《大数据机器学习》第 3 次实验

姓名：刘培源　　学号：2023214278

题目 1：

熟悉 SVD 的原理。

答： 奇异值分解（SVD）是线性代数中的一种重要分解方法，允许将任意一个 $m \times n$ 矩阵 A 分解为三个矩阵的乘积：

$$A = U\Sigma V^T$$

在这个分解中：

1. U 是一个 $m \times m$ 的酉矩阵（即满足 $U^*U = I$），其列向量称为左奇异向量，其形式如下：

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1m} \\ u_{21} & u_{22} & \cdots & u_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m1} & u_{m2} & \cdots & u_{mm} \end{bmatrix}$$

2. Σ 是一个 $m \times n$ 的矩阵，主要对角线上的元素是奇异值，按非增顺序排列。这些奇异值是矩阵 A 的奇异谱，提供了关于 A 的秩和“能量”分布等信息，其形式如下：

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{\min(m,n)} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

3. V^T 是一个 $n \times n$ 的酉矩阵 V 的转置，其列向量称为右奇异向量，其形式如下：

$$V^T = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{nn} \end{bmatrix}$$

SVD 在数据降维、矩阵近似、信号处理等领域中有广泛应用。通过选择最大的几个奇异值及其对应的奇异向量，可以有效近似原始矩阵，实现数据压缩和噪声过滤。这种技术在图像处理和推荐系统等领域尤为重要。

题目 2：

SVD 图片压缩。根据提供的代码，理解代码的计算流程，将代码补充完整，同时构建自己的 SVD 算法。将自己的算法与 numpy 提供的接口进行对比，分析自己的算法的压缩效率和压缩效果。

答： 我的 “ImageCompressor” 类的代码如下（完整的代码及注释可以在代码文件中查看）：

```python
class ImageCompressor:
    def __init__(self, image_path, k):
        self.image_path = image_path
        self.k = k

    def compress_image(self):
        reshaped_image = self.image_array.reshape(self.shape[0], -1)
        U, S, Vt = np.linalg.svd(reshaped_image, full_matrices=False)

        # 选取前k个奇异值和特征来重构图片，并恢复成与原图片相同大小
        U_k, S_k, Vt_k = U[:, :self.k], np.diag(S[:self.k]), Vt[:self.k, :]

        # 重构图片
        compressed_image = np.dot(np.dot(U_k, S_k), Vt_k)
        compressed_image = compressed_image.reshape(self.shape)

        # 将图片的像素值限制在0-255之间
        compressed_image = np.clip(compressed_image, 0, 255)
        compressed_image = compressed_image.astype('uint8')

        return compressed_image

    def svd(self, X, full_matrices=False):
        X = X.astype(float)

        #计算X的特征值和特征向量
        eigenvalues, V = np.linalg.eig(X.T @ X)

        # 对特征值进行排序，从大到小
        sorted_index = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[sorted_index]
        V = V[:, sorted_index]

        # 计算奇异值和左右奇异向量
        singular_values = np.sqrt(eigenvalues)
        min_dim = min(X.shape)
        singular_values = singular_values[:min_dim]

        if full_matrices:   # full_matrices=True时，U的大小为m×m
            sigma = np.zeros(X.shape)
            np.fill_diagonal(sigma, singular_values)
            U = np.dot(X, np.dot(V, np.linalg.pinv(sigma)))
            return U, singular_values, V.T
        else:    # full_matrices=False时，U的大小为m×min(m,n)
            sigma = np.diag(singular_values[:min_dim])
            U = np.dot(X, np.dot(V[:, :min_dim], np.linalg.pinv(sigma)))
            return U, singular_values, V[:, :min_dim].T

    def compress_image_yourself(self):
        reshaped_image = self.image_array.reshape(self.shape[0], -1)
        U, S, Vt = self.svd(reshaped_image, full_matrices=False)

        # ... 与compress_image函数后续操作一致

        return compressed_image
```

要想用 SVD 实现图片压缩，对于原始图片 image，我们可以首先调用 “np.linalg.svd”

来对 image 进行奇异值分解得到 U, Σ, V^T，然后选取前 k 大的特征值和特征向量，再将他们按照如下公式：

$$\text{image}_{\text{compress}} = U[:,:k] \cdot \Sigma[:k] \cdot V^T[:k,:]$$

得到压缩后的图片。

我还自己实现了 SVD 来替代了 “np.linalg.svd”，代码在上面的 23-47 行。具体操作是用 “np.linalg.eig” 求解了矩阵（$X^T \times X$）的特征根（Σ^2）和特征向量（V^T），然后对特征根进行从大到小的排列，最后根据 X, V, Σ 来算出 U 即可。这个方法主要是借用了 “np.linalg.eig” 来求解矩阵的特征根，纯手动实现高维矩阵的特征根近似求解是一个困难的问题。

Note

尽管我没有手动实现矩阵的特征根求解，但是我模仿了 “np.linalg.svd” 中的 full_matrices 的行为。具体来说，当 full_matrices=True 的时候，会返回 “完全” 奇异值分解。这意味着 U 是一个 $m \times m$ 的矩阵，V^* 是一个 $n \times n$ 的矩阵。而当 full_matrices=Fasle 的时候，会返回 “紧凑” 形式的 SVD，其中 U 是一个 $m \times \min(m,n)$ 的矩阵，V^* 是一个 $\min(m,n) \times n$ 的矩阵，可以节省内存与提高效率。

我实验了 $k = 10$ 下的图片压缩效果，如图2所示。我还通过 “np.allclose”‘来验证了我自己实现的 SVD 与 numpy 的 SVD 的压缩结果是一摸一样的。

(a) 原始输入图片　(b) numpy 的 svd 的压缩图　(c) 自己实现的 svd 的压缩图

图 1: $k = 10$ 的情况下图片压缩效果对比。

压缩效率对比。我统计了用 numpy 方法和自己方法进行奇异值分解的程序运行时间，结果如表1所示。可以看到 numpy 的实现还是具有效率优势，分析原因可能是其内部对矩阵的特征根求解做了更进一步的效率优化。

	numpy	自己实现
时间	$0.238s$	$0.610s$

表 1: 对比 numpy 和自己实现的 SVD 的运行时间。

题目 3： 对 SVD 压缩选取不同的参数（比如选取的奇异值个数），比较不同情况下的压缩效果。

答：

我探究了 $k = 10, 50, 100, 200, 300, 400$ 下的压缩效果，选取的评估指标是峰值信噪比（PSNR）和结构相似性指数（SSIM），他们的值越大，代表压缩图像与原本相似度更好，具体介绍如下：

1. 峰值信噪比（PSNR）是一种评估图像重建或压缩质量的常用指标。PSNR 通过比较原始图像和压缩或重建后的图像的像素差异来衡量质量，定义如下：

$$\text{PSNR} = 20 \log_{10} \left(\frac{\text{MAX}_I}{\sqrt{\text{MSE}}} \right)$$

   其中，MSE 是均方误差，计算原始图像 I 和重建图像 K 之间的像素差异平方和的平均值，定义为：

$$\text{MSE} = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2$$

   MAX_I 是图像可能的最大像素值。例如，对于 8 位图像，MAX_I 为 255。

2. 结构相似性指数（SSIM）是一种衡量两个图像相似度的指标，特别强调了图像的结构信息。SSIM 的值在 -1（无相似性）和 1（完全相似）之间变化。SSIM 的定义如下：

$$\text{SSIM}(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

   其中 x 和 y 是两个窗口，分别在原始图像和比较图像中。μ_x, μ_y 分别是 x, y 的平均值。σ_x^2, σ_y^2 是 x, y 的方差。σ_{xy} 是 x 和 y 的协方差。C_1 和 C_2 是用来维持稳定性的小常数，通常取 $C_1 = (0.01L)^2$ 和 $C_2 = (0.03L)^2$，其中 L 是图像的动态范围。

我的代码如下：

```python
ks = [10, 50, 100, 200, 300, 400]

for k in ks:
    image_path = "input.jpg"
    np_output_path = f"np_output_k{k}.jpg"
    self_output_path = f"self_output_k{k}.jpg"

    compressor = ImageCompressor(image_path, k)
    compressor.load_image()

    compressor.save_compressed_image(np_output_path)
    compressor.save_compressed_image(self_output_path, your_self=True)

    np_compressed_image = plt.imread(np_output_path)
    self_compressed_image = plt.imread(self_output_path)

    print(f"比较两种压缩方法的结果是否相同：{np.allclose(np_compressed_image, self_compressed_image)}")

    print(f"在k={k}时，压缩的PSNR值为：{PSNR(compressor.image_array, np_compressed_image)}, \
                        SSIM值为：{SSIM(compressor.image_array, np_compressed_image)}")
```

压缩结果如表2所示。可以看到，随着选择的特征值的个数增加，压缩后图像与原始图像的相似度在逐渐提高。

	PSNR↑	SSIM↑
$k = 10$	29.58990	0.92605
$k = 50$	32.38018	0.98531
$k = 100$	35.11647	0.99476
$k = 200$	37.95742	0.99778
$k = 300$	38.17244	0.99790
$k = 400$	38.18160	0.99790

表 2: 压缩结果随着不同 k 值的变化结果。

可视化结果如下：

(a) 原始输入图片

(b) numpy 的 svd 的压缩图

(c) 自己实现的 svd 的压缩图

图 2: $k = 10$ 的情况下图片压缩效果对比。

(a) 原始输入图片

(b) numpy 的 svd 的压缩图

(c) 自己实现的 svd 的压缩图

图 3: $k = 50$ 的情况下图片压缩效果对比。

(a) 原始输入图片

(b) numpy 的 svd 的压缩图

(c) 自己实现的 svd 的压缩图

图 4: $k = 100$ 的情况下图片压缩效果对比。

(a) 原始输入图片

(b) numpy 的 svd 的压缩图

(c) 自己实现的 svd 的压缩图

图 5: $k = 200$ 的情况下图片压缩效果对比。

(a) 原始输入图片

(b) numpy 的 svd 的压缩图

(c) 自己实现的 svd 的压缩图

图 6: $k = 300$ 的情况下图片压缩效果对比。

(a) 原始输入图片

(b) numpy 的 svd 的压缩图

(c) 自己实现的 svd 的压缩图

图 7: $k = 400$ 的情况下图片压缩效果对比。