，我们使用 Numpy 模块中提供的 **svd()** 函数来完成。Numpy 模块使用了 LAPACK 科学计算包来完成矩阵分解的相关操作。其中，SVD 分解采用了 **Householder 变换**的方式来完成 [Leh94]，这一操作的时间开销远低于一般的采用 **Gram-Schimdt 正交化**的求解方式，也远低于对协方差矩阵作特征值分解的时间开销。

3.3 2DPCA

由于常规的 PCA 方法是对向量组进行操作，因此当我们使用该方法对图像进行压缩时，需要先将其划分为列向量组再进行处理。由此得到的协方差矩阵规模十分巨大，需要极大的时间开销来完成计算。此外，由于图像压缩通常被作为其他图像处理任务的上游任务，如此操作会导致图像的特征信息出现大量的丢失。因此，之后的研究者提出了另一种简单的图像投影技术，称为**二维主成分分析 (2DPCA)** [YZFyY04]，专门用于图像特征提取。与传统的 PCA 方法不同，2DPCA 基于 2D 矩阵直接构建图像的协方差矩阵。与 PCA 的协方差矩阵相比，使用 2DPCA 的图像协方差矩阵的大小要小得多，这就意味着确定相应的特征向量所需的时间更少。此外，由于其更多的利用了图像的空间信息，对图像的特征也能够更好的保留。

2DPCA 的主要思想是直接利用一个 $n \times k$ 维矩阵对整张图像进行投影。这里我们先考虑投影矩阵为一维时的情况，再将其推广到 k 维上去。

若设图像矩阵为 $\boldsymbol{A}$，$\boldsymbol{X}$ 为投影向量，$\boldsymbol{Y} = \boldsymbol{A}\boldsymbol{X}$ 为投影后的特征。则 $\boldsymbol{Y}$ 的协方差矩阵 $\boldsymbol{S}_x$ 可以写为

$$\begin{aligned}\boldsymbol{S}_x &= E\left[(\boldsymbol{Y} - E[\boldsymbol{Y}])(\boldsymbol{Y} - E[\boldsymbol{Y}])^T\right] \\ &= E\left[(\boldsymbol{A}\boldsymbol{X} - E[\boldsymbol{A}\boldsymbol{X}])(\boldsymbol{A}\boldsymbol{X} - E[\boldsymbol{A}\boldsymbol{X}])^T\right] \\ &= E\left[((\boldsymbol{A} - E[\boldsymbol{A}])\boldsymbol{X})((\boldsymbol{A} - E[\boldsymbol{A}])\boldsymbol{X})^T\right]\end{aligned}$$

与一般的 PCA 方法类似，我们定义判断投影好坏的评价指标为 $J(\boldsymbol{X}) = \mathrm{tr}(\boldsymbol{S}_x)$，则代入上面的式子就可以写为

$$\begin{aligned}J(\boldsymbol{X}) &= \mathrm{tr}(\boldsymbol{S}_x) \\ &= \boldsymbol{X}^T E\left[(\boldsymbol{A} - E[\boldsymbol{A}])^T(\boldsymbol{A} - E[\boldsymbol{A}])\right]\boldsymbol{X}\end{aligned}$$

。从上面的式子我们可以看出，2DPCA 通常是同时作用于多张图像矩阵上的，这也是其协方差矩阵维度相对较小的原因。若设图像集合为 $\boldsymbol{A}_1, \cdots, \boldsymbol{A}_M$，则

$$\begin{aligned}\boldsymbol{G} &\overset{def}{=} E\left[(\boldsymbol{A} - E[\boldsymbol{A}])^T(\boldsymbol{A} - E[\boldsymbol{A}])\right] \\ &= \frac{1}{M}\sum_{i=1}^{M}\left(\boldsymbol{A}_i - \bar{\boldsymbol{A}}\right)^T\left(\boldsymbol{A}_i - \bar{\boldsymbol{A}}\right)\end{aligned}$$

现在我们将投影矩阵推广为 k 维，也即设投影矩阵为 $\boldsymbol{X}=\{\boldsymbol{X}_1,\cdots,\boldsymbol{X}_d\}$，则优化问题为

$$\begin{aligned}\min \quad & -J(\boldsymbol{X})\\ \textbf{s.t} \quad & \boldsymbol{X}_i^T\boldsymbol{X}_j=0, i\neq j\end{aligned}$$

由拉格朗日乘子法我们可以得知，要提取前 k 个特征，投影矩阵的最优解即为 $\boldsymbol{G}$ 前 k 大个特征值对应的特征向量所组成的矩阵。

由此我们就得到了使用 2DPCA 进行图像压缩的一般算法：（Algorithm 3）

Algorithm 3 基于 2DPCA 的图像压缩算法

载入图像序列矩阵并进行中心化得到 $\{\boldsymbol{X}_1,\boldsymbol{X}_2,\cdots,\boldsymbol{X}_M\}$

计算其平均图像 $\bar{\boldsymbol{X}}=\frac{1}{M}\sum_{i=1}^{M}\boldsymbol{X}_i$

计算整个数据集的协方差矩阵 $\boldsymbol{G}=\frac{1}{M}\sum_{i=1}^{M}\left(\boldsymbol{X}_i-\bar{\boldsymbol{X}}\right)^T\left(\boldsymbol{X}_i-\bar{\boldsymbol{X}}\right)$

求解 $\boldsymbol{G}$ 的特征值及对应的特征向量 $(\lambda_i,\boldsymbol{\alpha}_i)$

按特征值大小将前 k 个特征向量组成投影矩阵 $\boldsymbol{W}_k=(\boldsymbol{\alpha}_1,\cdots,\boldsymbol{\alpha}_k)$

对任一中心化图像 $\boldsymbol{X}$ 进行变换并作逆变换 $\tilde{\boldsymbol{Y}}=\boldsymbol{X}\boldsymbol{W}_k\boldsymbol{W}_k^T$

还原图像 $y_{ij}=\tilde{y}_{ij}+\bar{x}_i$

实现代码见附录。

在选择了不同主成分个数时，使用 2DPCA 算法进行图像压缩的效果结果如下图所示（Figure 2）。

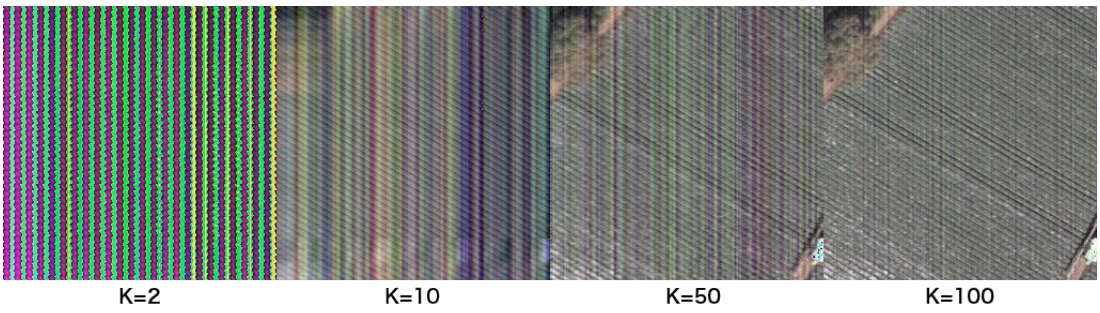

Figure 2: 不同主成分个数下使用 2DPCA 进行图像压缩的重构结果

3.4 2D-2DPCA

在上述介绍的 2DPCA 中，若我们用 $\boldsymbol{A}^{(t)}$ 来表示矩阵 $\boldsymbol{A}$ 的第 t 行，则

$$\boldsymbol{A}_i=\left(\left(\boldsymbol{A}_i^{(1)}\right)^T,\cdots,\left(\boldsymbol{A}_i^{(n)}\right)^T\right)^T$$

$$\bar{\boldsymbol{A}}=\left(\left(\bar{\boldsymbol{A}}^{(1)}\right)^T,\cdots,\left(\bar{\boldsymbol{A}}^{(n)}\right)^T\right)^T$$

，于是协方差矩阵 $\boldsymbol{G}$ 就可以被写为

$$\boldsymbol{G}=\frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{n}\left(\boldsymbol{A}_i^{(k)}-\bar{\boldsymbol{A}}^{(k)}\right)^T\left(\boldsymbol{A}_i^{(k)}-\bar{\boldsymbol{A}}^{(k)}\right)$$

。通过该式我们可以发现，$\boldsymbol{G}$ 可以通过图像集合中行向量的外积得到，这也就意味着 2DPCA 实际上仅按行提取了图像之间的联系。自然的，我们可以想到是否可以用类似的方法提取图像列之

间的联系并将它们结合在一起，从而挖掘出图像更多的特征联系。这一想法也就形成了所谓的**双方向二维主成分分析（2D-2DPCA）**[ZZ05]。

我们设 $\boldsymbol{Z} \in \mathbb{R}^{n\times k}$ 为另一投影矩阵，其作用即为对图像矩阵 $\boldsymbol{A}$ 的列进行投影。于是与 2DPCA 的做法类似，我们设矩阵 $\boldsymbol{A}$ 投影得到的特征为

$$\boldsymbol{B} = \boldsymbol{Z}^T\boldsymbol{A}$$

。于是投影得到的协方差矩阵即为

$$J(\boldsymbol{Z}) = \boldsymbol{Z}^T E\left[(\boldsymbol{A} - E[\boldsymbol{A}])(\boldsymbol{A} - E[\boldsymbol{A}])^T\right]\boldsymbol{Z}$$

我们定义

$$\begin{aligned}\boldsymbol{G}' &\overset{def}{=} E\left[(\boldsymbol{A} - E[\boldsymbol{A}])(\boldsymbol{A} - E[\boldsymbol{A}])^T\right] \\ &= \frac{1}{M}\sum_{i=1}^{M}\left(\boldsymbol{A}_i - \bar{\boldsymbol{A}}\right)\left(\boldsymbol{A}_i - \bar{\boldsymbol{A}}\right)^T \\ &= \frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{n}\left(\boldsymbol{A}_i^{(k)} - \bar{\boldsymbol{A}}^{(k)}\right)\left(\boldsymbol{A}_i^{(k)} - \bar{\boldsymbol{A}}^{(k)}\right)^T\end{aligned}$$

其中 $\boldsymbol{A}^{(t)}$ 为矩阵 $\boldsymbol{A}$ 的第 t 列。

与上面类似，我们最大化 $J(\boldsymbol{Z})$，同样可以得到 $\boldsymbol{Z}$ 的最优解即为 $\boldsymbol{G}$ 的前 k 大个特征值对应的特征向量所组成的矩阵。

将行投影和列投影结合，我们就能得到图像集合中的任一图像 $\boldsymbol{A}$ 经过投影后的特征为

$$\boldsymbol{C} = \boldsymbol{Z}^T\boldsymbol{A}\boldsymbol{Z}$$

，其重构图像为

$$\tilde{\boldsymbol{A}} = \boldsymbol{Z}\boldsymbol{C}\boldsymbol{X}^T$$

。

由此我们就得到了使用 2D-2DPCA 进行图像压缩的一般算法：（Algorithm 4）

Algorithm 4 基于 2D-2DPCA 的图像压缩算法

载入图像序列矩阵并进行中心化得到 $\{\boldsymbol{X}_1, \boldsymbol{X}_2, \cdots, \boldsymbol{X}_M\}$
计算其平均图像 $\bar{\boldsymbol{X}} = \frac{1}{M}\sum_{i=1}^{M}\boldsymbol{X}_i$
计算整个数据集的**行协方差矩阵** $\boldsymbol{G} = \frac{1}{M}\sum_{i=1}^{M}\left(\boldsymbol{X}_i - \bar{\boldsymbol{X}}\right)^T\left(\boldsymbol{X}_i - \bar{\boldsymbol{X}}\right)$
求解 $\boldsymbol{G}$ 的特征值及对应的特征向量 $(\lambda_i, \boldsymbol{\alpha}_i)$
按特征值大小将前 k 个特征向量组成投影矩阵 $\boldsymbol{W}_k = (\boldsymbol{\alpha}_1, \cdots, \boldsymbol{\alpha}_k)$
计算整个数据集的**列协方差矩阵** $\boldsymbol{G}' = \frac{1}{M}\sum_{i=1}^{M}\left(\boldsymbol{X}_i - \bar{\boldsymbol{X}}\right)\left(\boldsymbol{X}_i - \bar{\boldsymbol{X}}\right)^T$
求解 $\boldsymbol{G}'$ 的特征值及对应的特征向量 $(\lambda_i', \boldsymbol{\alpha}_i')$
按特征值大小将前 k 个特征向量组成投影矩阵 $\boldsymbol{U}_k = (\boldsymbol{\alpha}_1', \cdots, \boldsymbol{\alpha}_k')$
对任一中心化图像 $\boldsymbol{X}$ 进行变换并作逆变换 $\tilde{\boldsymbol{Y}} = \boldsymbol{U}_k\boldsymbol{U}_k^T\boldsymbol{X}\boldsymbol{W}_k\boldsymbol{W}_k^T$
还原图像 $y_{ij} = \tilde{y}_{ij} + \bar{x}_i$

实现代码见附录。

在选择了不同主成分个数时，使用 2D-2DPCA 算法进行图像压缩的效果结果如下图所示（Figure 3）。

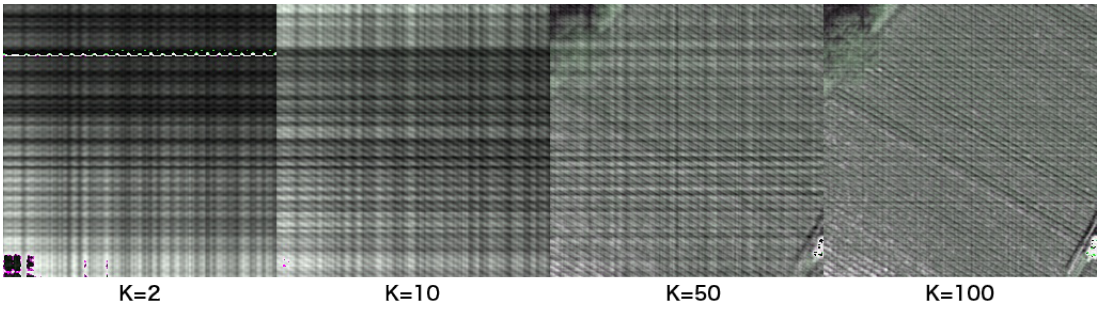

Figure 3: 不同主成分个数下使用 2D-2DPCA 进行图像压缩的重构结果

3.5 Generalized Hebbian Algorithm

可以看到，上面几种 PCA 方法均为基于矩阵分解的求解方法。近年来随着机器学习和神经网络模型的提出，研究者也提出了基于学习的方法来快速提取数据集的主成分。其中较为典型的即为基于 Gram-Schmidt 正交化方法的**扩展 Hebbian 算法（Generalized Hebbian Algorithm）**[San89]。GHA 算法是一种无监督的学习算法，我们可以将其看作 Oja 方法的一种推广 [Oja82]。

首先我们构建一个单层全连接神经网络（Single-layered Feed Forward Neural Network），如下图所示（Figure 4）。可以证明，通过 GHA 算法对该神经网络进行权重调整，则训练完成后，连接每个输出神经元的权重向量即为一个主成分向量。

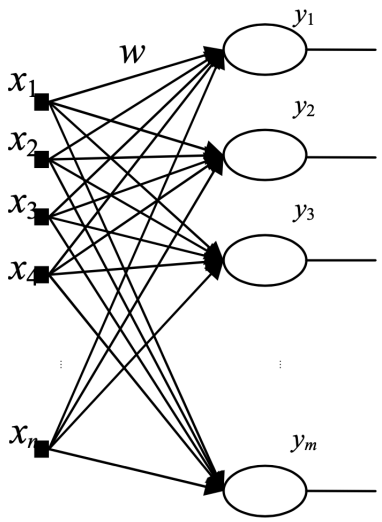

Figure 4: GHA 主成分提取算法所使用的单层感知机网络 [WJ12]

首先我们考虑输出层只有 1 个神经元的情况。若设第 t 次迭代的输入向量为 $\boldsymbol{x}_t \in \mathbb{R}$，网络的权重向量为 $\boldsymbol{w}_t \in \mathbb{R}^n$，输出 $y_t = \boldsymbol{w}_t^T \boldsymbol{x}_t$，$\eta$ 为学习率。则根据 Oja 方法，权重更新公式为

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta \boldsymbol{y}_t (\boldsymbol{x}_t - \boldsymbol{y}_t \boldsymbol{w}_t)$$

。现在我们将其推广到 k 个输出时的情况，即求解前 k 个主成分。设 $\boldsymbol{W}_t \in \mathbb{R}^{k \times n}$，输出 $\boldsymbol{y}_t = \boldsymbol{W}_t \boldsymbol{x}_t$，则

$$\boldsymbol{W}_{t+1} = \eta \left(\boldsymbol{y}_t \boldsymbol{x}_t^T - \mathrm{LOWER}(\boldsymbol{y}_t \boldsymbol{y}_t^T) \boldsymbol{W}_t \right)$$

，其中 LOWER($\boldsymbol{A}$) 为取矩阵 $\boldsymbol{A}$ 的下三角操作（上三角部分置为 0）。

于是，使用 GHA 进行图像压缩的算法流程如下：（Algorithm 5）

Algorithm 5 基于 GHA 的图像压缩算法

中心化图像点 $\tilde{x}_{ij} = x_{ij} - \bar{x}_i$

构建输出层有 k 个神经元的单层感知机模型 $\boldsymbol{y} = \boldsymbol{W}_k\tilde{\boldsymbol{x}}$

初始化权重矩阵 $\boldsymbol{W}_k$

repeat

前向传播 $\boldsymbol{y} \leftarrow \boldsymbol{W}_k\tilde{\boldsymbol{x}}$

更新权重 $\boldsymbol{W}_k \leftarrow \eta\left(\boldsymbol{y}\boldsymbol{x}^T - \text{LOWER}(\boldsymbol{y}\boldsymbol{y}^T)\boldsymbol{W}_k\right)$

until 到达指定迭代次数 t

对中心化图像点进行变换并作逆变换 $\tilde{\boldsymbol{Y}} = \tilde{\boldsymbol{X}}\boldsymbol{W}_k\boldsymbol{W}_k^T$

还原图像 $y_{ij} = \tilde{y}_{ij} + \bar{x}_i$

对于训练结果的评价指标，我们定义损失函数

$$\mathcal{L}(\boldsymbol{W}, \boldsymbol{Q}) = \sum_{i=1}^{k} ||\boldsymbol{w}_i - \boldsymbol{q}_i||^2 = ||\boldsymbol{W} - \boldsymbol{Q}||_F^2$$

及向量夹角

$$\mathcal{A}(\boldsymbol{w}_i, \boldsymbol{q}_i) = \arccos\left(\frac{\boldsymbol{w}_i \cdot \boldsymbol{q}_i}{||\boldsymbol{w}_i||_2 \cdot ||\boldsymbol{q}_i||_2}\right)$$

实现代码见附录。

由于该算法的收敛速度较为不可控，因此这里我们仅使用网络提取前两个主成分。我们使用正态分布 $\mathcal{N}(0, 0.5)$ 随机初始化了权重矩阵，并设置学习率 $\eta = 10^{-4}$ 进行了两次模拟，每次对网络进行了 20000 次训练迭代。两次迭代过程中 Loss 和向量夹角的变化情况如下图所示（Figure 5 和 Figure 6）。可以发现，随着权重矩阵初始值的不同，网络的收敛特征也会随之发生变化。

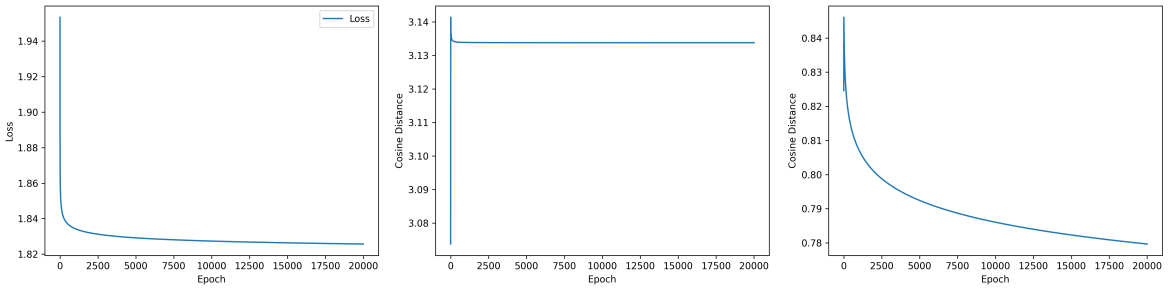

Figure 5: GHA 训练中 Loss 及向量夹角随迭代次数的变化（第一次模拟）

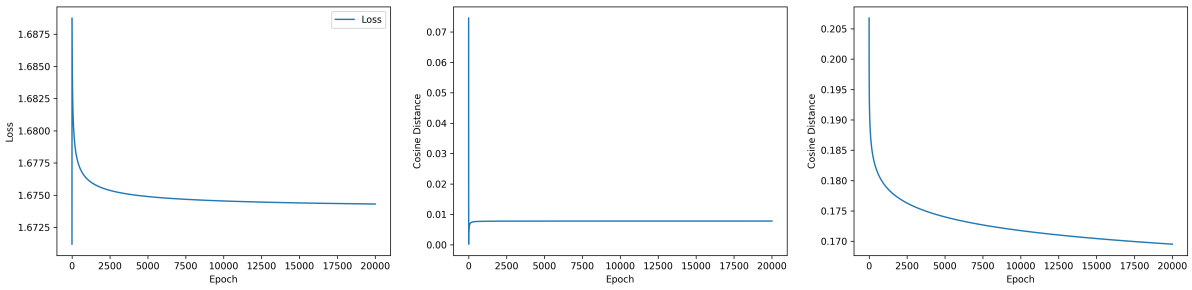

Figure 6: GHA 训练中 Loss 及向量夹角随迭代次数的变化（第二次模拟）

使用神经网络算法进行模型训练时一个经典的问题即为学习率过大。事实上，在训练该模型的过程中，我们同样遇到了这一问题，并且固定的学习率很难保证适合整个损失超平面。为此，研究者提出了适用于 GHA 的自适应学习率优化方法 [CC95]。由于该方法实现过于复杂，在此仅作为了解。

3.6 Kernel PCA

由 PCA 的推导过程我们可以看出，要对一个数据集使用 PCA 方法进行降维的一大前提即为数据集必须在当前维度下线性可分，而类似图像这样高度紧凑的数据集通常会出现线性不可分的问题。在这种情况下，我们通常会尝试使用一个映射 $\phi : \mathbb{R}^n \to \mathbb{R}^d, d > n$ 将数据集映射到更高维度的特征空间，使得其在该空间下线性可分，该方法被称为**核方法**（**Kernel Method**）。利用核方法，我们可以对朴素的 PCA 方法进行改进，使其能够表达更多的原始特征，这就形成了所谓的**核主成分分析**（**Kernel PCA**）[SSM98]。

若设图像矩阵为 $\boldsymbol{X}$，非线性映射 $\phi(\boldsymbol{X})$ 对应的核函数 $\boldsymbol{K} = \phi(\boldsymbol{X})^T\phi(\boldsymbol{X})$，特征空间为 $\mathcal{F}$，则特征空间中的协方差矩阵就可以写为

$$\boldsymbol{C}_{\mathcal{F}} = \frac{1}{N}\phi(\boldsymbol{X})(\phi(\boldsymbol{X}))^T$$

其特征值问题的方程 $\boldsymbol{C}_{\mathcal{F}}\boldsymbol{v} = \lambda\boldsymbol{v}$ 就可以写为

$$\sum_{i=1}^{N}\phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_i)^T\boldsymbol{v} = \lambda\boldsymbol{v}$$

由此我们发现其每一个特征向量 $\boldsymbol{v}_j$ 都可以表示为 $\phi(\boldsymbol{x}_i)$ 的线性组合

$$\boldsymbol{v} = \sum_{i=1}^{N}a_i\phi(\boldsymbol{x}_i) = \phi(\boldsymbol{X})\boldsymbol{a}$$

，其中 $\boldsymbol{a} = (a_1, \cdots, a_N)^T$。引入核函数，化简即可得到 [CE16]

$$\boldsymbol{K}(\boldsymbol{X})\boldsymbol{a} = \lambda\boldsymbol{a}$$

，也即经过特征空间所得的降维变换向量即为矩阵 $\boldsymbol{K}$ 的特征向量。

使用 Kernel PCA 进行图像压缩时所涉及的变换如下图所示（Figure 7）。

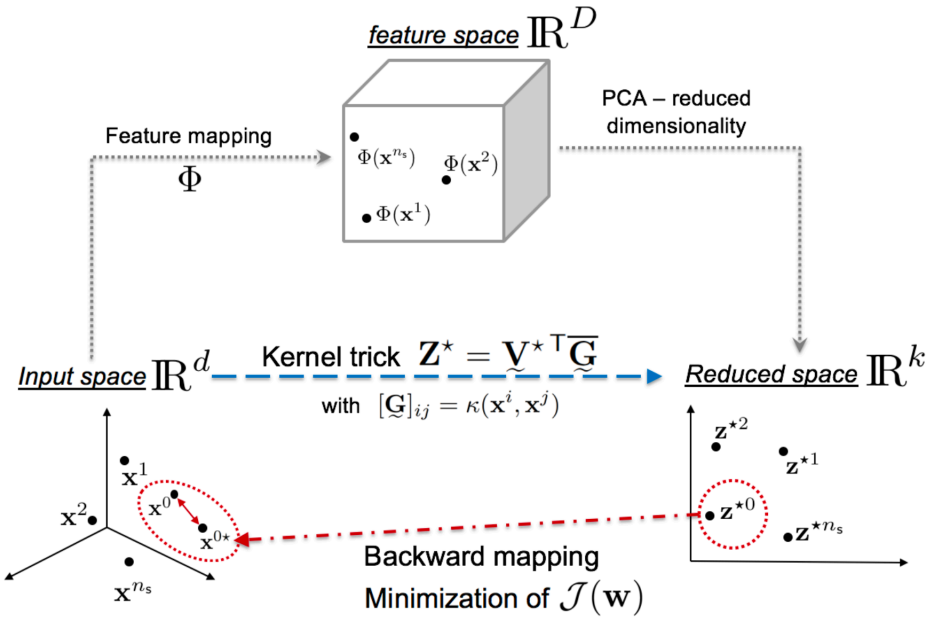

Figure 7: 使用核主成分分析进行图像压缩的变换过程示意图 [GGHZD21]

通常来说，核函数要求矩阵为正定矩阵。在本文中，我们实现了以下几种核函数：

- **线性核（Linear Kernel）**

$$\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T \boldsymbol{y}$$

- **多项式核（Polynomial Kernel）**

$$\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y}) = \left(\boldsymbol{x}^T \boldsymbol{y} + c\right)^d$$

- **高斯核（Gaussian Kernel/Radial Basis Function Kernel）**

$$\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{y}||^2}{2\sigma^2}\right) = \exp\left(-\gamma||\boldsymbol{x} - \boldsymbol{y}||^2\right)$$

- **指数核（Exponential Kernel）**

$$\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{y}||}{2\sigma^2}\right) = \exp\left(-\gamma||\boldsymbol{x} - \boldsymbol{y}||\right)$$

- **ANOVA 核**

$$\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\sigma\left(\boldsymbol{x}^k - \boldsymbol{y}^k\right)^2\right)^d$$

- **Sigmoid 核**

$$\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y}) = \tanh\left(a\boldsymbol{x}^T \boldsymbol{y} + r\right)$$

对于部分核函数，我们还需要给定合适的超参数以达到最好的特征提前效果。以高斯核为例，我们采用网格搜索的方式来选取合适的超参数 γ。图像的重构误差随 γ 的变化如下图所示（Figure 8 左一）。可以看到，随着 γ 值的增大，图像的重构误差逐渐减少。然而这并不意味着 γ 值越大

越好，这是由于当 γ 值过大时，模型会出现**过拟合**（**overfitting**）的问题。具体来说，当 γ 值过大时，核函数

$$\boldsymbol{K}(\boldsymbol{x},\boldsymbol{y}) \approx \begin{cases} e^0 = 1 & ,\boldsymbol{x} = \boldsymbol{y} \\ e^{-\infty} = 0 & ,\boldsymbol{x} \neq \boldsymbol{y} \end{cases}$$

，此时核矩阵退化为 $\boldsymbol{I}_n$，也即单位变换。这就导致了降维空间成为原空间的一个子空间，自然就失去了特征提取的功能。

若设核矩阵 $\boldsymbol{K}$ 的前 k 个特征值为 $\lambda_1, \cdots, \lambda_k$，我们定义其方差（该指标衡量了特征值的分散程度）为

$$Var(\lambda_1, \cdots, \lambda_k) = \frac{1}{k}\sum_{i=1}^{k}\left(\lambda_i - \bar{\lambda}\right)^2$$

，则核矩阵方差及核矩阵行列式的值如下图所示（Figure 8 左二、左三）。易见当 $\gamma \gg 10^{-3}$ 时，核矩阵的方差趋近于 0，其行列式趋近于 1，这也印证了上述的理论论述，表明模型确实出现了过拟合。

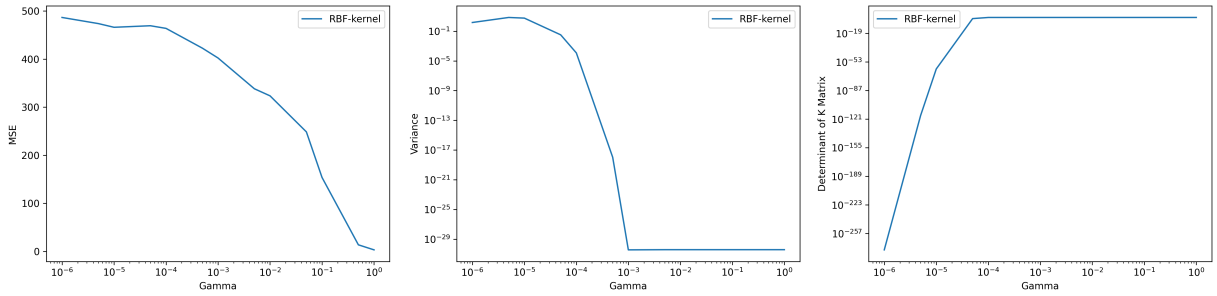

Figure 8: 使用 RBF 核实现 Kernel PCA 时 MSE、核矩阵特征值方差及核矩阵行列式值随参数 γ 的变化

可以看到，由于核函数基本都为非线性函数，其逆变换通常难以求得。因此使用核方法对数据集进行降维的一个很大的问题在于对数据集进行重构，这对于图像压缩问题来说是十分不友好的。

若设 $\mathcal{H}_K$ 为核 $\boldsymbol{K}(\boldsymbol{x},\boldsymbol{y})$ 所生成的再生希尔伯特核空间，其对应的特征变换 $\phi(\boldsymbol{x}) : \mathbb{R}^n \to \mathcal{H}_k$，则图像重构问题即为给定 $\mathcal{H}_K$ 中的一点 $\boldsymbol{\Psi}$，求输入空间中的一点 $\boldsymbol{z} \in \mathbb{R}^n$，使得

$$\boldsymbol{z} = \arg\min_{\boldsymbol{z}} ||\boldsymbol{\Psi} - \phi(\boldsymbol{z})||^2$$

事实上自 Kernel PCA 被提出以来，已经有大量的研究提出了一系列对 KPCA 降维后数据进行重构的方法，这些方法大多都是基于近似拟合的方法 [GGHZD21]。其中，基于**梯度下降**（**Gradient Descent**）的方法 [MSS+99] 和基于**回归**（**Regression**）的方法 [WSB04] 是两大较为有代表性的求解方法。由于这一过程实现过于复杂，我们直接使用了 Scikit-Learn 工具包中提供的 **inverse_transform()** 函数来完成。

使用 Kernel PCA 进行图像压缩的算法流程如下：（Algorithm 6）

Algorithm 6 基于 Kernel PCA 的图像压缩算法

中心化图像点 $\tilde{x}_{ij} = x_{ij} - \bar{x}_i$

选定核函数，以列为单位计算原矩阵核矩阵 $\boldsymbol{K}$

求解 $\boldsymbol{K}$ 的特征值及对应的特征向量 $(\lambda_i, \boldsymbol{\alpha}_i)$

按特征值从大到小将前 k 个特征向量按行排列成变换矩阵 $\boldsymbol{Q}_k$

对中心化图像点进行变换 $\tilde{\boldsymbol{Z}} = \boldsymbol{Q}_k \tilde{\boldsymbol{X}}$

近似求解中心化重构图像 $\tilde{\boldsymbol{Y}} = \arg\min_{\boldsymbol{Y}} ||\tilde{\boldsymbol{Z}} - \phi(\tilde{\boldsymbol{Y}})||_F^2$

还原图像 $y_{ij} = \tilde{y}_{ij} + \bar{x}_i$

实现代码见附录。

以高斯核为例，在选择了不同主成分个数时，使用 Kernel PCA 算法进行图像压缩的效果结果如下图所示（Figure 9）。可以发现，与朴素 PCA 方法不同，当选择的主成分个数为 10 时图像的主要特征仍然没有得到恢复，而当 $k = 50$ 时，图像的质量得到了极大的改善。这也表明经过变换后的数据集在核空间下的特征分离方式与原空间下是不同的。

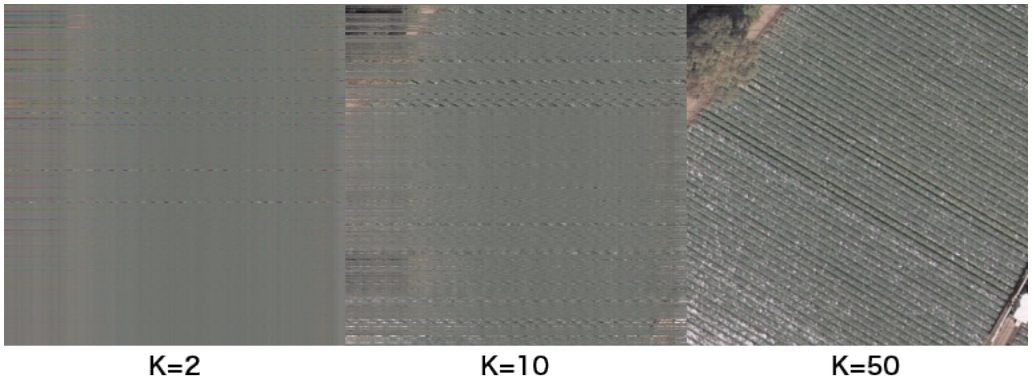

Figure 9: 不同主成分个数下使用 Kernel PCA 进行图像压缩的重构结果（RBF 核）

3.7 JPEG

上面使用的几种图像压缩算法均为即为基于 PCA 的方法。事实上，在日常场景下，人们更常使用基于信号处理和特殊编码的方法来对图像进行压缩。其中较为典型的代表即为基于离散余弦变换的 JPEG（JFIF）算法。

JPEG 图像压缩算法的具体流程如下图所示（Figure 10）。压缩算法主要分为如下几个步骤：[Wik21a]

1. $RGB \to YC_bC_r$ **空间转换**
2. **下采样**
3. **图像分割**
4. **离散余弦变换**
5. **数据量化**
6. **Huffman 编码**

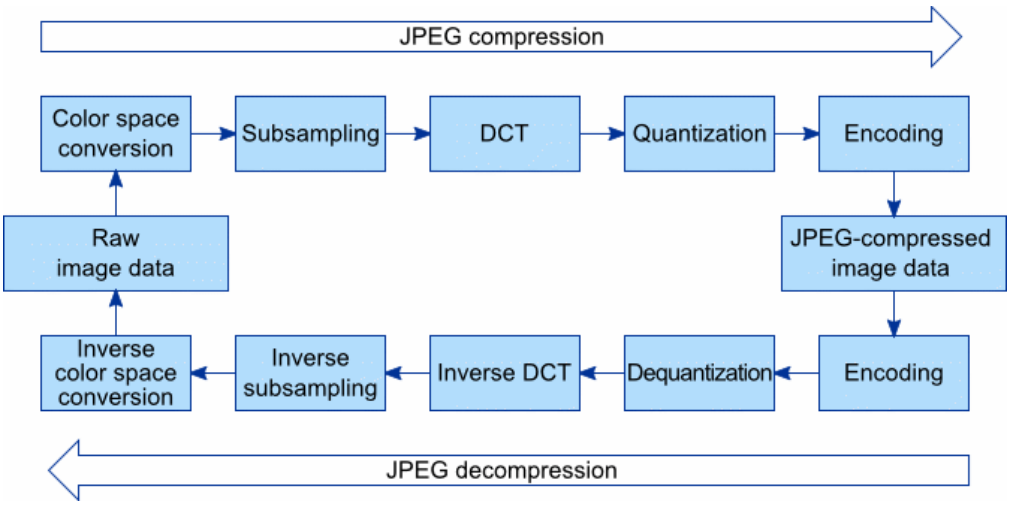

Figure 10: JPEG 图像压缩/重构流程 [Mil]

具体实现见附录和源码。

使用 JPEG 算法进行图像压缩的效果结果如下图所示（Figure 11）。可以看到，尽管 JPEG 为有损压缩算法，重构后的图片与原图几乎看不到可见的差异，这也从一定程度上解释了该算法流行的原因。

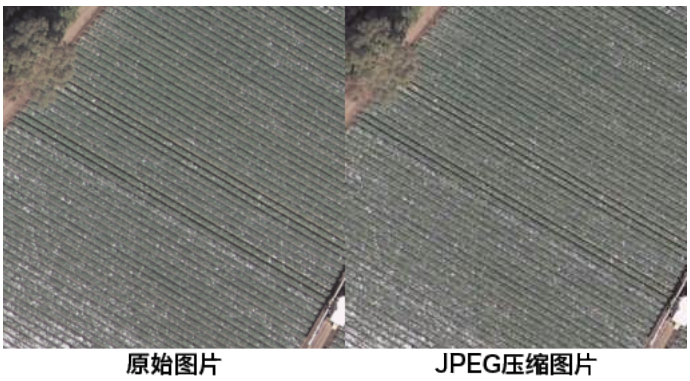

Figure 11: 使用 JPEG 进行图像压缩的重构结果

事实上，近年来的许多压缩方法还会将 JPEG 算法及其变体 JPEG2000 与 PCA 方法相结合，从而进一步提高压缩率及重构的准确率。例如在高光谱成像领域，由于原始图像通常还会附带许多频谱信息，将这两种方法相结合可以极大的压缩存储图像所需的空间，从而减少数据传输的开销 [BGM18]。

4 实验结果

前文中我们提及了一系列图像压缩的方法，现在我们从**压缩率**、**重构质量**和**压缩耗时**三个维度来对上述提及的所有方法进行分析和比较。

图像的压缩率被定义为

$$\eta = 1 - \frac{\text{Size}(\tilde{\boldsymbol{X}}) + \text{Size}(\boldsymbol{Q})}{\text{Size}(\boldsymbol{X})}$$

，其中 Size($\boldsymbol{A}$) 为 $\boldsymbol{A}$ 的空间度量，$\boldsymbol{X}$ 为原始图像，$\tilde{\boldsymbol{X}}$ 为重构图像，$\boldsymbol{Q}$ 为重构变换矩阵。

以 $k = 50$ 为例，本文中实现的不同算法的压缩率如下表所示：

压缩算法	单张图片压缩率	100 张图片压缩率	300 张图片压缩率
PCA	60.94%	60.94%	60.94%
2DPCA	60.94%	80.27%	80.40%
2D-2DPCA	41.41%	80.08%	80.34%
Kernel PCA	60.94%	60.94%	60.94%
JPEG	84.54%	86.09%	87.22%

Table 1: $k = 50$ 时不同图像压缩算法的压缩率

可以发现，朴素 PCA 和 Kernel PCA 对单张图片计算主成分，因此其在单张图片和多张图片上的压缩率相同；而 2DPCA 和 2D-2DPCA 由于对整个数据集计算特征，因此随着数据集的增长，总体的图像压缩率逐渐增长。JPEG 在所有算法中拥有最高的压缩率。

对于重构质量，我们使用**均方误差（Mean Square Error）**和**峰值信噪比（Peak Signal-to-Noise Ratio）**来进行评估。其中，原始图像 $\boldsymbol{X}$ 和重构图像 $\tilde{\boldsymbol{X}}$ 间的均方误差被定义为

$$\mathrm{MSE}(\boldsymbol{X}, \tilde{\boldsymbol{X}}) = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} (x_{ij} - \tilde{x}_{ij})^2$$

其之间的峰值信噪比被定义为

$$\mathrm{PSNR}(\boldsymbol{X}, \tilde{\boldsymbol{X}}) = 10 \cdot \log_{10} \left(\frac{\mathrm{MAX}_{\boldsymbol{X}}^2}{\mathrm{MSE}(\boldsymbol{X}, \tilde{\boldsymbol{X}})} \right)$$

，其中 $\mathrm{MAX}_{\boldsymbol{X}}$ 为矩阵 $\boldsymbol{X}$ 每个元素可能的最大值（对于一张 8 位图像即为 255）。

当 k 为 50 时，使用不同算法进行压缩重构后得到的 MSE 和 PSNR 值由下表给出：

压缩算法	MSE	PSNR
PCA	47.8774	31.3295
2DPCA	76.1444	29.3144
2D-2DPCA	90.1605	28.5806
Kernel PCA	11.1868	37.6437
JPEG	53.2189	30.8701

Table 2: $k = 50$ 时不同图像压缩算法的 MSE 和 PSNR 值

更进一步的，当主成分选择数 k 从 1 上升到 200 的过程中，不同算法的重构质量如下图所示（Figure 12）。可以看到，在所有实现的方法中，Kernel PCA 在两项指标中均获得了最好的结果。然而，由于其重构变换使用了近似的方法，这一过程并不稳定，因此其误差曲线出现了一定程度的波动。相比较而言，一般的 PCA 方法两项评价指标随主成分选择数增加的变化十分稳定。

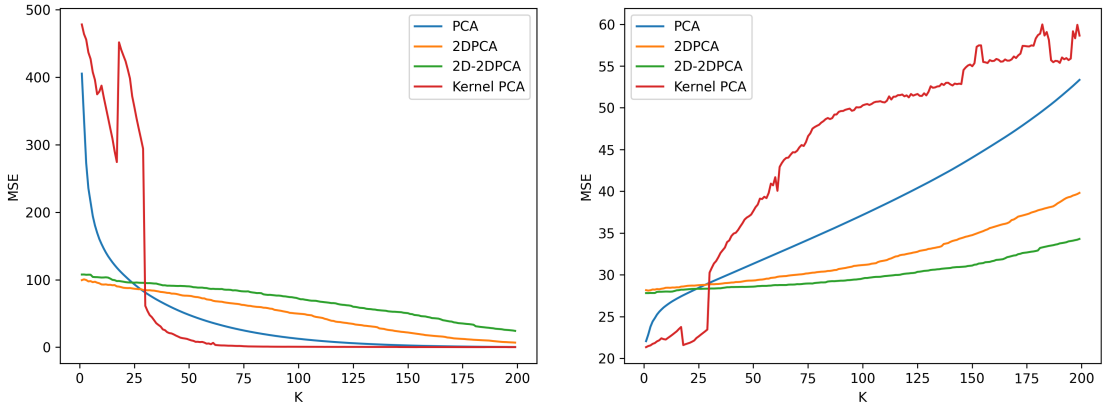

Figure 12: 不同 PCR 变种算法进行图像压缩的 MSE 和 PSNR 值随主成分选择数的变化情况

最后我们来考察不同压缩算法的压缩耗时。压缩耗时的计算公式被定义为

$$\mathcal{T} = \mathcal{T}_{Compress} + \mathcal{T}_{Reconstruct}$$

其中 $\mathcal{T}_{Compress}$ 为编码耗时，$\mathcal{T}_{Reconstruct}$ 重构耗时。

不同算法随着待压缩的图片总量从单张到 100 张所用的时间如下图所示（Figure 13）。可以发现，基于特征分解的朴素 PCA 算法随着数据集的增长进行压缩所用的时间迅速的增大，这是由于其计算协方差矩阵的特征值和特征向量的巨额时间开销。基于奇异值分解的 PCA 算法由于使用了更为高效的 Householder 变换算法，其耗时相比特征值分解得到了显著的下降。在我们的实现中，Kernel PCA 同样使用了 SVD 方法进行优化，但其时间开销仍然十分巨大，表明其主要耗时在图像重构上。2DPCA 和 2D-2DPCA 由于对整个数据集进行统一变换，因此在多张图像数据集上拥有极高的压缩效率。

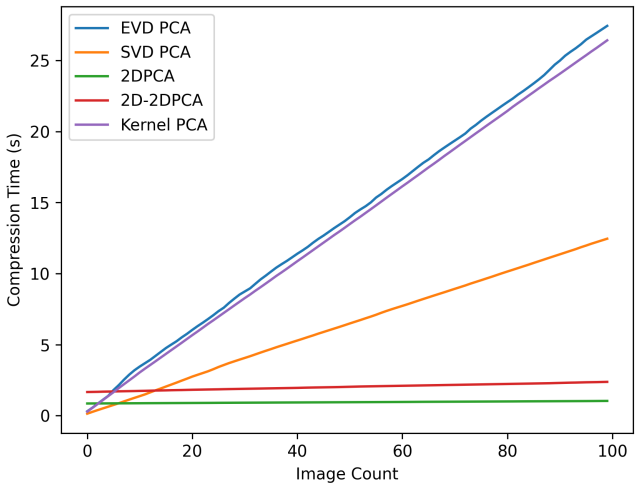

Figure 13: 不同 PCR 变种算法进行图像压缩耗时随数据集大小的变化情况

5 结论

在本实验中，我们完整推导并实现了**基于特征值分解的 PCA 算法**、**基于奇异值分解的 PCA 算法**、**基于 GHA 的 PCA 算法**、**2DPCA 算法**、**2D-2DPCA 算法**、**Kernel PCA 算法**及**JPEG 算法**，并将它们应用于图像压缩任务中。经过比较我们可以发现，在不同的度量标准下，不同的算法均有着相应的优势和劣势。一般的 PCA 算法具有较好的稳定性和可解释性，2DPCA 和 2D-2DPCA 算法拥有较高的压缩效率，Kernel PCA 算法拥有较高的重构精度。这也表明这些方法没有严格的好坏之分，在不同任务下需要根据实际情况选择合适的方法。

References

[BGM18] Daniel Báscones, Carlos González, and Daniel Mozos. Hyperspectral image compression using vector quantization, pca and jpeg2000. *Remote sensing*, 10(6):907, 2018.

[CC95] Liang-Hwa Chen and Shyang Chang. An adaptive learning algorithm for principal component analysis. *IEEE Transactions on Neural Networks*, 6(5):1255–1263, 1995.

[CE16] COMP-652 and ECSE-608. Dimensionality reduction. pca. kernel pca. https://www.cs.mcgill.ca/~dprecup/courses/ML/Lectures/ml-lecture13.pdf, 2016.

[GGHZD21] Alberto García-González, Antonio Huerta, Sergio Zlotnik, and Pedro Díez. A kernel principal component analysis (kpca) digest with a new backward mapping (pre-image reconstruction) strategy, 2021.

[Gor18] Matt Gormley. Deriving principal component analysis (pca). https://www.cs.cmu.edu/~mgormley/courses/606-607-f18/slides606/lecture11-pca.pdf, October 2018.

[Leh94] R.B. Lehoucq. The computation of elementary unitary matrices. Technical report, University of Tennessee, 1994.

[Mil] Graphics Mill. Working with jpeg. https://www.graphicsmill.com/docs/gm/working-with-jpeg.htm.

[MSS+99] Sebastian Mika, Bernhard Schölkopf, Alex Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. In M. Kearns, S. Solla, and D. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11. MIT Press, 1999.

[Oja82] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[San89] Terence D. Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network, 1989.

[SSM98] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319, 1998.

[Wik21a] Wikipedia contributors. Jpeg — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=JPEG&oldid=1056557277, 2021. [Online; accessed 23-November-2021].

[Wik21b] Wikipedia contributors. Singular value decomposition — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Singular_value_decomposition&oldid=1055826758, 2021. [Online; accessed 24-November-2021].

[WJ12] Chih-Wen Wang and Jyh-Horng Jeng. Image compression using pca with clustering. In *2012 International Symposium on Intelligent Signal Processing and Communications Systems*, pages 458–462, 2012.

[Woo09] Frank Wood. http://www.stat.columbia.edu/~fwood/Teaching/w4315/Fall2009/pca.pdf, December 2009.

[WSB04] Jason Weston, Bernhard Schölkopf, and Gökhan Bakir. Learning to find pre-images. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2004.

[YZFyY04] Jian Yang, D. Zhang, A.F. Frangi, and Jing yu Yang. Two-dimensional pca: a new approach to appearance-based face representation and recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(1):131–137, 2004.

[ZZ05] Daoqiang Zhang and Zhi-Hua Zhou. (2d)2pca: Two-directional two-dimensional pca for efficient face representation and recognition. *Neurocomputing*, 69(1):224–231, 2005. Neural Networks in Signal Processing.

[周 16] 周志华. 机器学习. 清华大学出版社, 2016.

A 附录. 本文中所使用算法的 Python 实现代码

A.1 基于特征值分解的 PCA 图像压缩

```python
import cv2
import numpy as np


def compression(mat, k):
    img_mean = np.mean(mat, axis=0)
    mat = mat - img_mean
    cov = mat @ mat.T
    eig_val, eig_vec = np.linalg.eig(cov)
    eig_val_index = np.argsort(-eig_val)
    trans_mat = []
    for i in range(k):
        trans_mat.append(eig_vec[:, eig_val_index[i]])
    trans_mat = np.array(trans_mat)
    compressed_mat = trans_mat.T @ trans_mat @ mat
    compressed_mat = compressed_mat + img_mean
    return compressed_mat


def evaluate(mat_origin, mat_comp):
    mse = np.mean(np.square(mat_origin - mat_comp))
    psnr = 10 * np.log10(255 * 255 / mse)
    print('MSE:', mse)
    print('PSNR:', psnr)
    return mse, psnr


comp_n = 50  # Number of principle components
i = 0
img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
b, g, r = cv2.split(img)
comp_b = compression(b, comp_n)
comp_g = compression(g, comp_n)
comp_r = compression(r, comp_n)
img_compressed = cv2.merge([comp_b, comp_g, comp_r])
evaluate(img, img_compressed)
cv2.imwrite('naive_pca.jpg', np.array(img_compressed, dtype='uint8'))
```

A.2 基于奇异值分解的 PCA 图像压缩

```python
import cv2
import numpy as np
import time
```

```
4
5
6  def compression(mat, k):
7      img_mean = np.mean(mat, axis=0)
8      mat = mat - img_mean
9      u, s, v = np.linalg.svd(mat)
10     trans_mat = v.T[:, :k]
11     compressed_mat = mat @ trans_mat @ trans_mat.T
12     compressed_mat = compressed_mat + img_mean
13     return compressed_mat
14
15
16 def evaluate(mat_origin, mat_comp):
17     mse = np.mean(np.square(mat_origin - mat_comp))
18     psnr = 10 * np.log10(255 * 255 / mse)
19     print('MSE:', mse)
20     print('PSNR:', psnr)
21
22
23 comp_n = 50  # Number of principle components
24 i = 0
25 img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
26 b, g, r = cv2.split(img)
27 comp_b = compression(b, comp_n)
28 comp_g = compression(g, comp_n)
29 comp_r = compression(r, comp_n)
30 img_compressed = cv2.merge([comp_b, comp_g, comp_r])
31 evaluate(img, img_compressed)
32 cv2.imwrite('svd_pca.jpg', np.array(img_compressed, dtype='uint8'))
```

A.3 基于 2D-PCA 的图像压缩

```
1  import numpy as np
2  import cv2
3
4
5  def compression_2dpca(images, k):
6      size = images[0].shape
7      mean_mat = np.zeros(size)
8      for im in images:
9          mean_mat = mean_mat + im
10     mean_mat /= float(len(images))
11     cov_row = np.zeros((size[1], size[1]))
12     for s in images:
13         diff_mat = s - mean_mat
14         cov_row = cov_row + np.dot(diff_mat.T, diff_mat)
15     cov_row /= float(len(images))
```

```
    eig_val, eig_vec = np.linalg.eig(cov_row)
    sorted_index = np.argsort(eig_val)
    x_k_mat = eig_vec[:, sorted_index[:-k - 1: -1]]
    return x_k_mat


def evaluate(mat_origin, mat_comp):
    mse = np.mean(np.square(mat_origin - mat_comp))
    psnr = 10 * np.log10(255 * 255 / mse)
    print('MSE:', mse)
    print('PSNR:', psnr)
    return mse, psnr


samples_b, samples_g, samples_r = [], [], []
for i in range(0, 100):
    img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
    img_b, img_g, img_r = cv2.split(img)
    img_b_normed = np.empty(img_b.shape)
    img_g_normed = np.empty(img_g.shape)
    img_r_normed = np.empty(img_r.shape)
    for j in range(img_b.shape[0]):
        for k in range(img_b.shape[1]):
            img_b_normed[j, k] = img_b[j, k] / 255.0
            img_g_normed[j, k] = img_g[j, k] / 255.0
            img_r_normed[j, k] = img_r[j, k] / 255.0
    samples_b.append(img_b_normed)
    samples_g.append(img_g_normed)
    samples_r.append(img_r_normed)

pc = 50
trans_b = compression_2dpca(samples_b, pc)
trans_g = compression_2dpca(samples_g, pc)
trans_r = compression_2dpca(samples_r, pc)
i = 0
img_origin = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
Y_b = samples_b[0] @ trans_b @ trans_b.T
Y_g = samples_g[0] @ trans_g @ trans_g.T
Y_r = samples_r[0] @ trans_r @ trans_r.T
Y_b, Y_g, Y_r = Y_b * 255.0, Y_g * 255.0, Y_r * 255.0
Y_b = np.array(Y_b, dtype='uint8')
Y_g = np.array(Y_g, dtype='uint8')
Y_r = np.array(Y_r, dtype='uint8')
img_compressed = cv2.merge([Y_b, Y_g, Y_r])
mse, psnr = evaluate(img_origin, img_compressed)
cv2.imwrite('2dpca.jpg', img_compressed)
```

A.4 基于 2D2D-PCA 的图像压缩

```
import numpy as np
import cv2


def compresion_2d2dpca(images, k_row, k_col):
    size = images[0].shape
    mean_mat = np.zeros(size)
    for im in images:
        mean_mat = mean_mat + im
    mean_mat /= float(len(images))
    cov_row = np.zeros((size[1], size[1]))
    for im in images:
        diff_mat = im - mean_mat
        cov_row = cov_row + np.dot(diff_mat.T, diff_mat)
    cov_row /= float(len(images))
    eig_val_row, eig_vec_row = np.linalg.eig(cov_row)
    sorted_index = np.argsort(eig_val_row)
    x_k_mat = eig_vec_row[:, sorted_index[:-k_row - 1: -1]]
    cov_col = np.zeros((size[0], size[0]))
    for im in images:
        diff_mat = im - mean_mat
        cov_col += np.dot(diff_mat, diff_mat.T)
    cov_col /= float(len(images))
    eig_val_col, eig_vec_col = np.linalg.eig(cov_col)
    sorted_index = np.argsort(eig_val_col)
    z_k_mat = eig_vec_col[:, sorted_index[:-k_col - 1: -1]]
    return x_k_mat, z_k_mat


def evaluate(mat_origin, mat_comp):
    mse = np.mean(np.square(mat_origin - mat_comp))
    psnr = 10 * np.log10(255 * 255 / mse)
    print('MSE:', mse)
    print('PSNR:', psnr)
    return mse, psnr


samples_b, samples_g, samples_r = [], [], []
for i in range(0, 100):
    img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
    img_b, img_g, img_r = cv2.split(img)
    img_b_normed = np.empty(img_b.shape)
    img_g_normed = np.empty(img_g.shape)
    img_r_normed = np.empty(img_r.shape)
    for j in range(img_b.shape[0]):
        for k in range(img_b.shape[1]):
```

```
            img_b_normed[j, k] = img_b[j, k] / 255.0
            img_g_normed[j, k] = img_g[j, k] / 255.0
            img_r_normed[j, k] = img_r[j, k] / 255.0
    samples_b.append(img_b_normed)
    samples_g.append(img_g_normed)
    samples_r.append(img_r_normed)

pc_row, pc_col = 50, 50
trans_b_x, trans_b_z = compresion_2d2dpca(samples_b, pc_row, pc_col)
trans_g_x, trans_g_z = compresion_2d2dpca(samples_g, pc_row, pc_col)
trans_r_x, trans_r_z = compresion_2d2dpca(samples_r, pc_row, pc_col)
i = 0
img_origin = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
res_b = trans_b_z @ trans_b_z.T @ samples_b[i] @ trans_b_x @ trans_b_x.T
res_g = trans_b_z @ trans_b_z.T @ samples_g[i] @ trans_b_x @ trans_b_x.T
res_r = trans_b_z @ trans_b_z.T @ samples_b[i] @ trans_b_x @ trans_b_x.T
res_b = np.array(res_b * 255.0, dtype='uint8')
res_g = np.array(res_g * 255.0, dtype='uint8')
res_r = np.array(res_r * 255.0, dtype='uint8')
img_compression = cv2.merge([res_b, res_g, res_r])
mse, psnr = evaluate(img_origin, img_compression)
cv2.imwrite('2d2dpca.jpg', img_compression)
```

A.5 基于 GHA 的主成分提取

```
import numpy as np
from copy import deepcopy
import cv2


def update_weight(x, w, iteration, lr):
    y = np.dot(w, x)
    LT = np.tril(np.matmul(y[:, np.newaxis], y[np.newaxis, :]))
    w = w + lr / iteration * ((y[:, np.newaxis] * x) - (np.matmul(LT, w)))
    return w


def get_pca(mat, lr, k, max_iter, correct_mat):
    row, col = mat.shape
    wig = np.random.normal(0, 0.5, (k, row))
    wig_norm = wig / np.linalg.norm(wig, axis=1).reshape(k, 1)
    w_new = deepcopy(wig_norm)
    epoch = 1
    while epoch <= max_iter:
        for i in range(0, row):
            w_new = update_weight(mat[i], w_new, epoch, lr)
        print('Epoch: ', epoch, ', Loss: ', np.linalg.norm(w_new - correct_mat))
```

```
        print('PC1 Angle:', np.arccos(np.dot(w_new.T[:, 0], correct_mat.T[:, 0])
            / (
                    np.linalg.norm(w_new.T[:, 0]) * (np.linalg.norm(correct_mat.T
                        [:, 0]))))
        print('PC2 Angle:', np.arccos(np.dot(w_new.T[:, 1], correct_mat.T[:, 1])
            / (
                    np.linalg.norm(w_new.T[:, 1]) * (np.linalg.norm(correct_mat.T
                        [:, 1]))))
        epoch += 1
    return w_new


eta = 1e-6
max_epoch = 20000

i = 0
img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i), 0)
img_mean = np.mean(img, axis=0)
img = img - img_mean

cov = img @ img.T
eig_val, eig_vec = np.linalg.eig(cov)
eig_val_index = np.argsort(-eig_val)
trans_mat = []
for i in range(2):
    trans_mat.append(eig_vec[:, eig_val_index[i]])
trans_mat = np.array(trans_mat)

w_new = get_pca(img, eta, 2, max_epoch, trans_mat)
print('Check if GHA 2 PCs are orthogonal: ')
print(np.dot(w_new[0], w_new[1]))
img_compressed = img @ w_new.T @ w_new
img_compressed = img_compressed + img_mean
cv2.imwrite('gha_pca.jpg', np.array(img_compressed, dtype='uint8'))
```

A.6 基于 Kernel PCA 的图像压缩

A.6.1 图像压缩（正变换）

```
import cv2
import numpy as np


def linear(self, x, y):
    return x.T @ y + self.c


def polynomial(self, x, y):
```

```
    return (self.alpha * (x.T @ y) + self.c) ** self.d_poly


def exp(self, x, y):
    return np.exp(- (1 / (2 * self.sigma ** 2)) * np.linalg.norm(x - y))


def rbf(x, y, gamma):
    return np.exp(- gamma * (np.linalg.norm(x - y) ** 2))


def anova(self, x, y):
    sum_a = 0
    for i in range(0, len(x)):
        term_1 = - self.sigma * ((x[i] - y[i]) ** 2)
        sum_a += np.exp(term_1) ** self.d_anova
    return sum_a


def sigmoid(self, x, y):
    return np.tanh(self.alpha * (x.T @ y) + self.c)


def compression_transform(mat, gamma, k, kernel):
    k_mat = []
    row, col = mat.shape
    for i in range(col):
        k_row = []
        for j in range(col):
            k_row.append(kernel(mat[:, i], mat[:, j], gamma))
        k_mat.append(k_row)
    k_mat = np.array(k_mat)
    ones = np.ones(k_mat.shape) / col
    k_mat = k_mat - ones @ k_mat - k_mat @ ones + ones @ k_mat @ ones
    eig_val, eig_vec = np.linalg.eig(k_mat)
    singular_list = [(np.sqrt(eig_val[i]), eig_vec[:, i] / np.sqrt(eig_val[i]))
        for i in range(len(eig_val))]
    singular_list.sort(key=lambda x: x[0], reverse=True)
    singular_list_k = singular_list[: k]
    sigma = np.diag([i[0] for i in singular_list_k])
    sigma = np.real_if_close(sigma, tol=1)
    v_mat = np.array([list(j[1]) for j in singular_list_k]).T
    v_mat = np.real_if_close(v_mat, tol=1)
    project_mat = sigma @ v_mat.T
    return project_mat


comp_n = 50  # Number of principle components
```

```
i = 0
gamma = 1e-3
img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i), 0)
project_mat = compression_transform(img, gamma, comp_n, rbf)
print(project_mat)
```

A.6.2 图像压缩（逆变换）

```
from sklearn.decomposition import KernelPCA
import numpy as np
import cv2


def evaluate(mat_origin, mat_comp):
    mse = np.mean(np.square(mat_origin - mat_comp))
    psnr = 10 * np.log10(255 * 255 / mse)
    print('MSE:', mse)
    print('PSNR:', psnr)
    return mse, psnr


comp_n = 50  # Number of principle components
gamma = 1e-3
i = 0
img = cv2.imread('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
b, g, r = cv2.split(img)
model = KernelPCA(n_components=comp_n, kernel='rbf', fit_inverse_transform=True,
    gamma=gamma, alpha=5e-3)
reduced_b = model.fit_transform(b)
comp_b = model.inverse_transform(reduced_b)
reduced_g = model.fit_transform(g)
comp_g = model.inverse_transform(reduced_g)
reduced_r = model.fit_transform(r)
comp_r = model.inverse_transform(reduced_r)
img_compressed = cv2.merge([comp_b, comp_g, comp_r])
mse, psnr = evaluate(img, img_compressed)
cv2.imwrite('kpca.jpg', img_compressed)
```

A.7 基于 JPEG 的图像压缩

```
import numpy as np
import os
from PIL import Image
import time


class JPEG:
```

```
    def __init__(self):
        self.dct_a = np.zeros(shape=(8, 8))
        for i in range(8):
            c = 0
            if i == 0:
                c = np.sqrt(1 / 8)
            else:
                c = np.sqrt(2 / 8)
            for j in range(8):
                self.dct_a[i, j] = c * np.cos(np.pi * i * (2 * j + 1) / (2 * 8))
        self.lq = np.array([
            16, 11, 10, 16, 24, 40, 51, 61,
            12, 12, 14, 19, 26, 58, 60, 55,
            14, 13, 16, 24, 40, 57, 69, 56,
            14, 17, 22, 29, 51, 87, 80, 62,
            18, 22, 37, 56, 68, 109, 103, 77,
            24, 35, 55, 64, 81, 104, 113, 92,
            49, 64, 78, 87, 103, 121, 120, 101,
            72, 92, 95, 98, 112, 100, 103, 99,
        ])
        self.cq = np.array([
            17, 18, 24, 47, 99, 99, 99, 99,
            18, 21, 26, 66, 99, 99, 99, 99,
            24, 26, 56, 99, 99, 99, 99, 99,
            47, 66, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
        ])
        self.lt = 0
        self.ct = 1
        self.zig = np.array([
            0, 1, 8, 16, 9, 2, 3, 10,
            17, 24, 32, 25, 18, 11, 4, 5,
            12, 19, 26, 33, 40, 48, 41, 34,
            27, 20, 13, 6, 7, 14, 21, 28,
            35, 42, 49, 56, 57, 50, 43, 36,
            29, 22, 15, 23, 30, 37, 44, 51,
            58, 59, 52, 45, 38, 31, 39, 46,
            53, 60, 61, 54, 47, 55, 62, 63
        ])
        self.zag = np.array([
            0, 1, 5, 6, 14, 15, 27, 28,
            2, 4, 7, 13, 16, 26, 29, 42,
            3, 8, 12, 17, 25, 30, 41, 43,
            9, 11, 18, 24, 31, 40, 44, 53,
            10, 19, 23, 32, 39, 45, 52, 54,
```

```
            20, 22, 33, 38, 46, 41, 55, 60,
            21, 34, 37, 47, 50, 56, 59, 61,
            35, 36, 48, 49, 57, 58, 62, 63
        ])

    def rgb2yuv(self, r, g, b):
        y = 0.299 * r + 0.587 * g + 0.114 * b
        u = -0.1687 * r - 0.3313 * g + 0.5 * b + 128
        v = 0.5 * r - 0.419 * g - 0.081 * b + 128
        return y, u, v

    def padding(self, matrix):
        fh, fw = 0, 0
        if self.height % 16 != 0:
            fh = 16 - self.height % 16
        if self.width % 16 != 0:
            fw = 16 - self.width % 16
        res = np.pad(matrix, ((0, fh), (0, fw)), 'constant', constant_values=(0,
            0))
        return res

    def encode(self, matrix, tag):
        matrix = self.padding(matrix)
        height, width = matrix.shape
        shape = (height // 8, width // 8, 8, 8)
        strides = matrix.itemsize * np.array([width * 8, 8, width, 1])
        blocks = np.lib.stride_tricks.as_strided(matrix, shape=shape, strides=
            strides)
        res = []
        for i in range(height // 8):
            for j in range(width // 8):
                res.append(self.quantize(self.dct_transform(blocks[i, j]).reshape
                    (64), tag))
        return res

    def dct_transform(self, block):
        res = np.dot(self.dct_a, block)
        res = np.dot(res, np.transpose(self.dct_a))
        return res

    def quantize(self, block, tag):
        res = block
        if tag == self.lt:
            res = np.round(res / self.lq)
        elif tag == self.ct:
            res = np.round(res / self.cq)
        return res

```

```
101     def zig_encode(self, blocks):
102         ty = np.array(blocks)
103         tz = np.zeros(ty.shape)
104         for i in range(len(self.zig)):
105             tz[:, i] = ty[:, self.zig[i]]
106         tz = tz.reshape(tz.shape[0] * tz.shape[1])
107         return tz.tolist()
108 
109     def rle_encode(self, blist):
110         res = []
111         cnt = 0
112         for i in range(len(blist)):
113             if blist[i] != 0:
114                 res.append(cnt)
115                 res.append(int(blist[i]))
116                 cnt = 0
117             elif cnt == 15:
118                 res.append(cnt)
119                 res.append(int(blist[i]))
120                 cnt = 0
121             else:
122                 cnt += 1
123         if cnt != 0:
124             res.append(cnt - 1)
125             res.append(0)
126         return res
127 
128     def compress_img(self, filename):
129         image = Image.open(filename)
130         self.width, self.height = image.size
131         image = image.convert('RGB')
132         image = np.asarray(image)
133         r = image[:, :, 0]
134         g = image[:, :, 1]
135         b = image[:, :, 2]
136         y, u, v = self.rgb2yuv(r, g, b)
137         start = time.process_time()
138         y_blocks = self.encode(y, self.lt)
139         u_blocks = self.encode(u, self.ct)
140         v_blocks = self.encode(v, self.ct)
141         y_code = self.rle_encode(self.zig_encode(y_blocks))
142         u_code = self.rle_encode(self.zig_encode(u_blocks))
143         v_code = self.rle_encode(self.zig_encode(v_blocks))
144         end = time.process_time()
145         print("process time:", end - start)
146         buff = 0
147         tfile = os.path.splitext(filename)[0] + ".gpj"
148         if os.path.exists(tfile):
```

```
149             os.remove(tfile)
150         with open(tfile, 'wb') as o:
151             o.write(self.height.to_bytes(2, byteorder='big'))
152             o.flush()
153             o.write(self.width.to_bytes(2, byteorder='big'))
154             o.flush()
155             o.write((len(y_code)).to_bytes(4, byteorder='big'))
156             o.flush()
157             o.write((len(u_code)).to_bytes(4, byteorder='big'))
158             o.flush()
159             o.write((len(v_code)).to_bytes(4, byteorder='big'))
160             o.flush()
161         self.write_file(tfile, y_code, u_code, v_code)
162 
163     def write_file(self, filename, y_code, u_code, v_code):
164         with open(filename, "ab+") as o:
165             buff = 0
166             bcnt = 0
167             data = y_code + u_code + v_code
168             for i in range(len(data)):
169                 if i % 2 == 0:
170                     td = data[i]
171                     for ti in range(4):
172                         buff = (buff << 1) | ((td & 0x08) >> 3)
173                         td <<= 1
174                         bcnt += 1
175                         if bcnt == 8:
176                             o.write(buff.to_bytes(1, byteorder='big'))
177                             o.flush()
178                             buff = 0
179                             bcnt = 0
180                 else:
181                     td = data[i]
182                     vtl, vts = self.vli_transform(td)
183                     for ti in range(4):
184                         buff = (buff << 1) | ((vtl & 0x08) >> 3)
185                         vtl <<= 1
186                         bcnt += 1
187                         if bcnt == 8:
188                             o.write(buff.to_bytes(1, byteorder='big'))
189                             o.flush()
190                             buff = 0
191                             bcnt = 0
192                     for ts in vts:
193                         buff <<= 1
194                         if ts == '1':
195                             buff |= 1
196                         bcnt += 1
```

```
                        if bcnt == 8:
                            o.write(buff.to_bytes(1, byteorder='big'))
                            o.flush()
                            buff = 0
                            bcnt = 0
            if bcnt != 0:
                buff <<= (8 - bcnt)
                o.write(buff.to_bytes(1, byteorder='big'))
                o.flush()
                buff = 0
                bcnt = 0

    def idct_transform(self, block):
        res = np.dot(np.transpose(self.dct_a), block)
        res = np.dot(res, self.dct_a)
        return res

    def i_quantize(self, block, tag):
        res = block
        if tag == self.lt:
            res *= self.lq
        elif tag == self.ct:
            res *= self.cq
        return res

    def i_padding(self, matrix):
        matrix = matrix[:self.height, :self.width]
        return matrix

    def decode(self, blocks, tag):
        tlist = []
        for b in blocks:
            b = np.array(b)
            tlist.append(self.idct_transform(self.i_quantize(b, tag).reshape(8,
                8)))
        height_fill, width_fill = self.height, self.width
        if height_fill % 16 != 0:
            height_fill += 16 - height_fill % 16
        if width_fill % 16 != 0:
            width_fill += 16 - width_fill % 16
        rlist = []
        for hi in range(height_fill // 8):
            start = hi * width_fill // 8
            rlist.append(np.hstack(tuple(tlist[start: start + (width_fill // 8)])
                ))
        matrix = np.vstack(tuple(rlist))
        res = self.i_padding(matrix)
        return res
```

```
243
244     def read_file(self, filename):
245         with open(filename, "rb") as o:
246             tb = o.read(2)
247             self.height = int.from_bytes(tb, byteorder='big')
248             tb = o.read(2)
249             self.width = int.from_bytes(tb, byteorder='big')
250             tb = o.read(4)
251             ylen = int.from_bytes(tb, byteorder='big')
252             tb = o.read(4)
253             ulen = int.from_bytes(tb, byteorder='big')
254             tb = o.read(4)
255             vlen = int.from_bytes(tb, byteorder='big')
256             buff = 0
257             bcnt = 0
258             rlist = []
259             itag = 0
260             icnt = 0
261             vtl, tb, tvtl = None, None, None
262             while len(rlist) < ylen + ulen + vlen:
263                 if bcnt == 0:
264                     tb = o.read(1)
265                     if not tb:
266                         break
267                     tb = int.from_bytes(tb, byteorder='big')
268                     bcnt = 8
269                 if itag == 0:
270                     buff = (buff << 1) | ((tb & 0x80) >> 7)
271                     tb <<= 1
272                     bcnt -= 1
273                     icnt += 1
274                     if icnt == 4:
275                         rlist.append(buff & 0x0F)
276                     elif icnt == 8:
277                         vtl = buff & 0x0F
278                         tvtl = vtl
279                         itag = 1
280                         buff = 0
281                 else:
282                     buff = (buff << 1) | ((tb & 0x80) >> 7)
283                     tb <<= 1
284                     bcnt -= 1
285                     tvtl -= 1
286                     if tvtl == 0 or tvtl == -1:
287                         rlist.append(self.i_vli_transform(vtl, bin(buff)[2:].
                                rjust(vtl, '0')))
288                         itag = 0
289                         icnt = 0
```

```
290         y_dcode = rlist[:ylen]
291         u_dcode = rlist[ylen:ylen+ulen]
292         v_dcode = rlist[ylen+ulen:ylen+ulen+vlen]
293         return y_dcode, u_dcode, v_dcode
294         pass
295 
296     def zag_decode(self, dcode):
297         dcode = np.array(dcode).reshape((len(dcode) // 64, 64))
298         tz = np.zeros(dcode.shape)
299         for i in range(len(self.zag)):
300             tz[:, i] = dcode[:, self.zag[i]]
301         rlist = tz.tolist()
302         return rlist
303 
304     def rle_decode(self, dcode):
305         rlist = []
306         for i in range(len(dcode)):
307             if i % 2 == 0:
308                 rlist += [0] * dcode[i]
309             else:
310                 rlist.append(dcode[i])
311         return rlist
312 
313     def decompress_image(self, filename):
314         y_dcode, u_dcode, v_dcode = self.read_file(filename)
315         y_blocks = self.zag_decode(self.rle_decode(y_dcode))
316         u_blocks = self.zag_decode(self.rle_decode(u_dcode))
317         v_blocks = self.zag_decode(self.rle_decode(v_dcode))
318         y = self.decode(y_blocks, self.lt)
319         u = self.decode(u_blocks, self.ct)
320         v = self.decode(v_blocks, self.ct)
321         r = (y + 1.402 * (v - 128))
322         g = (y - 0.34414 * (u - 128) - 0.71414 * (v - 128))
323         b = (y + 1.772 * (u - 128))
324         r = Image.fromarray(r).convert('L')
325         g = Image.fromarray(g).convert('L')
326         b = Image.fromarray(b).convert('L')
327         image = Image.merge("RGB", (r, g, b))
328         # image.save("jpeg_rst.jpg", "jpeg")
329         # image.show()
330 
331     def vli_transform(self, n):
332         ts, tl = 0, 0
333         if n > 0:
334             ts = bin(n)[2:]
335             tl = len(ts)
336         elif n < 0:
337             tn = (-n) ^ 0xFFFF
```

```
            tl = len(bin(-n)[2:])
            ts = bin(tn)[-tl:]
        else:
            tl = 0
            ts = '0'
        return (tl, ts)

    def i_vli_transform(self, tl, ts):
        if tl != 0:
            n = int(ts, 2)
            if ts[0] == '0':
                n = n ^ 0xFFFF
                n = int(bin(n)[-tl:], 2)
                n = -n
        else:
            n = 0
        return n


if __name__ == '__main__':
    jpeg = JPEG()
    i = 0
    jpeg.compress_img('images/agricultural/agricultural{0:0>2d}.tif'.format(i))
    jpeg.decompress_image('images/agricultural/agricultural{0:0>2d}.gpj'.format(i
        ))
```