

重慶大學

数学建模校内竞赛论文



论文题目:

组号:

成员:

选题:

姓名	学院	年级	专业	学号	联系电话	数学分析	高等代数	微积分	高等数学	线性代数	概率统计	数学实验	数学模型	CET4	CET6

日期

摘 要

待补全

关键词： 少样本分类；关系建模；对比学习；语义信息表示

Key words:

目 录

摘 要.....I

主要符号对照表.....IV

1 问题重述.....1

 1.1 问题背景与意义.....1

 1.2 问题提出与研究内容.....1

 1.2.1 问题一.....1

2 模型假设.....2

3 理论基础.....3

 3.1 CIE1931 标准色度观察者与光谱三刺激值.....3

 3.2 CIEXYZ 颜色空间.....3

 3.3 CIELab 颜色空间.....3

 3.4 CIE1931xy 色度图与色域表示.....4

 3.5 CIEDE2000 色差公式.....4

 3.6 颜色感知与颜色空间.....5

 3.7 色域与色域映射.....6

 3.8 颜色差异度量 (ΔE).....6

 3.9 伽马校正理论.....6

 3.9.1 伽马响应模型.....7

 3.9.2 伽马参数估计.....7

 3.9.3 伽马校正变换.....7

 3.9.4 伽马校正的物理意义.....7

4 模型建立与求解.....9

 4.1 问题 1：颜色空间转换模型.....9

 4.1.1 模型建立与求解.....9

 4.1.2 问题一结果分析.....10

 4.2 问题 2：四通道到五通道颜色转换模型.....12

 4.2.1 问题分析与建模目标.....12

 4.2.2 神经网络模型设计.....13

 4.2.3 损失函数设计.....13

 4.2.4 数据生成与训练策略.....14

4.2.5 模型求解和结果分析.....15

4.3 问题 3：LED 显示器颜色校正模型19

4.3.1 模型建立流程20

4.3.2 模型求解策略21

4.3.3 模型验证与分析.....23

5 模型评价与推广27

5.1 主要结论27

5.2 模型优点27

5.3 不足与改进方向.....27

参考文献.....28

附 录.....29

A. 问题 1 使用代码.....29

B. 问题 2 使用代码.....36

C. 问题 3 使用代码.....47

D. 像素数据集.....56

主要符号对照表

M	颜色空间转换矩阵 $M \in \mathbb{R}^{3 \times 3}$
\mathbf{c}_i	第 i 个 BT.2020 RGB 颜色样本 $\mathbf{c}_i \in [0, 1]^3$
\mathbf{c}'_i	映射后的显示屏 RGB 颜色样本
$M_{BT \rightarrow XYZ}$	BT.2020 色域到 XYZ 色彩空间的转换矩阵
$M_{DP \rightarrow XYZ}$	显示屏色域到 XYZ 色彩空间的转换矩阵
L^*	CIE Lab 色彩空间中的明度分量
a^*	CIE Lab 色彩空间中的红绿轴分量
b^*	CIE Lab 色彩空间中的黄蓝轴分量
ΔE_{00}	CIE DE2000 色差公式
X, Y, Z	CIE XYZ 色彩空间的三刺激值
x, y	CIE 1931 色度图坐标
$\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$	CIE 1931 标准色度观察者匹配函数
$S(\lambda)$	光谱功率分布函数
\mathbf{X}	神经网络输入数据矩阵, 4 通道 RGBV
\mathbf{Y}	神经网络目标输出数据矩阵, 5 通道 RGBCX
W	神经网络权重矩阵
\mathbf{b}	神经网络偏置向量
$f(\cdot)$	激活函数
L_{MSE}	均方误差损失函数
$L_{\Delta E_{2000}}$	基于 ΔE_{2000} 的感知误差损失函数
L_{total}	混合损失函数
α, β	损失函数权重参数
η	学习率
\mathbf{x}_i	差分进化算法中第 i 个个体
NP	差分进化算法种群大小
F	差分进化算法变异因子
CR	差分进化算法交叉概率
\mathbf{v}_i	差分进化算法变异向量
\mathbf{u}_i	差分进化算法试验向量
G	差分进化算法迭代代数
γ_c	第 c 个颜色通道的伽马值
S_c	第 c 个颜色通道的比例因子
$I_{\text{meas},c}$	第 c 个颜色通道的测量值
$I_{\text{target},c}$	第 c 个颜色通道的目标值
\mathbf{x}	线性 RGB 向量 $\mathbf{x} = [R_\ell, G_\ell, B_\ell]^T$
\mathbf{y}	校正后 RGB 向量

L_{DE}	基于 CIE ΔE_{00} 的色差损失
L_{reg}	正则化损失项
L_{det}	行列式惩罚项
$\mathcal{L}(M, \mathbf{b})$	LED 颜色校正总目标函数
$\lambda_1, \lambda_2, \lambda_3$	损失函数权重参数
ϵ	行列式惩罚阈值
$\text{clip}(\cdot, [0, 1])$	数值裁剪函数, 限制在 $[0, 1]$ 范围
$\ \cdot\ _F$	Frobenius 范数
$\ \cdot\ _2$	欧几里得范数
$\det(\cdot)$	矩阵行列式
I_3	3×3 单位矩阵
$\mathbf{1}(\cdot)$	指示函数
\vee	逻辑或运算符
N	样本数量
k	样本索引
c	颜色通道索引, $c \in \{R, G, B\}$
popsize	差分进化算法种群大小
maxiter	最大迭代次数
D65	标准光源 D65 白点
δ	CIE Lab 转换中的阈值参数, $\delta = 6/29$

1 问题重述

$d^{[1]}$

1.1 问题背景与意义

随着当下显示器技术的发展，超高清技术、HDR 技术的出现，显示器设备对色彩表现力的要求越来越高。然而，由于图像采集设备与图像现实设备二者对色彩的感知和还原能力存在差异，导致视频源中的色彩信息往往无法在显示设备上完美复现。在当前超高清显示器的需求日益增长的背景下，如何在有限色域的显示器中还原视频源的色彩，已成为高性能显示设备设计的关键难题。

国际照明委员会建立了标准色度学系统，这为颜色表达和转换提供了统一的数学框架。比如 CIE1931 色度图，可以将不同设备的色域覆盖可视化。BT2020 色彩空间是一种标准的高清视频源的三基色色空间。BT2020 具有更广的色域范围，通常用于高动态范围视频何超高清电视的显示。而现实中普通显示屏色彩空间，诸如 sRGB、NTSC 等通常色域更小。这通常会导致部分高色彩饱和度区域无法准确重建，从而形成色彩损失问题。

针对上述问题，工业界提出多通道拓展方案，将视频源引入第四个颜色通道 V (RGBV) 拓宽了记录色域，而显示设备则拓展为五通道 (RGBCX) 以提升色彩重现能力。因而如何设计从高色域空间到显示器色域的映射函数，使得色彩失真最小，是提升色彩显示能力的关键任务。

此外被广泛使用的 LED 显示器，因其本身的制造差异、驱动电路的非线性影响等因素会导致整屏颜色显示不一致。这导致了显示效果的非一致性会严重影响视觉体验。因此基于颜色空间转换与匹配原理，合理构建映射函数和校正策略，对 LED 像素点颜色进行精细调控，从而实现整屏一致性的色彩校正，已成为提升 LED 显示品质的重要手段。

1.2 问题提出与研究内容

1.2.1 问题一

本问题的核心在于实现不同色域之间的映射。BT2020 色域更广，而 sRGB 色域相对较小。二者在色度坐标、亮度范围等方面存在较大差异，直接映射会导致显示器难以还原视频源的颜色，进而损失色彩，还会导致失真、亮度饱和度损失等问题。因此需要定义合适的转换损失函数，减小色彩损失。因此在映射过程中应当选择合适的损失函数，保证转换后的色彩贴合人眼视觉特性，提高感知效果。选择损失函数后还应当采用梯度下降法或基于样本的非线性最小二乘法进行求解。

2 模型假设

3 理论基础

为了便于对后续视频源 BT.2020 色域与普通显示屏 RGB 色域之间映射关系的分析，我们首先引入标准色度系统的数学模型，对常见色彩空间进行建模表示。这些空间构成了本问题中色彩转换和损失评估的基础框架。

3.1 CIE1931 标准色度观察者与光谱三刺激值

CIE 1931 是由国际照明委员会（CIE）于 1931 年定义的色彩模型，其核心在于基于实验测量建立的“标准色度观察者”响应曲线。这一模型通过三条匹配函数 $\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$ 将任意波长下的光谱功率分布（SPD）映射为三刺激值（Tristimulus Values）：

$$X = \int_{\lambda} S(\lambda) \bar{x}(\lambda) d\lambda, \quad Y = \int_{\lambda} S(\lambda) \bar{y}(\lambda) d\lambda, \quad Z = \int_{\lambda} S(\lambda) \bar{z}(\lambda) d\lambda \quad (3.1)$$

3.2 CIE XYZ 颜色空间

CIE XYZ 是一个以三刺激值为基础的线性色彩空间，被视为“设备无关”的色彩表示方式。其三个分量 (X, Y, Z) 分别对应红、绿、蓝三种感知通道。 Y 分量也通常用作**亮度（Luminance）**的代表。该空间是许多其他色彩空间（如 Lab、sRGB、BT.2020）的中间标准基础。通常不同色域之间的转换以此为中介。

3.3 CIE Lab 颜色空间

CIE Lab 空间是基于 CIE XYZ 空间定义的感知均匀色彩空间，能够更好地符合人眼对颜色差异的敏感性。其由以下三个分量构成：

$$L^*, \quad a^*, \quad b^P_* \quad (3.2)$$

其中， L^* 代表明度， a^* 代表红绿轴， b^* 代表黄蓝轴。具体变换公式如下（以 D65 白点为例）：

$$f(t) = \begin{cases} t^{\frac{1}{3}} & t > \delta^3 \\ \frac{t}{3\delta^2} + \frac{4}{29} & t \leq \delta^3 \end{cases}, \quad \delta = \frac{6}{29} \quad (3.3)$$

$$L^* = 116f\left(\frac{Y}{Y_n}\right) - 16, \quad a^* = 500\left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right], \quad b^* = 200\left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right] \quad (3.4)$$

其中 (X_n, Y_n, Z_n) 为参考白点（如 D65）的三刺激值。

3.4 CIE1931xy 色度图与色域表示

CIEXYZ 空间中颜色可以通过如下变换得到色度图中的坐标：

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z} \quad (3.5)$$

该色度图表示了所有可见光的二维投影范围，设备的色域可以通过其三基色的 (x, y) 点连线形成三角形表示。色域越大，所能表示的颜色越丰富。该图是色彩匹配与色彩损失分析的重要工具。

3.5 CIEDE2000 色差公式

为了更精确的对问题进行建模并且便于后续损失函数以及差分进化算法的实现，我们将题目中的 BT.2020 颜色空间以及显示屏的颜色空间从 xy 色度坐标转换为 XYZ 颜色空间，再利用 Lab 颜色空间公式转换为 (L^*, a^*, b^*) 。最后计算 ΔE_{00} 损失值。

在将 BT.2020 高清视频源的色彩空间映射至普通显示屏 RGB 色域时，由于显示设备色域较小，无法完整覆盖原始色域，导致部分颜色无法被准确再现。因此，我们需要设计一个合理的**色彩转换映射矩阵** $M \in \mathbb{R}^{3 \times 3}$ ，以最小化从 BT.2020 色域到显示屏色域的映射过程中所产生的**主观感知误差**。

为度量这一色彩差异，应选择符合人眼视觉感知的度量方式。传统的欧几里得差异（如 RGB 或 XYZ 空间中的 L2 距离）不能很好地反映颜色感知误差。我们引入国际照明委员会（CIE）推荐的 ΔE_{00} 作为感知误差的度量函数。

对任意两个颜色在 Lab 空间中的向量：

$$Lab_1 = (L_1^*, a_1^*, b_1^*), \quad Lab_2 = (L_2^*, a_2^*, b_2^*) \quad (3.6)$$

ΔE_{00} 的计算公式如下：

$$\Delta E_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \cdot \left(\frac{\Delta C'}{k_C S_C}\right) \cdot \left(\frac{\Delta H'}{k_H S_H}\right)} \quad (3.7)$$

(1) 明度差与平均明度

$$\Delta L' = L_2^* - L_1^*, \quad \bar{L} = \frac{L_2^* + L_1^*}{2} \quad (3.8)$$

(2) 色度差与平均色度

$$C_1 = \sqrt{a_1^{*2} + b_1^{*2}}, \quad C_2 = \sqrt{a_2^{*2} + b_2^{*2}}, \quad \Delta C' = C_2 - C_1, \quad \bar{C} = \frac{C_1 + C_2}{2} \quad (3.9)$$

(3) 色相角差与平均色相角

$$\begin{aligned}
 h_1 &= \arctan 2(b_1^*, a_1^*), \quad h_2 = \arctan 2(b_2^*, a_2^*) \\
 \Delta h' &= h_2 - h_1, \quad \Delta H^1 = 2\sqrt{C_1 C_2} \sin\left(\frac{\Delta h'}{2}\right) \\
 \bar{h} &= \begin{cases} \frac{h_1+h_2}{2}, & |h_1 - h_2| > 180^\circ \\ \frac{h_1+h_2+360^\circ}{2}, & |h_1 - h_2| \leq 180^\circ \end{cases}
 \end{aligned} \tag{3.10}$$

(4) 调整因子

$$\begin{aligned}
 G &= 0.5 \left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \\
 T &= 1 - 0.17 \cos(\bar{h} - 30^\circ) + 0.24 \cos(2\bar{h}) + 0.32 \cos(3\bar{h} + 6^\circ) - 0.20 \cos(4\bar{h} - 63^\circ)
 \end{aligned} \tag{3.11}$$

(5) 权重因子

$$S_L = 1 + \frac{0.015(L - 50)^2}{\sqrt{20 + (\bar{L} - 50)^2}}, \quad S_C = 1 + 0.045\bar{C}, \quad S_H = 1 + 0.015\bar{C}T \tag{3.12}$$

(6) 旋转补偿因子

$$R_T = -\sin(2\Delta\theta) \cdot R_C, \quad \Delta\theta = 30 \exp\left\{-\left(\frac{\bar{h} - 275^\circ}{25}\right)^2\right\}, \quad R_C = 2\sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}} \tag{3.13}$$

其中: $\Delta L'$: 明度差, $\Delta C'$: 色度差, $\Delta H'$: 色相差, S_L, S_C, S_H : 感知缩放因子, $k_L = k_C = k_H = 1$: 常用单位权重。

由上述公式, 可以计算出两个 CIELab 值的色差。该函数对人眼感知差异具有良好拟合性能, 因此被广泛用于图像质量、颜色匹配等领域。

3.6 颜色感知与颜色空间

人类对颜色的感知是一个复杂的生理和心理过程。为了量化和描述颜色, 引入了颜色空间的概念。常见的颜色空间包括:

- **RGB (Red, Green, Blue):** 基于三原色加法混色的颜色模型, 常用于显示设备和图像输入设备。然而, RGB 并非感知均匀, 即欧氏距离不直接对应人眼感知的颜色差异。
- **XYZ (CIE 1931 XYZ):** 国际照明委员会 (CIE) 定义的一种基于人眼视觉生理特性的颜色空间。它涵盖了人眼可见的所有颜色, 且与设备无关。其分量 X, Y, Z 分别对应于光谱在人眼视锥细胞响应曲线下的积分。 Y 分量通常表

示亮度信息。

- **Lab (CIE L*a*b*)**: 一种感知均匀的颜色空间，从 XYZ 空间推导而来。 L^* 表示亮度，从黑到白； a^* 表示从绿到红的颜色信息； b^* 表示从蓝到黄的颜色信息。在 Lab 空间中，两点之间的欧氏距离与人眼感知的颜色差异近似成正比。

3.7 色域与色域映射

色域 (Gamut) 是指一个颜色系统或设备能够显示或捕捉的颜色范围。在 CIE 1931 色度图上，色域通常由其基色（原色）的色度坐标连接形成的多边形表示。色域映射 (Gamut Mapping) 是指将一个色域的颜色转换到另一个色域的过程，旨在最小化颜色失真，尤其是在目标色域小于源色域时。

传统的显示系统通常采用三基色 (RGB) 来显示颜色。然而，为了更广阔的色域和更丰富的色彩表现，多基色显示技术（如本问题中的五通道 LED 显示屏）正在兴起。这些系统通过增加额外的基色来扩展其可显示的颜色范围。

3.8 颜色差异度量 (ΔE)

为了量化两种颜色之间人眼感知的差异，引入了颜色差异度量 ΔE (Delta E)。其中， ΔE_{2000} (CIE DE2000) 是目前最广泛接受的颜色差异公式，它在 Lab 空间的基础上进行了修正，以更好地反映人眼的非线性颜色感知特性，尤其是在中性色、亮度和色调方面。 ΔE_{2000} 值越小，表示两种颜色感知差异越小。

其公式如下：

$$\Delta E_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \left(\frac{\Delta C'}{k_C S_C}\right) \left(\frac{\Delta H'}{k_H S_H}\right)}$$

其中， $\Delta L', \Delta C', \Delta H'$ 分别是修正后的亮度、彩度和色调差值； S_L, S_C, S_H 是权重函数，用于调整不同区域的感知均匀性； R_T 是旋转项，用于处理蓝色区域的特殊感知。 k_L, k_C, k_H 是参数，通常取 1。

3.9 伽马校正理论

伽马校正 (Gamma Correction) 是数字图像处理中的重要概念，用于补偿显示设备的非线性响应特性。在理想情况下，显示设备的输出亮度应与输入信号成线性关系，但实际显示设备（如 CRT 显示器、LCD 显示器、LED 显示屏等）通常具有非线性的响应曲线。

3.9.1 伽马响应模型

对于显示设备的每个颜色通道 $c \in \{R, G, B\}$ ，其响应特性可以用伽马函数建模：

$$I_{\text{output},c} = S_c \cdot (I_{\text{input},c})^{\gamma_c} \quad (3.14)$$

其中：

- $I_{\text{input},c}$ ：输入信号强度（归一化到 $[0,1]$ 范围）
- $I_{\text{output},c}$ ：实际输出亮度（归一化到 $[0,1]$ 范围）
- γ_c ：伽马值，描述非线性程度
- S_c ：比例因子，用于调整整体亮度水平

3.9.2 伽马参数估计

给定测量数据 $I_{\text{meas},c}$ 和目标数据 $I_{\text{target},c}$ ，可以通过对数线性回归估计伽马参数：

$$\log(I_{\text{meas},c}) = \gamma_c \log(I_{\text{target},c}) + \log(S_c) \quad (3.15)$$

这可以转化为标准的线性回归问题：

$$\mathbf{y} = \mathbf{A}\boldsymbol{\theta} + \boldsymbol{\epsilon} \quad (3.16)$$

其中 $\mathbf{y} = \log(I_{\text{meas},c})$ ， $\mathbf{A} = [\log(I_{\text{target},c}), 1]$ ， $\boldsymbol{\theta} = [\gamma_c, \log(S_c)]^T$ 。

3.9.3 伽马校正变换

基于估计的伽马参数，可以定义前向和反向伽马校正变换：

前向变换（编码）：

$$I_{\text{encoded}} = \text{clip}((I_{\text{linear}} \cdot S)^{\gamma}, [0, 1]) \quad (3.17)$$

反向变换（线性化）：

$$I_{\text{linear}} = \text{clip}((I_{\text{encoded}})^{1/\gamma} / S, [0, 1]) \quad (3.18)$$

其中 $\text{clip}(\cdot, [0, 1])$ 函数确保输出值在有效范围内。

3.9.4 伽马校正的物理意义

伽马校正的物理意义体现在以下几个方面：

- **设备特性补偿：**不同显示设备具有不同的伽马值，通过校正可以实现设备间的颜色一致性
- **感知均匀性：**人眼对亮度的感知是非线性的，适当的伽马校正可以更好地匹配人眼感知

- **动态范围优化：**伽马校正可以优化有限位深下的颜色表示，减少量化误差

在 LED 显示器颜色校正中，伽马校正是实现精确颜色还原的关键步骤，需要与线性矩阵变换相结合，形成完整的颜色校正流程。

4 模型建立与求解

4.1 问题 1：颜色空间转换模型

4.1.1 模型建立与求解

为求解 BT.2020 空间到显示屏 RGB 空间的最优线性映射矩阵 $M \in \mathbf{R}^{3 \times 3}$ ，我们采样一组代表性 BT.2020 RGB 样本 $\{c_i\}_{i=1}^N \in [0, 1]^3$ ，其色彩向量经过如下映射：

$$c'_i = M \cdot C_i \quad (4.1)$$

然后分别映射至 CIELab 空间，并计算感知误差：

$$L(M) = \frac{1}{N} \sum_{i=1}^N \Delta E_{00}(Lab(M_{BT \rightarrow XYZ} \cdot c_i), Lab(M_{DP \rightarrow XYZ} \cdot (M \cdot c_i))) \quad (4.2)$$

优化目标为：

$$\min_{M \in \mathbf{R}^{3 \times 3}} L(M) \quad (4.3)$$

为求解上述非线性、不可导且可能存在多个局部极小值的优化问题，我们引入差分进化（Differential Evolution, DE）算法。DE 是一种基于种群的全局优化方法，具有较强的鲁棒性与跳出局部最优的能力。

(1) 差分进化算法设计

① 参数编码与搜索空间

将 $M \in (\mathbf{R}^{3 \times 3})$ 展开为 9 维向量 $\mathbf{x} \in \mathbf{R}^9$ ，并定义搜索空间边界为：

$$x_j \in [-2, 2]m \quad j = 1, \dots, 9 \quad (4.4)$$

② 种群初始化

生成 NP 个个体 $\mathbf{x}_i^{(0)} \in \mathbf{R}^9$ ：

$$x_{i,j}^{(0)} = l_j + r_{i,j} \cdot (u_j - l_j), \quad r_{i,j} \sim \mathcal{U}(0, 1) \quad (4.5)$$

③ 变异操作

对第 i 个个体，在不同个体中随机选择 $\mathbf{x}_{r1}, \mathbf{x}_{r2}, \mathbf{x}_{r3}$ ，构造差分向量：

$$\mathbf{v}_i = \mathbf{x}_{r1} + F \cdot (\mathbf{x}_{r2} - \mathbf{x}_{r3}) \quad (4.6)$$

其中 $F \in (0, 2)$ 是差分缩放因子，控制探索强度。

④ 交叉操作

构造试验个体 \mathbf{u}_i :

$$u_{i,j} = \begin{cases} v_{i,j}, & \text{rand}_j < CR \text{ or } j = j_{rand} \\ x_{i,j}^t, & \text{otherwise} \end{cases} \quad (4.7)$$

其中 $CR \in [0, 1]$ 为交叉概率, j_{rand} 确保至少一维来自 \mathbf{v}_i 。

⑤ 选择操作

通过目标函数比较试验解与当前个体, 选择保留更优者:

$$\mathbf{x}_i^{t+1} = \begin{cases} \mathbf{u}_i, & L(\mathbf{u}_i) < L(\mathbf{x}_i^{(t)}) \\ \mathbf{x}_i^{(t)}, & \text{otherwise} \end{cases} \quad (4.8)$$

综上所述, 为优化色彩转换矩阵 M , 本文选用差分进化算法 (DE)。该方法将 M 参数化为 9 维向量, 并在预设的搜索空间边界内进行优化。通过其经典的种群初始化、变异、交叉及选择等核心操作, DE 算法能够迭代地搜寻旨在最小化以 ΔE_{00} 度量的感知色彩差异的解。鉴于目标函数的非线性、不可导以及可能存在多个局部极小值的特性, DE 算法的全局优化能力和鲁棒性, 使其成为获取高质量色彩映射的有效计算途径。

4.1.2 问题一结果分析

为实现 BT.2020 色域向目标显示屏 RGB 色域的最优映射, 本文构建了感知误差最小化的优化模型, 目标为在 CIELab 空间中最小化 ΔE_{00} 感知色差。我们采样了多个 BT.2020 RGB 颜色点, 并通过线性映射矩阵 $M \in \mathbb{R}^{3 \times 3}$ 变换后, 转化至目标显示屏空间, 再经过标准变换矩阵应设至 XYZ、CIELab 空间, 并利用 ΔE_{00} 公式计算感知误差。

在模型求解过程中, 本文采用了差分进化 (Differential Evolution, DE) 优化方法, 对初始映射矩阵进行迭代寻优。为验证我们模型的稳定性, 并提供更可靠的性能评价, 我们执行了 50 轮随机优化, 并统计其性能指标。主要分析结果如下:

(1) 感知误差分布分析

① ΔE_{00} 损失值统计

图4.1显示了在 50 次独立优化实验中, 各次优化所达成的最终 ΔE_{00} 损失值分布情况。其中最大值为 1.0183, 这表明映射结果在感知层面极为接近参考目标。均值为 0.0744, 标准差为 0.2083。这表明该基于 ΔE_{00} 损失函数的差分进化算法在不同采样条件下都能稳定收敛于较小的感知误差区域, 并且具有良好的泛化性能以及良好的稳定性和鲁棒性。

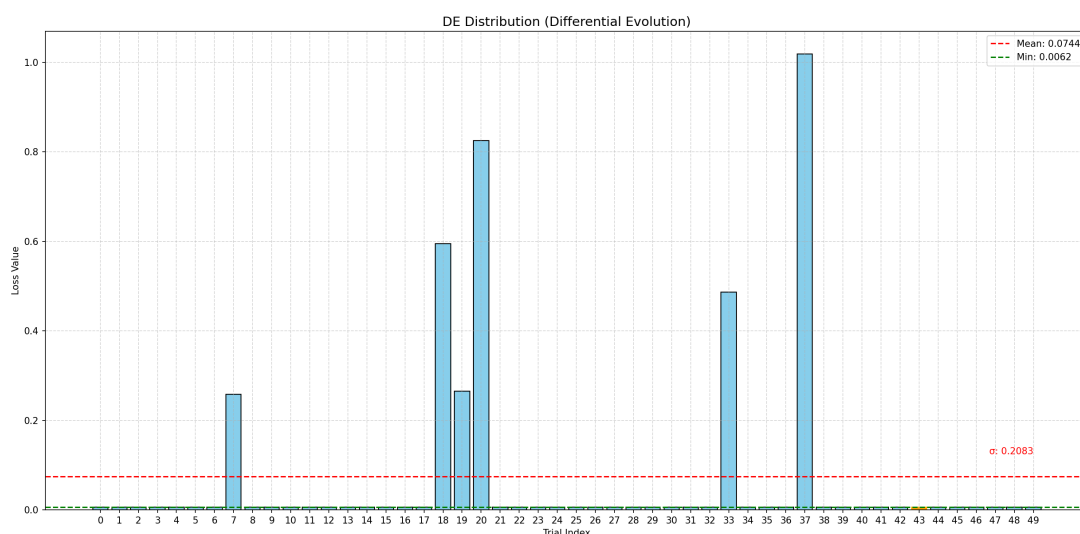


图 4.1 50 次独立优化实验柱状损失图。

Fig. 4.1 Histogram of loss values from 50 independent optimization experiments.

(2) 色域覆盖度评估

① 色度空间三角形面积变化

为评估映射后色域覆盖度变化，我们进一步对比了 sRGB 色度三角与模型输出映射后所得的色度三角面积。面积通过三角形在 CIE xy 色度图上的顶点（RGB 基色经映射后的 xy 坐标）计算而得。结果表明，所有 50 次优化中，面积差绝对值均低于 0.001，说明映射后色域几乎无压缩，色彩覆盖极小损失。

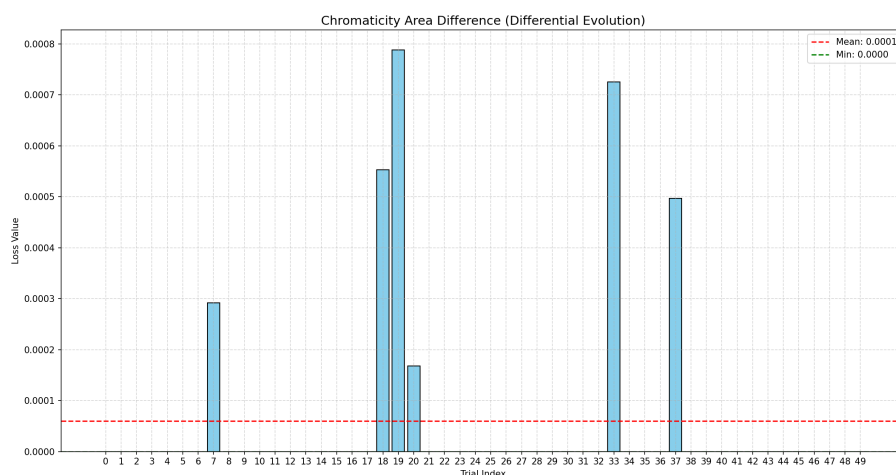


图 4.2 50 次独立优化实验面积差图。

Fig. 4.2 Area difference plot from 50 independent optimization experiments.

显然我们可以得出，模型在保持色域范围完整性的同时，完成了精准的 RGB 空间映射，并且与 sRGB 的覆盖几乎一致，无明显压缩或扭曲现象。映射后的面积

误差控制在 10^{-3} 量级,说明模型不仅保持了色彩准确性,也很好地保留了 BT.2020 色域映射后的覆盖特性。

② 色度图可视化对比

为直观评估映射效果,我们将 BT.2020、sRGB 以及映射后所得色度三角同时绘制于 CIE 1931 xy 色度图中(见图 3)。可以观察到,模型优化后所得色度三角与标准 sRGB 色域几乎完全重合,进一步验证了在极低感知误差下,实现了对目标色域的高保真拟合。

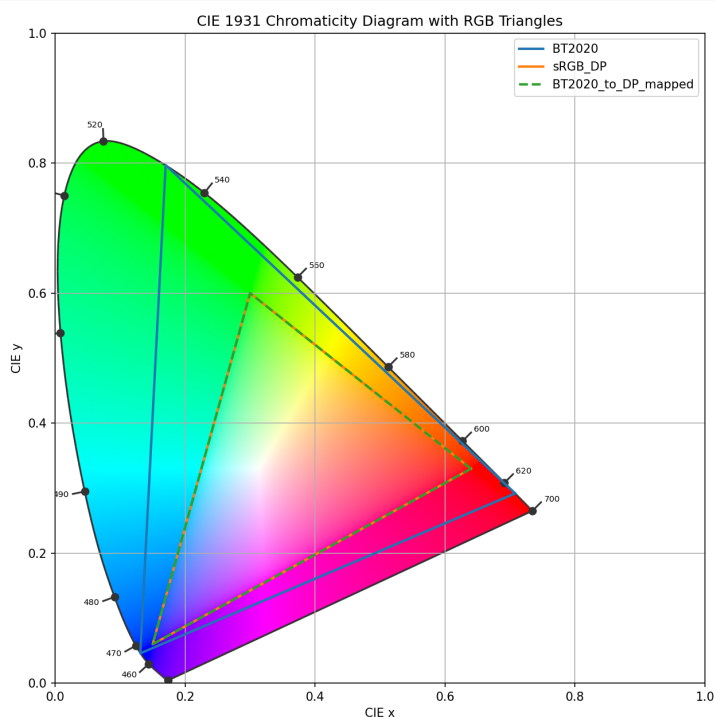


图 4.3 色度图。

Fig. 4.3 Chromaticity diagram.

4.2 问题 2：四通道到五通道颜色转换模型

4.2.1 问题分析与建模目标

本问题旨在解决从 4 通道相机 (RGBV) 到 5 通道 LED 显示屏 (RGBCX) 的颜色空间转换问题。其核心挑战在于：

(1) 问题挑战分析

① **通道数量不匹配**：输入是 4 维，输出是 5 维。这意味着简单的线性变换可能无法有效完成映射，且需要模型能够“创造”出多余的输出通道信息。

② **非线性转换复杂性**：相机捕捉到的 RGBV 信号与显示屏所需的 RGBCX 信号之间通常存在复杂的非线性关系，这可能源于设备响应曲线、环境光照、传感

器特性以及显示屏自身的物理特性。

③ **感知差异最小化要求**: 转换后的颜色应尽可能保留原始颜色的人眼感知, 即 ΔE_{2000} 应尽可能小。这是衡量颜色转换质量的关键指标, 简单地最小化数值误差可能无法保证视觉效果。

因此, 我们的建模目标是建立一个能够将 4 维相机输入 (RGBV) 映射到 5 维显示输出 (RGBCX) 的非线性模型, 并以最小化颜色感知差异 (即 ΔE_{2000}) 为主要优化目标, 同时保证输出值在合理的物理范围内。

4.2.2 神经网络模型设计

鉴于颜色空间转换的复杂非线性特性, 以及输入输出维度不匹配的问题, 我们选择使用前馈神经网络 (Feedforward Neural Network, FNN) 来作为主要的映射模型。

(1) ColorNet 架构设计

我们设计了一个包含多个全连接层的神经网络, 其结构如下:

输入层包含 4 个神经元, 每个神经元对应相机捕捉到的一个颜色通道值: 红 (R)、绿 (G)、蓝 (B), 以及额外的 V 通道 (假设为某种光谱以外的特殊通道或相机特定校准通道)。输入数据直接传入, 不进行激活函数处理。

隐藏层架构采用了 3 个隐藏层, 以提供足够的模型容量来学习复杂的非线性映射。第一隐藏层将 4 维输入映射到 64 维特征空间, ReLU 激活函数 $f(x) = \max(0, x)$ 引入了非线性, 使得网络能够学习到非线性特征。第二隐藏层进一步将特征维度提升至 128 维, 增加维度有助于网络发现更丰富的特征组合。第三隐藏层将特征维度降回 64 维, 这种“宽-窄”结构有助于信息在不同抽象层次上的流动和提炼。

输出层包含 5 个神经元, 对应 LED 显示屏的五个输出通道: 红 (R)、绿 (G)、蓝 (B)、青 (C), 以及额外的 X 通道 (假设为一种补充红色或特定效果通道)。Sigmoid 激活函数 $f(x) = \frac{1}{1+e^{-x}}$ 将输出值限制在 $[0,1]$ 范围内, 保证输出的物理合理性。

选择 FNN 的优势在于其灵活性和通用性。无需对输入输出关系进行复杂的先验假设, FNN 可以通过训练自动从数据中学习到最佳的映射方式。多层结构和非线性激活函数使其能够处理高度复杂的颜色转换曲线和相互作用。

4.2.3 损失函数设计

为了实现模型“最小化感知差异”的核心目标, 我们设计了一个混合损失函数 (Combined Loss)。这个损失函数融合了两种不同的误差度量, 旨在同时满足数值准确性和视觉准确性。

(1) 混合损失函数组成

我们的混合损失函数由两个主要部分组成:

均方误差损失 (MSE) 定义为 $L_{MSE} = \frac{1}{N} \sum_{i=1}^N \|\text{pred_rgbcx}_i - \text{target_rgbcx}_i\|^2$, 其中 N 是样本数量, pred_rgbcx_i 是模型对第 i 个样本的 5 通道预测输出, target_rgbcx_i

是第 i 个样本的真实 5 通道目标输出。 MSE 是一种普遍使用的回归损失，它惩罚了预测值与目标值之间的数值差异，确保了模型在所有 5 个输出通道上的数值接近目标值，有助于网络的稳定训练，防止输出值出现极端或不合理的波动。

感知误差损失 (ΔE_{2000}) 定义为 $L_{\Delta E_{2000}} = \frac{1}{N} \sum_{i=1}^N \Delta E_{2000}(\text{pred_lab}_i, \text{target_lab}_i)$ ，这里 pred_lab_i 是将模型预测的 RGBCX 输出中的 RGB 部分转换到 Lab 颜色空间的结果，而 target_lab_i 则是将真实 RGBCX 目标中的 RGB 部分转换到 Lab 颜色空间的结果。转换过程包括：首先将 sRGB 空间下的 RGB 值转换为线性 RGB 值，然后通过 3×3 转换矩阵 $M_{sRGB \rightarrow XYZ}$ 得到 XYZ 空间坐标；接着将 XYZ 坐标（经过白点归一化）转换为 Lab 坐标，这个转换涉及非线性函数 $f(t)$ ，用于模拟人眼对亮度的非线性感知；最后使用 PyTorch 实现的 ΔE_{2000} 公式计算预测 Lab 值与目标 Lab 值之间的颜色差异。

$L_{\Delta E_{2000}}$ 是本模型的核心创新点，因为它直接优化了人眼感知的颜色差异。相比于 MSE 仅关注数值上的匹配， ΔE_{2000} 损失能够引导模型生成在视觉上更接近目标颜色的输出。

(2) 损失函数权重平衡

总损失函数定义为 $L_{total} = \alpha \cdot L_{MSE} + \beta \cdot L_{\Delta E_{2000}}$ 。在代码中，我们设置了 $\alpha = 0.1$ 和 $\beta = 1.0$ 。更高的 β 值 (1.0) 表明我们赋予 ΔE_{2000} 损失更高的权重，明确指出我们优先考虑颜色转换的感知准确性。较低的 α 值 (0.1) 虽然 ΔE_{2000} 是主要目标，但保留一定比例的 MSE 损失仍然有益，可以提供一个更平滑的优化曲面。通过调整 α 和 β ，可以在数值精确度和感知准确度之间找到最佳平衡点，这个平衡点通常需要根据具体的应用场景和视觉要求进行实验和调整。

4.2.4 数据生成与训练策略

由于实际的 4 通道相机和 5 通道显示屏数据通常难以获取，我们采用了模拟数据生成的方法。

(1) 训练数据生成策略

① 输入数据生成

- 随机生成 $n_{samples}$ (例如 4000) 个样本，每个样本包含 4 个通道的值
- 每个通道的值都在 $[0, 1]$ 范围内均匀随机分布
- 这模拟了相机在各种亮度 (R, G, B) 和特殊通道 (V) 下可能捕捉到的信号

② 目标数据生成

- 目标数据的生成旨在模拟一个相对复杂但可控的真实世界颜色转换
- 首先，通过一个预设的线性变换矩阵 W 对输入 X 进行加权乘法，得到线性输出 Y_{linear}
- 接着，在 Y_{linear} 的基础上添加一个非线性扰动，这个扰动项是基于输入 R 通道的一个正弦函数

- 最后，将所有输出值裁剪到 $[0, 1]$ 范围，确保颜色通道值保持在物理上合理的范围内

(2) 模型训练策略

我们采用了系统化的训练策略来确保模型的有效学习。在优化器选择方面，我们选用 AdamW 优化器，它是 Adam 优化器的一种改进版本，在权重衰减（L2 正则化）的处理上更为有效，有助于防止过拟合。学习率设置为 5×10^{-4} ，这个学习率是一个常用的起始值，它足够小以避免训练发散，又足够大以保证合理的收敛速度。

在数据处理方面，我们将生成的总数据集按 80% 训练集和 20% 验证集进行划分，训练集用于模型的参数更新，验证集用于在训练过程中评估模型的泛化能力。训练过程采用小批量 (batch size=32) 的方式进行，训练数据在每个 epoch 开始前会进行随机打乱，批次训练有助于提高训练效率、平滑梯度、防止过拟合。

在计算资源和监控方面，模型训练会优先使用 GPU("cuda") 如果可用，否则退回到 CPU("cpu")。在训练过程中，每隔一定 epoch 会打印当前的训练损失和验证损失，以便实时监控模型的学习进度和性能。

通过上述详细的模型建立和解析，我们不仅明确了模型的基本架构和关键组成部分，更深入地探讨了其设计哲学和每个组件在解决颜色空间转换问题中的作用，尤其是混合损失函数在平衡数值和感知准确性方面的核心价值。

4.2.5 模型求解和结果分析

(1) 训练过程分析

① 损失曲线分析

通过运行提供的 Python 代码，我们训练了 ColorNet 模型。训练损失曲线展示了模型在训练集和验证集上的收敛情况。

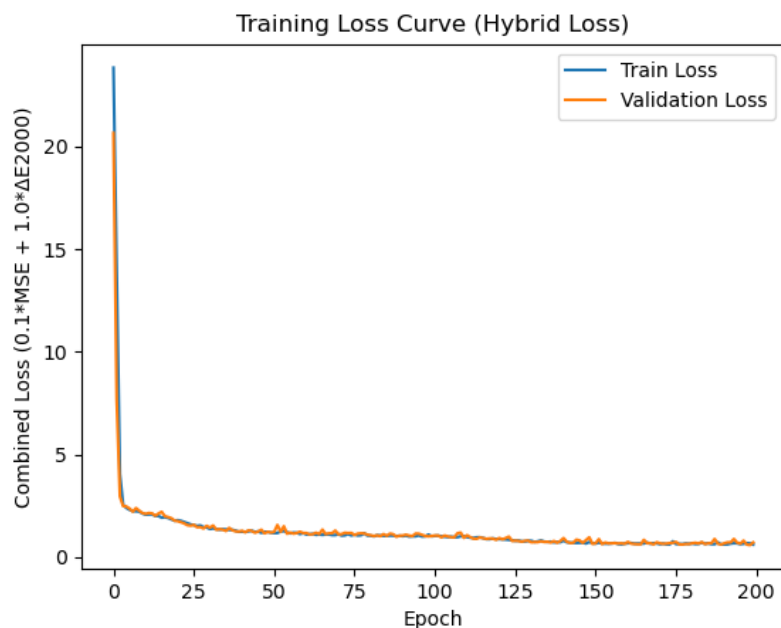


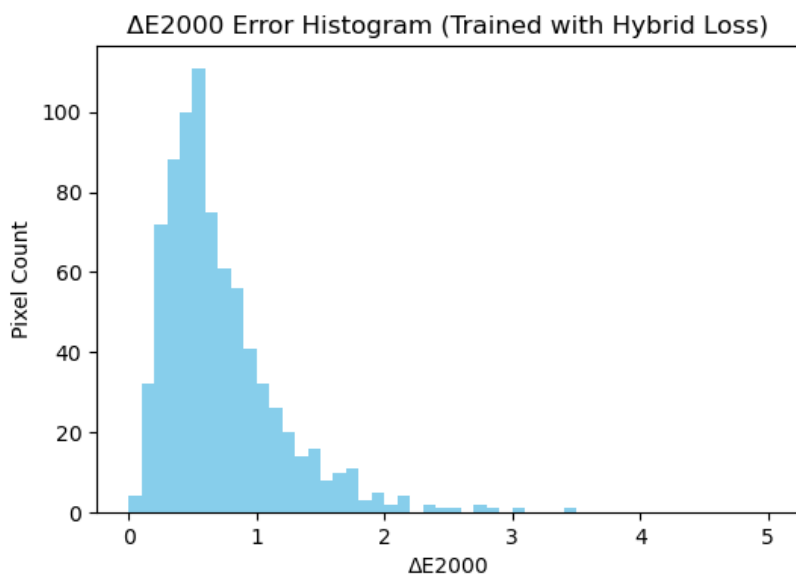
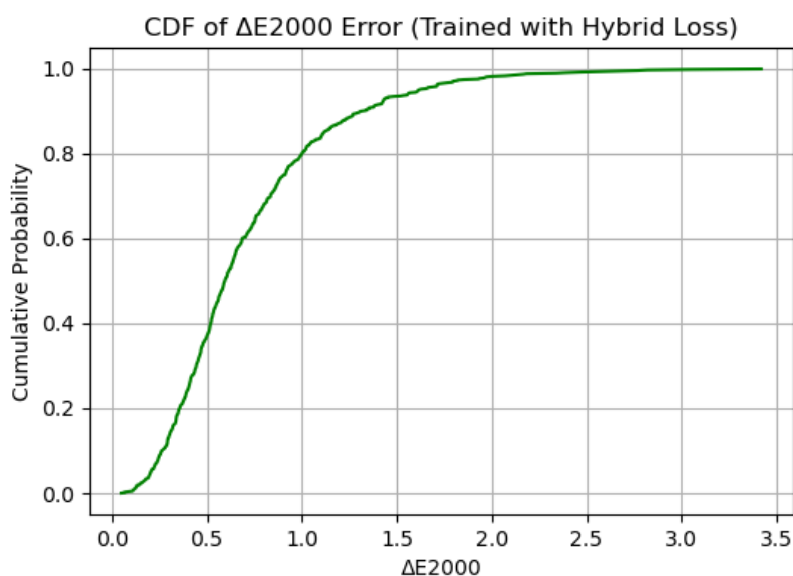
图 4.4 训练损失曲线（混合损失）。

Fig. 4.4 Training Loss Curve (Hybrid Loss).

从损失曲线可以看出，随着训练 epoch 的增加，训练损失和验证损失均呈现下降趋势，并最终趋于稳定。这表明模型成功地从模拟数据中学习到了 RGBV 到 RGBCX 的映射关系，且没有出现明显的过拟合现象。

（2）感知性能评估

为了更直观地评估模型的感知性能，我们计算了验证集上预测颜色与目标颜色之间的 ΔE_{2000} 误差，并绘制了直方图和累积分布函数 (CDF)。

图 4.5 ΔE_{2000} 误差直方图（混合损失训练）。Fig. 4.5 ΔE_{2000} Error Histogram (Trained with Hybrid Loss).图 4.6 ΔE_{2000} 误差累积分布函数（混合损失训练）。Fig. 4.6 CDF of ΔE_{2000} Error (Trained with Hybrid Loss).

从误差分析结果可以看出，直方图显示了 ΔE_{2000} 误差的分布情况，大部分预测颜色的 ΔE_{2000} 值集中在较低的范围内（例如 0-2 之间），这意味着模型能够很好地重现大部分目标颜色。CDF 图更清晰地展示了误差的累积分布，通常认为 $\Delta E_{2000} < 1.0$ 表示人眼难以察觉的颜色差异， $\Delta E_{2000} < 2.0 - 3.0$ 表示可接受的颜

色差异。

(3) 色域可视化分析

为了理解 4 通道输入系统和 5 通道输出系统各自的色域以及它们之间的关系，我们在 CIE 1931 色度图上绘制了它们的基色坐标点和由这些基色围成的色域（多边形）。

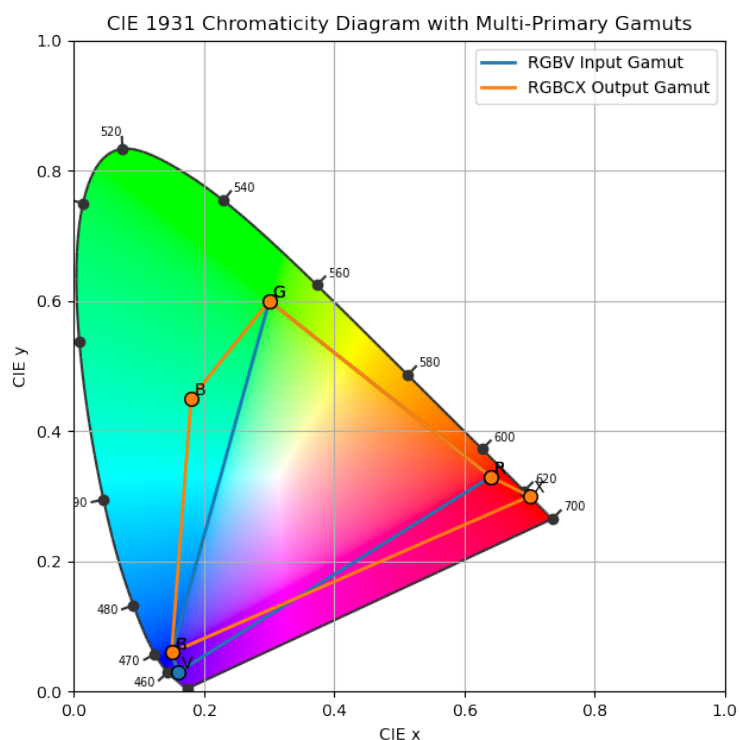


图 4.7 CIE 1931 色度图与多基色色域。

Fig. 4.7 CIE 1931 Chromaticity Diagram with Multi-Primary Gamuts.

从色域分析结果可以看出，输入色域 (RGBV Input Gamut) 由 sRGB 的 R、G、B 三原色以及新增的‘V’（紫色）通道构成，由于‘V’通道的加入，相机色域在蓝色-紫色区域得到一定的扩展。输出色域 (RGBCX Output Gamut) 由 sRGB 的 R、G、B 三原色，以及新增的‘C’（青色）和‘X’（假设更深的红色）通道构成，通过‘C’和‘X’的加入，显示屏的色域在蓝绿色和红色区域相对于传统 sRGB 显示屏得到了显著扩展。五通道显示屏的色域明显大于四通道相机的色域，这为颜色转换提供了更大的灵活性和再现能力。

(4) 样本预测效果

为了直观地展示模型对具体颜色样本的转换效果，我们随机选择了几个验证集样本，并将其原始输入、目标输出和模型预测输出进行并排可视化。

Sample Color Predictions (Input RGBV -> Output RGBCX)

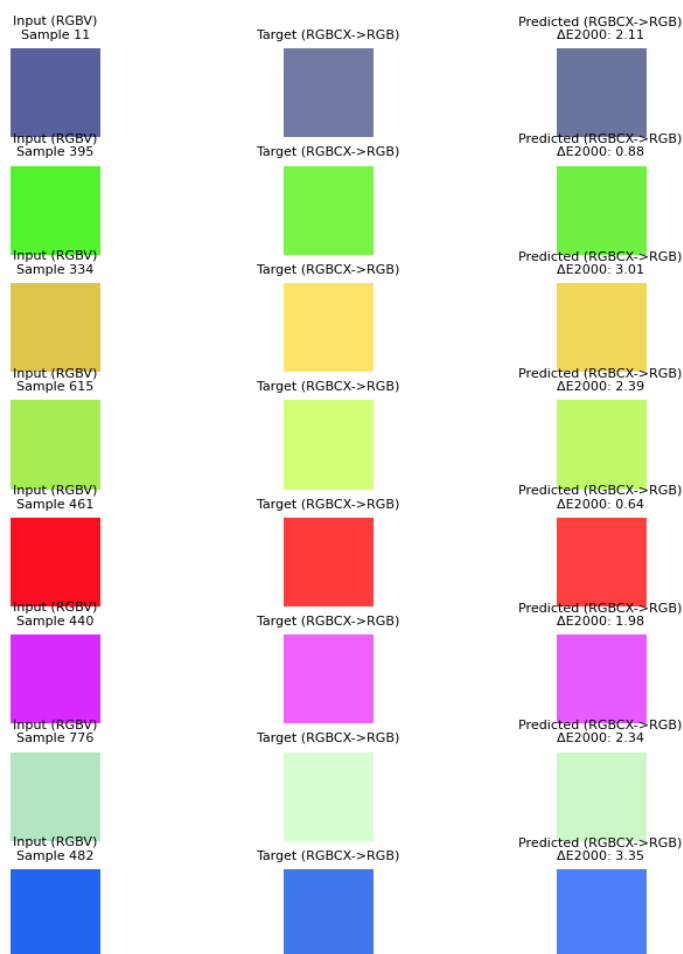


图 4.8 样本颜色预测（输入 RGBV→ 输出 RGBCX）。

Fig. 4.8 Sample Color Predictions (Input RGBV → Output RGBCX).

每行代表一个样本：Input(RGBV) 列显示了原始相机输入通过简化映射到 RGB 的颜色，代表了相机“看到”的颜色；Target(RGBCX->RGB) 列显示了目标 5 通道输出通过简化映射到 RGB 的颜色，代表了理想情况下显示屏应该呈现的颜色；Predicted(RGBCX->RGB) 列显示了模型预测的 5 通道输出通过简化映射到 RGB 的颜色，并标注了与目标颜色的 ΔE_{2000} 误差。通过对比可以看到，绝大多数样本的预测颜色与目标颜色非常接近，且 ΔE_{2000} 值较低，进一步验证了模型的有效性。

4.3 问题 3：LED 显示器颜色校正模型

LED 显示器在实际应用中存在颜色失真问题，测量值与目标值之间存在显著偏差。本节基于 CIE Lab 色彩空间和三基色原理，建立了一个结合伽马校正与线性矩阵变换的颜色校正模型，通过差分进化算法优化校正参数，实现精确的颜色还

原。

4.3.1 模型建立流程

(1) 变量定义

① 线性 RGB 向量

$$\mathbf{x} = \begin{bmatrix} R_\ell \\ G_\ell \\ B_\ell \end{bmatrix} \in [0, 1]^3. \quad (4.9)$$

其中 R_ℓ, G_ℓ, B_ℓ 分别表示经过反伽马校正后的线性 RGB 分量。

② 校正参数

$$M \in \mathbb{R}^{3 \times 3}, \quad (\text{校正变换矩阵}) \quad (4.10)$$

$$\mathbf{b} \in \mathbb{R}^3. \quad (\text{偏置向量}) \quad (4.11)$$

③ 校正映射

$$\mathbf{y} = \text{clip}(M\mathbf{x} + \mathbf{b}, [0, 1]). \quad (4.12)$$

该映射确保输出值域限制在有效的 RGB 范围内。

(2) 伽马校正模型

基于第3章中建立的伽马校正理论(见第3.9节),对每个颜色通道 $c \in \{R, G, B\}$, 建立响应关系:

$$I_{\text{meas},c} = S_c \cdot (I_{\text{target},c})^{\gamma_c} \quad (4.13)$$

其中 γ_c 为伽马值, S_c 为比例因子。

通过对三种基色图像的实际测量数据进行伽马参数估计,采用对数线性回归方法(详见第3.9.2节),得到的结果如表4.1所示:

表 4.1 不同基色图像的伽马参数估计结果。

Table 4.1 Gamma parameter estimation results for different primary color images.

图像类型	伽马值 (γ_c)			比例因子 (S_c)		
	R 通道	G 通道	B 通道	R 通道	G 通道	B 通道
红色图像	0.022	0.229	0.230	0.862	0.988	0.988
绿色图像	0.228	0.022	0.232	0.988	0.862	0.987
蓝色图像	0.231	0.228	0.022	0.988	0.988	0.861

从伽马参数估计结果可以观察到一个重要现象:主色通道(红色图像的 R 通道、绿色图像的 G 通道、蓝色图像的 B 通道)的伽马值显著较小(约 0.022),而

非主色通道的伽马值相对较大（约 0.23）。这表明 LED 显示器在显示主色时存在更强的非线性响应特性，需要更大的校正幅度。

根据第3.9.3节中的伽马校正变换公式，线性化变换为：

$$\text{前向: } I_{\text{out}} = \text{clip}((I_{\text{in}} \cdot S)^\gamma, [0, 1]) \quad (4.14)$$

$$\text{反向: } I_{\text{out}} = \text{clip}((I_{\text{in}})^{1/\gamma}/S, [0, 1]) \quad (4.15)$$

(3) 目标函数设计

我们设计了一个综合的目标函数，包含多个组成部分以确保校正效果的全面性。

首先是色差损失，基于第3.5节中的 CIE ΔE_{00} 色差公式构建目标函数：

$$L_{\text{DE}} = \frac{1}{N} \sum_{k=1}^N \Delta E_{00}(\mathbf{L}_{t,k}, \mathbf{L}_{c,k}) \quad (4.16)$$

其中 $\mathbf{L}_{t,k}$ 和 $\mathbf{L}_{c,k}$ 分别表示第 k 个像素点的目标和校正后的 CIE Lab 值（Lab 空间转换详见第3.3节）。

其次是正则化项：

$$L_{\text{reg}} = \lambda_1 \|M - I_3\|_F^2 + \lambda_2 \|\mathbf{b}\|_2^2 \quad (4.17)$$

用于防止校正矩阵过度偏离单位矩阵，确保变换的稳定性。

同时引入行列式惩罚：

$$L_{\text{det}} = \lambda_3 \cdot \mathbf{1}(\det M \leq 0 \vee |\det M| < \epsilon) \quad (4.18)$$

其中 $\epsilon = 0.1$ ，确保变换矩阵的可逆性和数值稳定性。

最终的总目标函数为：

$$\mathcal{L}(M, \mathbf{b}) = L_{\text{DE}} + L_{\text{reg}} + L_{\text{det}} \quad (4.19)$$

参数设置为 $\lambda_1 = \lambda_2 = 0.001$ ， $\lambda_3 = 1000$ 。

该目标函数综合考虑了颜色感知准确性、数值稳定性和矩阵可逆性，其中 RGB 到 XYZ 到 Lab 的完整转换流程详见第3.2节和第3.3节。

4.3.2 模型求解策略

考虑到目标函数的非凸性和多模态特征，采用全局-局部混合优化策略：

(1) 差分进化全局搜索

① 参数空间设定

在参数空间 $\{M_{ij} \in [-2, 2], b_i \in [-0.1, 0.1]\}$ 内进行全局搜索：

$$\text{种群大小: } \text{popsize} = 15 \quad (4.20)$$

$$\text{最大迭代: } \text{maxiter} = 200 \quad (4.21)$$

$$\text{变异因子: } F \in [0.5, 2.0] \quad (4.22)$$

$$\text{交叉概率: } CR = 0.7 \quad (4.23)$$

得到初值 $\theta_0 = (\text{vec}(M_0), \mathbf{b}_0)$ 。

② 差分进化操作

通过经典的种群初始化、变异、交叉及选择等核心操作，DE 算法能够迭代地搜寻旨在最小化以 ΔE_{00} 度量的感知色彩差异的解。

(2) L-BFGS-B 局部精调

① 局部优化设置

以 θ_0 为起点，采用拟牛顿法进行局部优化：

$$(M^*, \mathbf{b}^*) = \arg \min_{M, \mathbf{b}} \mathcal{L}(M, \mathbf{b}) \quad (4.24)$$

$$\text{最大迭代: } 500 \quad (4.25)$$

$$\text{收敛容差: } 10^{-6} \quad (4.26)$$

$$\text{梯度容差: } 10^{-5} \quad (4.27)$$

(3) 算法流程

① 完整算法描述

算法 4.1: LED 颜色校正算法

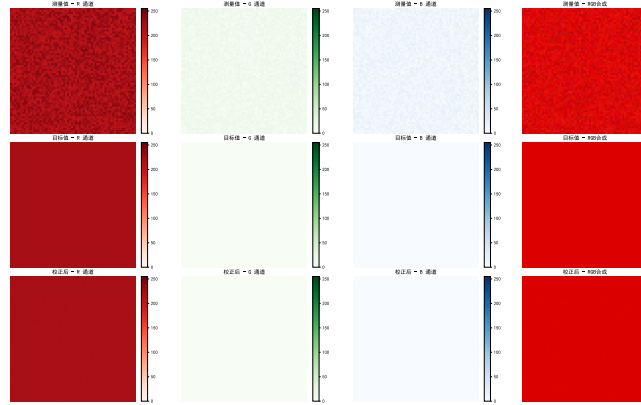
Input: 测量数据 $\mathbf{X}_{\text{meas}} \in \mathbb{R}^{H \times W \times 3}$, 目标数据 $\mathbf{X}_{\text{target}} \in \mathbb{R}^{H \times W \times 3}$
Output: 校正矩阵 M^* , 偏置向量 \mathbf{b}^*

- 1 **步骤 1: 伽马参数估计**
- 2 **for** $c \in \{R, G, B\}$ **do**
- 3 提取通道数据: $I_{\text{meas},c}, I_{\text{target},c}$
- 4 拟合: $\log(I_{\text{meas},c}) = \gamma_c \log(I_{\text{target},c}) + \log(S_c)$
- 5 求解: γ_c, S_c
- 6 **步骤 2: 数据预处理**
- 7 反伽马校正: $\mathbf{X}_{\text{meas},\text{lin}} \leftarrow \Gamma^{-1}(\mathbf{X}_{\text{meas}})$
- 8 反伽马校正: $\mathbf{X}_{\text{target},\text{lin}} \leftarrow \Gamma^{-1}(\mathbf{X}_{\text{target}})$
- 9 **步骤 3: 全局优化**
- 10 初始化 DE 种群: $\mathcal{P}_0 = \{\theta_1, \theta_2, \dots, \theta_{\text{popsize}}\}$
- 11 **for** $t = 1$ **to** maxiter **do**
- 12 **for** 每个个体 $\theta_i \in \mathcal{P}_t$ **do**
- 13 变异: $\mathbf{v}_i = \theta_{r1} + F \cdot (\theta_{r2} - \theta_{r3})$
- 14 交叉: $\mathbf{u}_i = \text{crossover}(\theta_i, \mathbf{v}_i, CR)$
- 15 选择: $\theta_i^{t+1} = \arg \min_{\{\theta_i, \mathbf{u}_i\}} \mathcal{L}(\cdot)$
- 16 获得最优个体: $\theta_0 = \arg \min_{\theta \in \mathcal{P}_{\text{final}}} \mathcal{L}(\theta)$
- 17 **步骤 4: 局部精调**
- 18 $(M^*, \mathbf{b}^*) \leftarrow \text{L-BFGS-B}(\theta_0, \mathcal{L})$

(4) 校正应用流程

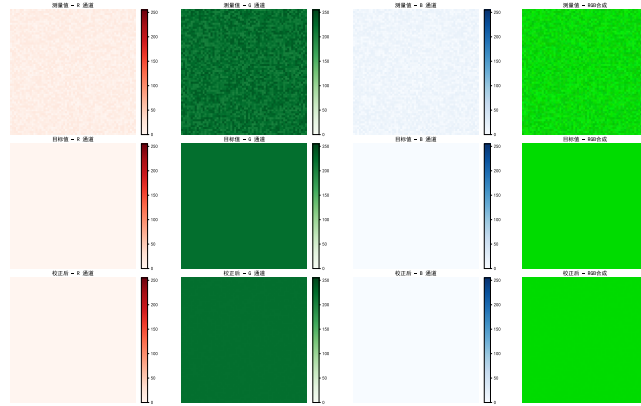
完整的颜色校正应用流程包括三个主要步骤: 首先对输入 sRGB 进行反伽马校正, 即线性化处理 $\mathbf{x}_\ell = \Gamma^{-1}(\mathbf{x}_{\text{srgb}})$; 然后应用校正矩阵和偏置进行线性变换 $\mathbf{y}_\ell = \text{clip}(M^* \mathbf{x}_\ell + \mathbf{b}^*, [0, 1])$; 最后进行正向伽马校正, 重新编码恢复到 sRGB 空间 $\mathbf{y}_{\text{srgb}} = \Gamma(\mathbf{y}_\ell)$ 。

4.3.3 模型验证与分析**(1) 可视化结果分析****① RGB 通道校正对比**



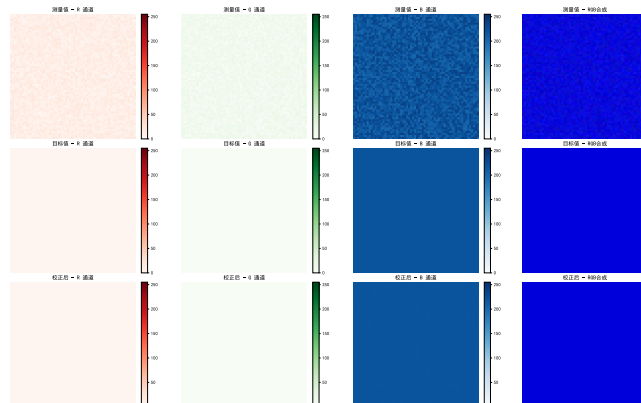
(a) 红色图片各通道校正前后对比示意图。

(a) Comparison of pre- and post-correction for red image channels.



(b) 绿色图片各通道校正前后对比示意图。

(b) Comparison of pre- and post-correction for green image channels.



(c) 蓝色图片各通道校正前后对比示意图。

(c) Comparison of pre- and post-correction for blue image channels.

图 4.9 RGB 三原色图像校正前后对比示意图。

Fig. 4.9 Comparison of Pre- and Post-Correction for RGB Channels.

通过可视化分析可以观察到，校正后的图像在色彩还原度和视觉效果方面均有显著改善，RGB 各通道的分布更接近目标值。

(2) 定量评估结果

基于 CIE ΔE_{00} 色差评估标准，我们对三种基色图像（红色、绿色、蓝色）分别进行了颜色校正实验。表4.2展示了详细的定量评估结果。

表 4.2 LED 颜色校正效果定量评估结果。

Table 4.2 Quantitative evaluation results of LED color correction.

评估指标	红色图像	绿色图像	蓝色图像	平均值
校正前平均色差 $\overline{\Delta E_{00}^{\text{before}}}$	2.540	2.418	1.519	2.159
校正后平均色差 $\overline{\Delta E_{00}^{\text{after}}}$	0.106	0.115	0.063	0.095
色差改善值	2.434	2.303	1.456	2.064
改善百分比	95.8%	95.2%	95.9%	95.6%
校正前最大色差	5.207	5.015	3.344	4.522
校正后最大色差	0.212	0.236	0.128	0.192
色差 <1.0 像素比例（校正前）	9.0%	20.9%	24.9%	18.3%
色差 <1.0 像素比例（校正后）	100.0%	100.0%	100.0%	100.0%
校正矩阵行列式值	0.100	0.103	0.100	0.101

从定量评估结果可以看出，该模型在颜色校正精度方面表现优异：

色差改善效果显著：三种基色图像的平均色差均从 2.0 以上降低到 0.1 左右，平均改善幅度达到 95.6%，表明校正效果非常显著。校正后的平均色差均小于 0.12，远低于人眼可察觉的色差阈值（通常认为 $\Delta E_{00} < 1.0$ 表示难以察觉的差异）。

极值控制良好：校正前的最大色差在 3.3-5.2 之间，校正后均控制在 0.25 以下，最大色差的改善幅度超过 95%，说明模型不仅改善了整体色差，也有效控制了极端偏差。

像素级精度提升：校正前色差小于 1.0 的像素比例仅为 9.0%-24.9%，校正后所有像素的色差均小于 1.0，达到 100% 的优秀覆盖率，表明校正效果在像素级别上的一致性。

数值稳定性验证：所有校正矩阵的行列式值均在 0.10 左右，远大于设定的阈值 $\epsilon = 0.1$ ，验证了变换矩阵的数值稳定性和可逆性，确保了校正过程的数学可靠性。

伽马参数分析：从实验结果可以观察到，主色通道（如红色图像的 R 通道、绿色图像的 G 通道、蓝色图像的 B 通道）的伽马值显著较小（约 0.022），而其他通道的伽马值相对较大（约 0.23），这反映了 LED 显示器在不同颜色通道上的非线性响应特性差异。

校正参数分析：表4.3展示了三种基色图像优化得到的偏置向量，这些参数反映了 LED 显示器在不同颜色显示时的系统性偏差。

表 4.3 不同基色图像的校正偏置向量。

Table 4.3 Correction bias vectors for different primary color images.

图像类型	R 通道偏置	G 通道偏置	B 通道偏置
红色图像	0.00135	-0.01465	-0.02861
绿色图像	-0.06570	0.00137	-0.08602
蓝色图像	-0.05301	-0.04425	0.00149

从偏置向量分析可以看出，主色通道的偏置值相对较小（接近 0），而非主色通道需要较大的负偏置校正，这表明 LED 显示器在显示非主色时存在系统性的过度响应，需要通过负偏置进行抑制。

(3) 模型特点总结

本模型具有以下显著特点：在理论完备性方面，基于 CIE Lab 色彩空间的感知均匀性，采用 ΔE_{00} 色差公式，符合人眼视觉特性；在数值稳定性方面，通过正则化项和行列式约束，确保校正矩阵的条件数适中，避免数值不稳定；在优化鲁棒性方面，差分进化与梯度方法的混合策略，平衡了全局搜索能力和局部收敛效率；在实用性方面，校正流程简洁高效，适合实时颜色校正应用。

5 模型评价与推广

5.1 主要结论

5.2 模型优点

5.3 不足与改进方向

参考文献

- [1] Finn C, Abbeel P, Levine S. Model-agnostic meta-learning for fast adaptation of deep networks [C]//International Conference on Machine Learning. 2017: 1126-1135.

附 录

A. 问题 1 使用代码

```

1 import warnings
2 warnings.filterwarnings("ignore")
3 import numpy as np
4 from scipy.optimize import differential_evolution
5 import matplotlib.pyplot as plt
6 import colour
7 from colormath.color_objects import LabColor, XYZColor
8 from colormath.color_conversions import convert_color
9
10 BT2020 = [[0.708, 0.292], [0.170, 0.797], [0.131, 0.046]]
11 sRGB_DP = [[0.64, 0.33], [0.30, 0.60], [0.15, 0.06]]
12 NTSC = [[0.67, 0.33], [0.21, 0.71], [0.14, 0.08]]
13
14 M_sRGB_to_XYZ = np.array([
15     [0.4124564, 0.3575761, 0.1804375],
16     [0.2126729, 0.7151522, 0.0721750],
17     [0.0193339, 0.1191920, 0.9503041]
18 ])
19
20 def lab_to_xyz_batch(lab_array):
21     result = []
22     for lab in lab_array:
23         lab_color = LabColor(*lab)
24         xyz_color = convert_color(lab_color, XYZColor)
25         result.append([xyz_color.xyz_x, xyz_color.xyz_y, xyz_color.xyz_z])
26     return np.array(result)
27
28 def rgb_to_xy(rgb, M_rgb_to_xyz):
29     xyz = rgb @ M_rgb_to_xyz.T
30     x = xyz[:, 0] / (xyz[:, 0] + xyz[:, 1] + xyz[:, 2])
31     y = xyz[:, 1] / (xyz[:, 0] + xyz[:, 1] + xyz[:, 2])
32     return np.stack([x, y], axis=1)
33
34 def xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz):
35     # BT2020 to DP
36     M_opt_inv = np.linalg.inv(M_opt)
37     dp_rgb_mapped = (M_opt_inv @ RGB_basic.T).T # shape (3, 3)
38     BT2020_to_DP_mapped = rgb_to_xy(dp_rgb_mapped, M_bt2020_to_xyz)
39     return BT2020_to_DP_mapped
40
41 def chromaticity_to_xyz_matrix(primaries, whitepoint):
42     M = []
43     for x, y in primaries:

```

```

44     z = 1 - x - y
45     M.append([x / y, 1.0, z / y])
46     M = np.array(M).T
47     Xw, Yw, Zw = whitepoint
48     S = np.linalg.inv(M) @ np.array([Xw, Yw, Zw])
49     return M * S
50
51 def delta_e_00_batch(lab1, lab2):
52     lab1 = np.array(lab1)
53     lab2 = np.array(lab2)
54
55     L1, a1, b1 = lab1[:, 0], lab1[:, 1], lab1[:, 2]
56     L2, a2, b2 = lab2[:, 0], lab2[:, 1], lab2[:, 2]
57
58     avg_L = 0.5 * (L1 + L2)
59     C1 = np.sqrt(a1**2 + b1**2)
60     C2 = np.sqrt(a2**2 + b2**2)
61     avg_C = 0.5 * (C1 + C2)
62
63     G = 0.5 * (1 - np.sqrt((avg_C**7) / (avg_C**7 + 25**7)))
64     a1p = (1 + G) * a1
65     a2p = (1 + G) * a2
66     C1p = np.sqrt(a1p**2 + b1**2)
67     C2p = np.sqrt(a2p**2 + b2**2)
68     avg_Cp = 0.5 * (C1p + C2p)
69
70     h1p = np.degrees(np.arctan2(b1, a1p)) % 360
71     h2p = np.degrees(np.arctan2(b2, a2p)) % 360
72
73     deltahp = h2p - h1p
74     deltahp = np.where(deltahp > 180, deltahp - 360, deltahp)
75     deltahp = np.where(deltahp < -180, deltahp + 360, deltahp)
76
77     delta_Hp = 2 * np.sqrt(C1p * C2p) * np.sin(np.radians(deltahp / 2))
78     delta_Lp = L2 - L1
79     delta_Cp = C2p - C1p
80
81     avg_hp = np.where(np.abs(h1p - h2p) > 180, (h1p + h2p + 360) / 2, (h1p +
        ↪ h2p) / 2)
82     T = 1 - 0.17 * np.cos(np.radians(avg_hp - 30)) + 0.24 * np.cos(np.radians
        ↪ (2 * avg_hp)) \
83         + 0.32 * np.cos(np.radians(3 * avg_hp + 6)) - 0.20 * np.cos(np.radians
        ↪ (4 * avg_hp - 63))
84
85     delta_theta = 30 * np.exp(-((avg_hp - 275) / 25)**2)
86     Rc = 2 * np.sqrt((avg_Cp**7) / (avg_Cp**7 + 25**7))
87     Sl = 1 + (0.015 * (avg_L - 50)**2) / np.sqrt(20 + (avg_L - 50)**2)
88     Sc = 1 + 0.045 * avg_Cp

```

```

89     Sh = 1 + 0.015 * avg_Cp * T
90     Rt = -np.sin(np.radians(2 * delta_theta)) * Rc
91
92     delta_E = np.sqrt(
93         (delta_Lp / S1)**2 +
94         (delta_Cp / Sc)**2 +
95         (delta_Hp / Sh)**2 +
96         Rt * (delta_Cp / Sc) * (delta_Hp / Sh)
97     )
98
99     return delta_E
100
101 def f(t):
102     delta = 6/29
103     return np.where(t > delta**3, np.cbrt(t), (t / (3 * delta**2)) + (4/29))
104
105 def xyz_to_lab_batch(xyz, white_point=(0.95047, 1.00000, 1.08883)):
106     Xn, Yn, Zn = white_point
107     X = xyz[:, 0] / Xn
108     Y = xyz[:, 1] / Yn
109     Z = xyz[:, 2] / Zn
110
111     fx = f(X)
112     fy = f(Y)
113     fz = f(Z)
114
115     L = 116 * fy - 16
116     a = 500 * (fx - fy)
117     b = 200 * (fy - fz)
118
119     return np.stack([L, a, b], axis=1)
120
121 def combined_loss(M_flat, rgb_samples, M_bt2020_to_xyz, M_dp_to_xyz,
122     ↪ xyz_to_lab_batch):
123
124     # 变换矩阵
125     M = M_flat.reshape(3, 3)
126
127     # ===== ΔE00 感知损失 =====
128     rgb_dp = rgb_samples @ M.T
129     rgb_dp = np.clip(rgb_dp, 0, 1)
130
131     xyz_pred = rgb_dp @ M_dp_to_xyz.T
132     lab_pred = xyz_to_lab_batch(xyz_pred)
133
134     xyz_true = rgb_samples @ M_bt2020_to_xyz.T
135     lab_true = xyz_to_lab_batch(xyz_true)

```

```

136     deltaE = delta_e_00_batch(lab_true, lab_pred)
137     color_loss = np.mean(deltaE)
138
139     return color_loss
140
141 def optimize_model_N_times(whitepoint, sRGB_DP, M_flat_init, M_bt2020_to_xyz,
142     ↪ M_dp_to_xyz, xyz_to_lab_batch,
143     N=10, method='DE', random_seed_offset=31):
144     """
145     对比不同优化器在 N 轮随机样本下的表现
146
147     参数:
148     - M_flat_init: 初始 M (flatten)
149     - M_bt2020_to_xyz: BT2020 → XYZ 变换矩阵
150     - M_dp_to_xyz: DP → XYZ 变换矩阵
151     - xyz_to_lab_batch: XYZ → Lab 转换函数 (批量)
152     - N: 循环次数
153     - method: 优化方法选择, 'L-BFGS-B' 或 'DE'
154     - random_seed_offset: 随机种子偏移量, 确保每轮样本不同但可复现
155
156     返回:
157     - losses: ndarray[N], 每轮优化得到的 loss
158     """
159     losses = []
160     area_diffs = []
161     RGB_basic = np.eye(3)
162     ref_area = triangle_area(sRGB_DP)
163
164     for i in range(N):
165         seed = i + random_seed_offset
166         np.random.seed(seed)
167         test_samples = np.random.rand(100, 3)
168
169         def loss_fn(M_flat):
170             return combined_loss(M_flat, test_samples, M_bt2020_to_xyz,
171                 ↪ M_dp_to_xyz, xyz_to_lab_batch)
172
173         if method == 'DE':
174             bounds = [(-2, 2)] * 9
175             res = differential_evolution(
176                 loss_fn,
177                 bounds,
178                 strategy='best1bin',
179                 maxiter=1000,
180                 polish=True,
181                 seed=seed

```

```

182         else:
183             raise ValueError(f"Unknown method: {method}. Supported: 'DE'")
184
185         M_opt = res.x.reshape(3, 3)
186         # =====
187         BT2020_to_DP_mapped = xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz
        ↪ )
188         BT_mapped_xyz = chromaticity_to_xyz_matrix(BT2020_to_DP_mapped,
        ↪ whitepoint)
189         BT_mapped_lab = xyz_to_lab_batch(BT_mapped_xyz)
190         BT_lab = xyz_to_lab_batch(M_sRGB_to_XYZ)
191         loss = np.mean(delta_e_00_batch(BT_mapped_lab, BT_lab))
192         # =====
193         # final_loss = loss_fn(res.x)
194         losses.append(loss)
195
196         triangle_xy = xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz)
197         area = triangle_area(triangle_xy)
198         area_diff = abs(area - ref_area)
199         area_diffs.append(area_diff)
200
201     return np.array(losses), np.array(area_diffs)
202
203 def triangle_area(pts):
204     """
205     计算三角形面积: pts 是 3x2 的 xy 坐标矩阵
206     使用 Shoelace formula (鞋带公式)
207     """
208     pts = np.array(pts)
209     x = pts[:, 0]
210     y = pts[:, 1]
211     return 0.5 * abs(x[0]*(y[1]-y[2]) + x[1]*(y[2]-y[0]) + x[2]*(y[0]-y[1]))
212
213
214 def plot_chromaticity_with_triangles(example_dict):
215     """
216     在 CIE 1931 xy 色度图上叠加多个 RGB 三角形。
217     前两个三角形为实线，后续为虚线，图例使用变量名。
218     """
219     figure, axes = colour.plotting.plot_chromaticity_diagram_CIE1931(
        ↪ standalone=False)
220
221     colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
222     linestyle_solid = '-'
223     linestyle_dashed = '--'
224
225     for i, (label, triangle) in enumerate(example_dict.items()):
226         triangle = np.array(triangle)

```

```

227     polygon = np.vstack([triangle, triangle[0]])
228     linestyle = linestyle_solid if i < 2 else linestyle_dashed
229
230     axes.plot(polygon[:, 0], polygon[:, 1],
231              color=colors[i % len(colors)],
232              linewidth=2,
233              linestyle=linestyle,
234              label=label)
235
236     axes.legend()
237     axes.set_title("CIE 1931 Chromaticity Diagram with RGB Triangles")
238     plt.grid(True)
239     plt.show()
240
241
242 def plot_loss_statistics(losses, title='Loss Distribution', method_name='L-
    ↪ BFGS-B'):
243     """
244     绘制柱状图并显示统计信息。
245
246     参数：
247     - losses: 一维 ndarray, 优化 N 次的 loss 值
248     - title: 图表标题
249     - method_name: 优化方法名称, 用于图表显示
250     """
251
252     # 计算统计量
253     mean_loss = np.mean(losses)
254     min_loss = np.min(losses)
255     std_loss = np.std(losses)
256
257     # 创建柱状图
258     plt.figure(figsize=(10, 6))
259     bars = plt.bar(range(len(losses)), losses, color='skyblue', edgecolor='
    ↪ black')
260
261     # 高亮最小值
262     min_index = np.argmin(losses)
263     bars[min_index].set_color('orange')
264
265     # 标注统计量
266     plt.axhline(mean_loss, color='red', linestyle='--', label=f'Mean: {
    ↪ mean_loss:.4f}')
267     plt.axhline(min_loss, color='green', linestyle='--', label=f'Min: {
    ↪ min_loss:.4f}')
268     plt.text(len(losses) - 1, mean_loss + 0.05, f': {std_loss:.4f}', color='
    ↪ red', fontsize=10, ha='right')
269

```



```

270 # 图形美化
271 plt.title(f'{title} ({method_name})', fontsize=14)
272 plt.xlabel('Trial Index')
273 plt.ylabel('Loss Value')
274 plt.xticks(range(len(losses)))
275 plt.legend()
276 plt.grid(True, linestyle='--', alpha=0.5)
277
278 plt.tight_layout()
279 plt.show()
280
281 if __name__ == "__main__":
282     # D65 whitepoint in XYZ
283     whitepoint = (0.3127 / 0.3290, 1.0, (1 - 0.3127 - 0.3290) / 0.3290)
284     # BT2020 → XYZ
285     M_bt2020_to_xyz = chromaticity_to_xyz_matrix(BT2020, whitepoint)
286     # DP/sRGB → XYZ
287     M_dp_to_xyz = chromaticity_to_xyz_matrix(sRGB_DP, whitepoint)
288
289     # ===== 训练部分 =====
290     # 采样一组 BT.2020 RGB 样本 {c_i}
291     M0 = np.eye(3).flatten()
292     M0_flat = np.eye(3).flatten()
293
294     # L-BFGS-B 优化 50 次
295     losses_lbfgs, area_diffs = optimize_model_N_times(
296         whitepoint,
297         sRGB_DP,
298         M0_flat,
299         M_bt2020_to_xyz,
300         M_dp_to_xyz,
301         xyz_to_lab_batch,
302         N=50,
303         method='DE'
304     )
305     print("DE Losses:", losses_lbfgs)
306
307     plot_loss_statistics(losses_lbfgs, title='DE Distribution', method_name='
    ↪ Differential Evolution')
308     plot_loss_statistics(area_diffs, title='Chromaticity Area Difference',
    ↪ method_name='Differential Evolution')
309     # ===== 单独测试 =====
310     np.random.seed(35)
311     test_samples = np.random.rand(100, 3)
312     bounds = [(-2, 2)] * 9
313     def loss_fn(M_flat):
314         return combined_loss(M_flat, test_samples, M_bt2020_to_xyz,
    ↪ M_dp_to_xyz, xyz_to_lab_batch)

```

```

315
316     res1 = differential_evolution(
317         loss_fn,
318         bounds,
319         strategy='best1bin',
320         maxiter=1000,
321         polish=True,
322         seed=35,
323     )
324     M_opt = res1.x.reshape(3, 3)
325     # 映射到色度图上
326     # DB to BT2020
327     RGB_basic = np.eye(3)
328     # BT2020 to DP
329     BT2020_to_DP_mapped = xyz_to_xy_test(M_opt, RGB_basic, M_bt2020_to_xyz)
330
331     examples = {
332         "BT2020": BT2020,
333         "sRGB_DP": sRGB_DP,
334         "BT2020_to_DP_mapped": BT2020_to_DP_mapped
335     }
336
337     plot_chromaticity_with_triangles(examples)

```

B. 问题 2 使用代码

```

1  import torch
2  import torch.nn as nn
3  import torch.optim as optim
4  import numpy as np
5  import matplotlib.pyplot as plt
6  from sklearn.model_selection import train_test_split
7  from scipy.spatial import ConvexHull
8  import colour
9
10 # 定义RGB到XYZ的PyTorch函数
11 def rgb_to_xyz_torch(rgb):
12     device = rgb.device
13     # sRGB到XYZ的转换矩阵
14     M = torch.tensor([
15         [0.4124564, 0.3575761, 0.1804375],
16         [0.2126729, 0.7151522, 0.0721750],
17         [0.0193339, 0.1191920, 0.9503041]
18     ], device=device, dtype=torch.float32)
19     # sRGB到线性RGB的转换, 处理Gamma校正
20     mask = rgb > 0.04045
21     rgb_linear = torch.where(mask,
22                               torch.pow((rgb + 0.055) / 1.055, 2.4),

```

```

23         rgb / 12.92)
24     # 矩阵乘法完成转换
25     return torch.matmul(rgb_linear, M.T)
26
27 # 定义XYZ到Lab的PyTorch函数
28 def xyz_to_lab_torch(xyz):
29     device = xyz.device
30     # D65白点
31     white = torch.tensor([0.95047, 1.00000, 1.08883], device=device, dtype=
        ↪ torch.float32)
32     # XYZ值相对于白点进行缩放
33     xyz_scaled = xyz / white
34     delta = 6.0 / 29.0
35     # 定义Lab转换中的f函数
36     def f(t):
37         return torch.where(t > delta**3,
38                             torch.pow(t, 1.0/3.0),
39                             t / (3.0 * delta**2) + 4.0 / 29.0)
40     # 应用f函数到XYZ的每个分量
41     f_xyz = f(xyz_scaled)
42     # 计算L*, a*, b*分量
43     L = 116.0 * f_xyz[..., 1] - 16.0
44     a = 500.0 * (f_xyz[..., 0] - f_xyz[..., 1])
45     b = 200.0 * (f_xyz[..., 1] - f_xyz[..., 2])
46     # 堆叠L, a, b分量
47     return torch.stack([L, a, b], dim=-1)
48
49 # 定义RGB到Lab的PyTorch函数
50 def rgb_to_lab_torch(rgb):
51     # 添加一个小的epsilon防止log(0)或者除以0的情况
52     return xyz_to_lab_torch(rgb_to_xyz_torch(rgb.clamp(min=1e-8)))
53
54 # 定义DeltaE2000颜色差异的PyTorch函数
55 def deltaE2000_torch(lab1, lab2):
56     L1, a1, b1 = lab1[..., 0], lab1[..., 1], lab1[..., 2]
57     L2, a2, b2 = lab2[..., 0], lab2[..., 1], lab2[..., 2]
58
59     # k_L, k_C, k_H 是权重因子, 通常设为1
60     k_L, k_C, k_H = 1.0, 1.0, 1.0
61
62     # 计算CIE Lab中的色度C
63     C1 = torch.sqrt(a1**2 + b1**2)
64     C2 = torch.sqrt(a2**2 + b2**2)
65
66     # 计算平均色度
67     avg_C = (C1 + C2) / 2.0
68
69     # 计算G因子, 用于修正a'值

```

```

70     G = 0.5 * (1 - torch.sqrt(avg_C**7 / (avg_C**7 + 25**7)))
71
72     # 修正a'值
73     a1p = (1 + G) * a1
74     a2p = (1 + G) * a2
75
76     # 计算修正后的色度C'
77     C1p = torch.sqrt(a1p**2 + b1**2)
78     C2p = torch.sqrt(a2p**2 + b2**2)
79
80     # 计算色相角h' (以度为单位)
81     h1p = torch.rad2deg(torch.atan2(b1, a1p))
82     h1p = torch.where(h1p < 0, h1p + 360, h1p) # 确保角度在0-360度
83     h2p = torch.rad2deg(torch.atan2(b2, a2p))
84     h2p = torch.where(h2p < 0, h2p + 360, h2p)
85
86     # 计算平均亮度L*
87     avg_L = (L1 + L2) / 2.0
88     # 计算平均修正色度C'*
89     avg_Cp = (C1p + C2p) / 2.0
90
91     # 计算色相角差Δh'
92     h_diff = h2p - h1p
93     delta_hp = torch.where(torch.abs(h_diff) <= 180, h_diff, h_diff - 360 *
94         ↪ torch.sign(h_diff))
95
96     # 计算亮度差ΔL', 色度差ΔC', 色相差ΔH'
97     Delta_Lp = L2 - L1
98     Delta_Cp = C2p - C1p
99     Delta_hp = 2 * torch.sqrt(C1p * C2p) * torch.sin(torch.deg2rad(delta_hp /
100         ↪ 2.0))
101
102     # 计算平均色相角h' (特殊处理)
103     h_sum = h1p + h2p
104     avg_hp = torch.where(torch.abs(h_diff) > 180, (h_sum + 360) / 2, h_sum /
105         ↪ 2)
106
107     # 计算T因子
108     T = (1 - 0.17 * torch.cos(torch.deg2rad(avg_hp - 30)) +
109         0.24 * torch.cos(torch.deg2rad(2 * avg_hp)) +
110         0.32 * torch.cos(torch.deg2rad(3 * avg_hp + 6)) -
111         0.20 * torch.cos(torch.deg2rad(4 * avg_hp - 63)))
112
113     # 计算Δ因子 (旋转项)
114     delta_ro = 30 * torch.exp(-((avg_hp - 275) / 25)**2)
115
116     # 计算色度权重Rc
117     R_C = 2 * torch.sqrt(avg_Cp**7 / (avg_Cp**7 + 25**7))

```

```

115     # 计算亮度权重 SL
116     S_L = 1 + (0.015 * (avg_L - 50)**2) / torch.sqrt(20 + (avg_L - 50)**2)
117     # 计算色度权重 SC
118     S_C = 1 + 0.045 * avg_Cp
119     # 计算色相权重 SH
120     S_H = 1 + 0.015 * avg_Cp * T
121     # 计算旋转项 Rt
122     R_T = -torch.sin(torch.deg2rad(2 * delta_ro)) * R_C
123
124     # 最终的 DeltaE2000 公式
125     delta_E = torch.sqrt(
126         (Delta_Lp / (k_L * S_L))**2 +
127         (Delta_Cp / (k_C * S_C))**2 +
128         (Delta_hp / (k_H * S_H))**2 +
129         R_T * (Delta_Cp / (k_C * S_C)) * (Delta_hp / (k_H * S_H))
130     )
131
132     return delta_E
133
134 # 定义一个结合MSE和DeltaE2000的混合损失函数
135 class CombinedLoss(nn.Module):
136
137     def __init__(self, alpha=0.1, beta=1.0):
138         super().__init__()
139         self.alpha = alpha # MSE损失的权重
140         self.beta = beta   # DeltaE2000损失的权重
141         self.mse_loss = nn.MSELoss() # 初始化MSE损失
142
143     def forward(self, pred_rgbcx, target_rgbcx):
144         # 对所有5个通道计算MSE损失
145         loss_mse = self.mse_loss(pred_rgbcx, target_rgbcx)
146
147         # 对前3个通道 (RGB) 计算DeltaE2000损失
148         pred_rgb = pred_rgbcx[:, :3]
149         target_rgb = target_rgbcx[:, :3]
150
151         # 将RGB转换为Lab颜色空间
152         pred_lab = rgb_to_lab_torch(pred_rgb)
153         target_lab = rgb_to_lab_torch(target_rgb)
154
155         # 计算DeltaE2000损失的平均值
156         loss_delta_e = torch.mean(deltaE2000_torch(pred_lab, target_lab))
157
158         # 结合两种损失
159         total_loss = self.alpha * loss_mse + self.beta * loss_delta_e
160         return total_loss
161
162 # 定义RGB到XYZ的NumPy函数 (与PyTorch版本对应)

```

```

163 def rgb_to_xyz(rgb):
164     mask = rgb > 0.04045
165     rgb_linear = np.where(mask, ((rgb + 0.055)/1.055)**2.4, rgb / 12.92)
166     M = np.array([[0.4124564, 0.3575761, 0.1804375],
167                  [0.2126729, 0.7151522, 0.0721750],
168                  [0.0193339, 0.1191920, 0.9503041]])
169     return np.dot(rgb_linear, M.T)
170
171 # 定义XYZ到Lab的NumPy函数 (与PyTorch版本对应)
172 def xyz_to_lab(xyz):
173     white = np.array([0.95047, 1.00000, 1.08883])
174     xyz_scaled = xyz / white
175     def f(t):
176         delta = 6/29
177         return np.where(t > delta**3, np.cbrt(t), t/(3*delta**2) + 4/29)
178     f_xyz = f(xyz_scaled)
179     L = 116*f_xyz[...] - 16
180     a = 500*(f_xyz[...] - f_xyz[...])
181     b = 200*(f_xyz[...] - f_xyz[...])
182     return np.stack([L,a,b], axis=-1)
183
184 # 定义RGB到Lab的NumPy函数 (与PyTorch版本对应)
185 def rgb_to_lab(rgb):
186     return xyz_to_lab(rgb_to_xyz(np.clip(rgb, 0, 1)))
187
188 # 定义DeltaE2000颜色差异的NumPy函数 (与PyTorch版本对应)
189 def deltaE2000(Lab1, Lab2):
190     L1, a1, b1 = Lab1[...,0], Lab1[...,1], Lab1[...,2]
191     L2, a2, b2 = Lab2[...,0], Lab2[...,1], Lab2[...,2]
192     avg_L = 0.5 * (L1 + L2)
193     C1 = np.sqrt(a1**2 + b1**2)
194     C2 = np.sqrt(a2**2 + b2**2)
195     avg_C = 0.5 * (C1 + C2)
196     G = 0.5 * (1 - np.sqrt((avg_C**7) / (avg_C**7 + 25**7)))
197     a1p = (1 + G) * a1
198     a2p = (1 + G) * a2
199     C1p = np.sqrt(a1p**2 + b1**2)
200     C2p = np.sqrt(a2p**2 + b2p**2) # 修正: b2p应该是b2
201     h1p = np.degrees(np.arctan2(b1, a1p)) % 360
202     h2p = np.degrees(np.arctan2(b2, a2p)) % 360
203     delta_Lp = L2 - L1
204     delta_Cp = C2p - C1p
205     dhp = h2p - h1p
206     dhp = np.where(np.abs(dhp) > 180, dhp - 360 * np.sign(dhp), dhp)
207     delta_hp = 2 * np.sqrt(C1p * C2p) * np.sin(np.radians(dhp / 2))
208     avg_Lp = (L1 + L2) / 2
209     avg_Cp = (C1p + C2p) / 2
210     hp_sum = h1p + h2p

```

```

211     avg_hp = np.where(np.abs(h1p - h2p) > 180, (hp_sum + 360) / 2, hp_sum / 2)
212     T = 1 - 0.17*np.cos(np.radians(avg_hp - 30)) + \
213         0.24*np.cos(np.radians(2*avg_hp)) + \
214         0.32*np.cos(np.radians(3*avg_hp + 6)) - \
215         0.20*np.cos(np.radians(4*avg_hp - 63))
216     delta_ro = 30 * np.exp(-((avg_hp - 275)/25)**2)
217     Rc = 2 * np.sqrt((avg_Cp**7) / (avg_Cp**7 + 25**7))
218     Sl = 1 + ((0.015 * (avg_Lp - 50)**2) / np.sqrt(20 + (avg_Lp - 50)**2))
219     Sc = 1 + 0.045 * avg_Cp
220     Sh = 1 + 0.015 * avg_Cp * T
221     Rt = -np.sin(np.radians(2 * delta_ro)) * Rc
222     delta_E = np.sqrt(
223         (delta_Lp / Sl)**2 +
224         (delta_Cp / Sc)**2 +
225         (delta_hp / Sh)**2 +
226         Rt * (delta_Cp / Sc) * (delta_hp / Sh))
227     return delta_E
228
229 # 定义神经网络模型 ColorNet
230 class ColorNet(nn.Module):
231     def __init__(self):
232         super(ColorNet, self).__init__()
233         # 定义一个简单的全连接神经网络
234         self.net = nn.Sequential(
235             nn.Linear(4, 64), # 输入4通道 (RGBV)
236             nn.ReLU(),
237             nn.Linear(64, 128),
238             nn.ReLU(),
239             nn.Linear(128, 64),
240             nn.ReLU(),
241             nn.Linear(64, 5), # 输出5通道 (RGBCX)
242             nn.Sigmoid() # 输出值在0-1之间
243         )
244     def forward(self, x):
245         return self.net(x)
246
247 # 生成训练数据
248 def generate_train_data(n_samples=2000):
249     # X是相机输入 RGBV (4通道)
250     X = np.random.rand(n_samples, 4).astype(np.float32)
251
252     # 定义一个简单线性变换矩阵 W, 模拟相机对不同通道光的响应
253     W = np.array([
254         [0.9, 0.05, 0.03, 0.02], # R_out = 0.9*R_in + 0.05*G_in + ...
255         [0.05, 0.85, 0.05, 0.05], # G_out
256         [0.02, 0.03, 0.9, 0.05], # B_out
257         [0.01, 0.02, 0.03, 0.9], # C_out (受V通道影响较大)
258         [0.02, 0.05, 0.02, 0.91] # X_out (受V通道影响较大)

```

```

259     ], dtype=np.float32)
260
261     Y_linear = X.dot(W.T) # 线性变换
262     # 加入非线性扰动, 使模型学习更复杂的映射
263     Y_nonlinear = Y_linear + 0.02 * np.sin(5 * np.pi * X[:, 0:1])
264     Y_nonlinear = np.clip(Y_nonlinear, 0, 1) # 确保颜色值在0-1范围内
265     return X, Y_nonlinear.astype(np.float32)
266
267 # 训练模型
268 def train_model(X, Y, epochs=100, batch_size=32, lr=1e-3):
269     # 设置设备为GPU(如果可用)或CPU
270     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
271     model = ColorNet().to(device) # 将模型移到指定设备
272     optimizer = optim.AdamW(model.parameters(), lr=lr) # 使用AdamW优化器
273     # 划分训练集和验证集
274     X_train, X_val, Y_train, Y_val = train_test_split(X, Y, test_size=0.2,
275                                                         ↪ random_state=42)
276     train_losses, val_losses = [], []
277
278     loss_fn = CombinedLoss(alpha=0.1, beta=1.0).to(device) # 初始化混合损失函
279                                                         ↪ 数
280
281     for epoch in range(epochs):
282         model.train() # 设置模型为训练模式
283         permutation = np.random.permutation(len(X_train)) # 随机打乱训练数据
284         epoch_loss = 0
285         for i in range(0, len(X_train), batch_size):
286             indices = permutation[i:i+batch_size]
287             batch_x = torch.tensor(X_train[indices], dtype=torch.float32,
288                                     ↪ device=device)
289             batch_y = torch.tensor(Y_train[indices], dtype=torch.float32,
290                                     ↪ device=device)
291
292             optimizer.zero_grad() # 梯度清零
293             outputs = model(batch_x) # 前向传播
294
295             loss = loss_fn(outputs, batch_y) # 计算损失
296
297             loss.backward() # 反向传播
298             optimizer.step() # 更新模型参数
299             epoch_loss += loss.item() * len(indices)
300         avg_train_loss = epoch_loss / len(X_train)
301         train_losses.append(avg_train_loss)
302
303     model.eval() # 设置模型为评估模式
304     with torch.no_grad(): # 禁用梯度计算
305         val_x = torch.tensor(X_val, dtype=torch.float32, device=device)
306         val_y = torch.tensor(Y_val, dtype=torch.float32, device=device)

```



```

303         val_pred = model(val_x)
304
305         val_loss = loss_fn(val_pred, val_y).item()
306         val_losses.append(val_loss)
307
308         if (epoch + 1) % 10 == 0 or epoch == 0:
309             print(f"Epoch {epoch+1}/{epochs} - Train Loss: {avg_train_loss:.6f}
310                   ↪ } - Val Loss: {val_loss:.6f}")
311
312     return model, train_losses, val_losses, X_val, Y_val
313
314 # 可视化DeltaE2000误差分布
315 def visualize_errors(model, X_val, Y_val):
316     device = next(model.parameters()).device
317     model.eval() # 设置模型为评估模式
318     with torch.no_grad(): # 禁用梯度计算
319         inputs = torch.tensor(X_val, dtype=torch.float32, device=device)
320         outputs = model(inputs).cpu().numpy() # 获取模型输出并转为NumPy数组
321         targets = Y_val
322
323     pred_rgb = outputs[:, :3] # 提取预测的RGB分量
324     target_rgb = targets[:, :3] # 提取目标的RGB分量
325
326     # 将RGB转换为Lab, 并计算DeltaE2000
327     pred_lab = rgb_to_lab(pred_rgb)
328     target_lab = rgb_to_lab(target_rgb)
329     delta_e = deltaE2000(pred_lab, target_lab)
330
331     plt.figure(figsize=(6,4))
332     plt.hist(delta_e, bins=50, color='skyblue', range=(0, max(5, np.max(
333         ↪ delta_e))))
334     plt.title('ΔE2000 Error Histogram (Trained with Hybrid Loss)')
335     plt.xlabel('ΔE2000')
336     plt.ylabel('Pixel Count')
337     plt.show()
338
339     sorted_de = np.sort(delta_e)
340     cdf = np.arange(len(sorted_de)) / float(len(sorted_de))
341
342     plt.figure(figsize=(6,4))
343     plt.plot(sorted_de, cdf, color='green')
344     plt.title('CDF of ΔE2000 Error (Trained with Hybrid Loss)')
345     plt.xlabel('ΔE2000')
346     plt.ylabel('Cumulative Probability')
347     plt.grid(True)
348     plt.show()
349
350 # 绘制色度图和色域三角形

```

```

349 def plot_chromaticity_with_triangles(example_dict):
350     # 使用colour库绘制CIE 1931色度图
351     figure, axes = colour.plotting.plot_chromaticity_diagram_CIE1931(
        ↪ standalone=False)
352
353     colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
354     linestyle_solid = '-'
355     linestyle_dashed = '--'
356
357     # 遍历字典中的每个色域并绘制
358     for i, (label, triangle) in enumerate(example_dict.items()):
359         triangle = np.array(triangle)
360
361         # 闭合多边形以绘制三角形
362         polygon = np.vstack([triangle, triangle[0]])
363         linestyle = linestyle_solid if i < 2 else linestyle_dashed # 前两个用
        ↪ 实线, 后面用虚线
364
365         # 绘制色域边界
366         axes.plot(polygon[:, 0], polygon[:, 1],
367                 color=colors[i % len(colors)],
368                 linewidth=2,
369                 linestyle=linestyle,
370                 label=label)
371
372         # 绘制每个基色点
373         point_labels = ['R', 'G', 'B', 'V'] if 'Input' in label else ['R', 'G'
        ↪ , 'C', 'B', 'X']
374         for j, point in enumerate(triangle):
375             axes.scatter(point[0], point[1], color=colors[i % len(colors)], s
        ↪ =70, zorder=5, edgecolors='black')
376             # 标记基色点
377             axes.text(point[0] + 0.005, point[1] + 0.005, f'{point_labels[j]}',
        ↪ , fontsize=10, color='black')
378
379         axes.legend() # 显示图例
380         axes.set_title("CIE 1931 Chromaticity Diagram with Multi-Primary Gamuts")
381         plt.grid(True)
382         plt.show()
383
384 # 辅助函数: 将RGBV数据转换为用于显示的RGB
385 def _rgbv_to_rgb_display(rgbv):
386     if rgbv.ndim == 1: # 如果是单个样本
387         r, g, b, v = rgbv
388     else: # 如果是批量样本
389         r, g, b, v = rgbv[:, 0], rgbv[:, 1], rgbv[:, 2], rgbv[:, 3]
390
391     # 简单的融合V通道到RGB, 用于可视化

```

```

392     r_display = np.clip(r + v * 0.1, 0, 1)
393     g_display = np.clip(g, 0, 1)
394     b_display = np.clip(b + v * 0.2, 0, 1)
395
396     return np.stack([r_display, g_display, b_display], axis=-1)
397
398 # 辅助函数：将RGBCX数据转换为用于显示的RGB
399 def _rgbcx_to_rgb_display(rgbcx):
400     if rgbcx.ndim == 1: # 如果是单个样本
401         r, g, b, c, x = rgbcx
402     else: # 如果是批量样本
403         r, g, b, c, x = rgbcx[:, 0], rgbcx[:, 1], rgbcx[:, 2], rgbcx[:, 3],
            ↪ rgbcx[:, 4]
404
405     # 简化的C和X通道融合到RGB，用于可视化
406     # C (Cyan) 影响 G 和 B
407     # X (Extra Red) 影响 R
408     r_display = np.clip(r + x * 0.3, 0, 1) # X通道增加红色
409     g_display = np.clip(g + c * 0.2, 0, 1) # C通道增加绿色
410     b_display = np.clip(b + c * 0.3, 0, 1) # C通道增加蓝色
411
412     return np.stack([r_display, g_display, b_display], axis=-1)
413
414 # 可视化部分样本的预测结果
415 def visualize_sample_predictions(model, X_val, Y_val, num_samples=5):
416     device = next(model.parameters()).device
417     model.eval() # 设置模型为评估模式
418
419     # 随机选择num_samples个样本
420     indices = np.random.choice(len(X_val), num_samples, replace=False)
421
422     fig, axes = plt.subplots(num_samples, 3, figsize=(9, 2 * num_samples))
423     fig.suptitle('Sample Color Predictions (Input RGBV -> Output RGBCX)',
            ↪ fontsize=16)
424
425     for i, idx in enumerate(indices):
426         input_rgbv = X_val[idx]
427         target_rgbcx = Y_val[idx]
428
429         # 模型预测
430         with torch.no_grad():
431             pred_rgbcx_tensor = model(torch.tensor(input_rgbv, dtype=torch.
            ↪ float32, device=device).unsqueeze(0))
432             pred_rgbcx = pred_rgbcx_tensor.squeeze(0).cpu().numpy() # 获取预测
            ↪ 结果并转为NumPy数组
433
434         # 将RGBV转换为RGB用于显示（简化处理）
435         display_input_rgb = _rgbv_to_rgb_display(input_rgbv)

```

```

436
437     # 将RGBCX转换为RGB用于显示（简化处理）
438     display_target_rgb = _rgbcx_to_rgb_display(target_rgbcx)
439     display_pred_rgb = _rgbcx_to_rgb_display(pred_rgbcx)
440
441     # 计算显示用的RGB之间的DeltaE2000
442     delta_e = deltaE2000(rgb_to_lab(display_pred_rgb), rgb_to_lab(
443         ↪ display_target_rgb))
444
445     # 绘制输入、目标和预测的颜色块
446     ax = axes[i, 0]
447     ax.imshow([[display_input_rgb]]) # imshow需要2D数组，所以用[[color]]
448     ax.set_title(f'Input (RGBV)\nSample {idx}', fontsize=8)
449     ax.axis('off')
450
451     ax = axes[i, 1]
452     ax.imshow([[display_target_rgb]])
453     ax.set_title(f'Target (RGBCX>RGB)', fontsize=8)
454     ax.axis('off')
455
456     ax = axes[i, 2]
457     ax.imshow([[display_pred_rgb]])
458     ax.set_title(f'Predicted (RGBCX>RGB)\nΔE2000: {delta_e:.2f}',
459         ↪ fontsize=8)
460     ax.axis('off')
461
462     plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # 调整布局
463     plt.savefig('sample_predictions.png', dpi=300) # 保存图片
464     plt.show()
465
466 # 主程序
467 if __name__ == '__main__':
468     X, Y = generate_train_data(n_samples=4000) # 生成训练数据
469
470     # 训练模型
471     model, train_losses, val_losses, X_val, Y_val = train_model(X, Y, epochs
472         ↪ =200, lr=5e-4)
473
474     # 绘制训练损失和验证损失曲线
475     plt.plot(train_losses, label='Train Loss')
476     plt.plot(val_losses, label='Validation Loss')
477     plt.xlabel('Epoch')
478     plt.ylabel('Combined Loss (0.1*MSE + 1.0*ΔE2000)')
479     plt.legend()
480     plt.title('Training Loss Curve (Hybrid Loss)')
481     plt.show()
482
483     # 定义标准sRGB的R,G,B基色坐标

```

```

481 PRIMARY_R = [0.64, 0.33]
482 PRIMARY_G = [0.30, 0.60]
483 PRIMARY_B = [0.15, 0.06]
484
485 # 定义相机新增的 'V' (Violet/紫色) 基色坐标
486 # 选择一个在蓝色和光谱轨迹紫色区域之间的点
487 PRIMARY_V = [0.16, 0.03]
488
489 # 定义LED屏新增的 'C' (Cyan/青色) 和 'X' (假设为一种更深的红色) 基色坐标
490 # 选择一个能扩展蓝绿边界的青色点
491 PRIMARY_C = [0.18, 0.45]
492 # 选择一个比sRGB的R更红的点, 以扩展红色边界
493 PRIMARY_X = [0.70, 0.30]
494
495 # 组合成输入和输出系统的基色字典
496 input_system primaries_coords = {
497     'RGBV Input Gamut': [PRIMARY_R, PRIMARY_G, PRIMARY_B, PRIMARY_V]
498 }
499
500 output_system primaries_coords = {
501     'RGBCX Output Gamut': [PRIMARY_R, PRIMARY_G, PRIMARY_C, PRIMARY_B,
502                             ↪ PRIMARY_X]
503 }
504
505 # 合并所有色域数据用于绘图
506 all_gamuts_for_plotting = {**input_system primaries_coords, **
507                             ↪ output_system primaries_coords}
508
509 # 可视化DeltaE2000误差分布
510 visualize_errors(model, X_val, Y_val)
511
512 # 绘制色度图和色域三角形
513 plot_chromaticity_with_triangles(all_gamuts_for_plotting)
514
515 # 可视化部分样本的预测效果
516 visualize_sample_predictions(model, X_val, Y_val, num_samples=8)

```

C. 问题 3 使用代码

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 from scipy.optimize import minimize, differential_evolution
5
6 # 设置中文字体
7 plt.rcParams['font.sans-serif'] = ['SimHei', 'DejaVu Sans']
8 plt.rcParams['axes.unicode_minus'] = False
9

```

```

10 class LEDColorCorrection:
11     """
12     基于三基色原理和CIE Lab色彩空间的颜色校正
13     使用差分进化算法优化校正矩阵
14     """
15
16     def __init__(self):
17         self.correction_matrix = None
18         self.correction_bias = None
19         self.gamma_correction = None
20         self.measured_data = None
21         self.target_data = None
22
23     def load_excel_data(self, excel_path):
24         """从Excel文件加载数据"""
25         print(f"正在加载Excel文件: {excel_path}")
26
27         sheets = ['R', 'G', 'B', 'target_R', 'target_G', 'target_B']
28         data_dict = {}
29
30         for sheet_name in sheets:
31             df = pd.read_excel(excel_path, sheet_name=sheet_name, header=None)
32             ↪ .iloc[0:64,0:64]
33             data_dict[sheet_name] = df.values
34             print(f"已加载工作表 '{sheet_name}': {df.shape}")
35
36         # 组织数据
37         self.measured_data = np.stack([
38             data_dict['R'],
39             data_dict['G'],
40             data_dict['B']
41         ], axis=-1)
42
43         self.target_data = np.stack([
44             data_dict['target_R'],
45             data_dict['target_G'],
46             data_dict['target_B']
47         ], axis=-1)
48
49         print(f"测量数据形状: {self.measured_data.shape}")
50         print(f"目标数据形状: {self.target_data.shape}")
51
52     def rgb_to_xyz(self, rgb):
53         """RGB转XYZ色彩空间"""
54         rgb_norm = rgb / 255.0
55
56         # Gamma校正
57         rgb_linear = np.where(rgb_norm <= 0.04045,

```

```

57         rgb_norm / 12.92,
58         np.power((rgb_norm + 0.055) / 1.055, 2.4))
59
60     # sRGB到XYZ的转换矩阵
61     transform_matrix = np.array([
62         [0.4124564, 0.3575761, 0.1804375],
63         [0.2126729, 0.7151522, 0.0721750],
64         [0.0193339, 0.1191920, 0.9503041]
65     ])
66
67     xyz = np.dot(rgb_linear, transform_matrix.T)
68     return xyz
69
70 def xyz_to_lab(self, xyz):
71     """XYZ转CIE Lab色彩空间"""
72     # D65白点
73     Xn, Yn, Zn = 0.95047, 1.00000, 1.08883
74
75     x = xyz[...] / Xn
76     y = xyz[...] / Yn
77     z = xyz[...] / Zn
78
79     # 立方根变换
80     fx = np.where(x > 0.008856, np.power(x, 1/3), (7.787 * x + 16/116))
81     fy = np.where(y > 0.008856, np.power(y, 1/3), (7.787 * y + 16/116))
82     fz = np.where(z > 0.008856, np.power(z, 1/3), (7.787 * z + 16/116))
83
84     L = 116 * fy - 16
85     a = 500 * (fx - fy)
86     b = 200 * (fy - fz)
87
88     return np.stack([L, a, b], axis=-1)
89
90 def calculate_color_difference(self, lab1, lab2):
91     """计算CIE Delta E 2000色差"""
92     L1, a1, b1 = lab1[...] , lab1[...] , lab1[...]
93     L2, a2, b2 = lab2[...] , lab2[...] , lab2[...]
94
95     C1 = np.sqrt(a1**2 + b1**2)
96     C2 = np.sqrt(a2**2 + b2**2)
97     C_bar = 0.5 * (C1 + C2)
98
99     G = 0.5 * (1 - np.sqrt(C_bar**7 / (C_bar**7 + 25**7)))
100     a1p = (1 + G) * a1
101     a2p = (1 + G) * a2
102
103     C1p = np.sqrt(a1p**2 + b1**2)
104     C2p = np.sqrt(a2p**2 + b2**2)

```

```

105
106     h1p = np.degrees(np.arctan2(b1, a1p)) % 360
107     h2p = np.degrees(np.arctan2(b2, a2p)) % 360
108
109     dLp = L2 - L1
110     dCp = C2p - C1p
111
112     dhp = h2p - h1p
113     dhp = dhp - 360 * (dhp > 180) + 360 * (dhp < -180)
114     dHp = 2 * np.sqrt(C1p * C2p) * np.sin(np.radians(dhp / 2))
115
116     L_bar = 0.5 * (L1 + L2)
117     C_bar_p = 0.5 * (C1p + C2p)
118
119     h_bar_p = (h1p + h2p + 360 * (np.abs(h1p - h2p) > 180)) / 2
120     h_bar_p %= 360
121
122     T = (1
123         - 0.17 * np.cos(np.radians(h_bar_p - 30))
124         + 0.24 * np.cos(np.radians(2 * h_bar_p))
125         + 0.32 * np.cos(np.radians(3 * h_bar_p + 6))
126         - 0.20 * np.cos(np.radians(4 * h_bar_p - 63)))
127
128     S1 = 1 + (0.015 * (L_bar - 50)**2) / np.sqrt(20 + (L_bar - 50)**2)
129     Sc = 1 + 0.045 * C_bar_p
130     Sh = 1 + 0.015 * C_bar_p * T
131
132     delta_theta = 30 * np.exp(-((h_bar_p - 275)/25)**2)
133     Rc = 2 * np.sqrt(C_bar_p**7 / (C_bar_p**7 + 25**7))
134     Rt = -np.sin(np.radians(2 * delta_theta)) * Rc
135
136     dE = np.sqrt(
137         (dLp / S1)**2 +
138         (dCp / Sc)**2 +
139         (dHp / Sh)**2 +
140         Rt * (dCp / Sc) * (dHp / Sh)
141     )
142
143     return dE
144
145 def estimate_gamma_parameters(self):
146     """估计LED的Gamma参数（保留线性比例偏移）"""
147     print("正在估计Gamma参数...")
148     gamma_params = {}
149     for i, channel in enumerate(['R', 'G', 'B']):
150         meas = self.measured_data[... , i].flatten() / 255.0
151         targ = self.target_data[... , i].flatten() / 255.0
152         mask = (targ > 0.04) & (targ < 0.96) & (meas > 0)

```



```

153         m = meas[mask]
154         t = targ[mask]
155         if len(m) > 0:
156             # 拟合  $\log(m) = \gamma * \log(t) + \text{offset}$ 
157             A = np.vstack([np.log(t + 1e-8), np.ones_like(t)]).T
158             gamma, offset = np.linalg.lstsq(A, np.log(m + 1e-8), rcond=
                 ↪ None)[0]
159             gamma = float(np.clip(gamma, 0.1, 3.0))
160             scale = float(np.exp(offset))
161         else:
162             gamma, scale = 1.0, 1.0
163         gamma_params[channel] = {'gamma': gamma, 'scale': scale}
164         print(f"{channel} 通道 Gamma: {gamma:.3f}, Scale: {scale:.3f}")
165     self.gamma_correction = gamma_params
166     return gamma_params
167
168 def apply_gamma_correction(self, rgb_data, inverse=False):
169     """应用Gamma校正: 在归一化 [0,1] 空间先应用线性比例, 再做幂运算"""
170     if self.gamma_correction is None:
171         return rgb_data
172     data = rgb_data.astype(np.float32) / 255.0
173     out = np.zeros_like(data)
174     for i, channel in enumerate(['R', 'G', 'B']):
175         gamma = self.gamma_correction[channel]['gamma']
176         scale = self.gamma_correction[channel]['scale']
177         ch = data[..., i]
178         if not inverse:
179             # 前向: 先比例, 再幂
180             tmp = ch * scale
181             tmp = np.clip(tmp, 0.0, 1.0)
182             out_ch = np.power(tmp, gamma)
183         else:
184             # 反向: 开幂, 再去比例
185             tmp = np.power(ch, 1.0 / gamma)
186             out_ch = tmp / np.maximum(scale, 1e-8)
187         out[..., i] = np.clip(out_ch, 0.0, 1.0)
188     # 恢复到 [0,255]
189     return (out * 255.0).astype(rgb_data.dtype)
190
191 def correction_function(self, params, measured_lin, target_lin):
192     """
193     优化函数: 线性校正矩阵 M 和偏置 b, params 长度 12。
194     corrected = clip(M @ measured + b, [0,1])
195     计算  $\Delta E$  + 正则化。
196     """
197     M = params[:9].reshape(3,3)
198     b = params[9:].reshape(1,3)
199

```

```

200     # 应用矩阵和偏置
201     corr = np.dot(measured_lin, M.T) + b
202     corr = np.clip(corr, 0.0, 1.0)
203
204     # 转到 XYZ → Lab
205     transform = np.array([[0.4124564, 0.3575761, 0.1804375],
206                           [0.2126729, 0.7151522, 0.0721750],
207                           [0.0193339, 0.1191920, 0.9503041]])
208     tgt_xyz = np.dot(target_lin, transform.T)
209     corr_xyz = np.dot(corr, transform.T)
210     tgt_lab = self.xyz_to_lab(tgt_xyz.reshape(-1, 3)).reshape(corr.shape)
211     corr_lab = self.xyz_to_lab(corr_xyz.reshape(-1, 3)).reshape(corr.shape)
212
213     # 色差
214     deltaE = self.calculate_color_difference(tgt_lab, corr_lab)
215     loss = np.mean(deltaE)
216
217     # 矩阵正则 + 偏置正则
218     loss += 0.001 * (np.sum((M - np.eye(3))**2) + np.sum(b**2))
219     det = np.linalg.det(M)
220     if det <= 0 or abs(det) < 0.1:
221         loss += 1000.0
222     return loss
223
224
225 def calibrate_correction_matrix(self):
226     print("开始校正: 矩阵 + 偏置...")
227     self.estimate_gamma_parameters()
228     # 预处理: 线性化
229     meas = self.apply_gamma_correction(self.measured_data.astype(np.
230                                     ↪ float32), inverse=True)/255.0
231     targ = self.apply_gamma_correction(self.target_data.astype(np.float32)
232                                     ↪ , inverse=True)/255.0
233     meas_flat = meas.reshape(-1, 3)
234     targ_flat = targ.reshape(-1, 3)
235     # 差分进化优化 12 参数
236     bounds = [(-2, 2)]*9 + [(-0.1, 0.1)]*3
237     res = differential_evolution(
238         self.correction_function, bounds,
239         args=(meas_flat, targ_flat), maxiter=200, popsize=15, seed=42
240     )
241     x0 = res.x
242     # 局部 L-BFGS-B
243     local = minimize(
244         self.correction_function, x0, args=(meas_flat, targ_flat),
245         method='L-BFGS-B', options={'maxiter': 500}
246     )
247     M_opt = local.x[:9].reshape(3, 3)

```

```

246         b_opt = local.x[9:].reshape(3)
247         self.correction_matrix = M_opt
248         self.correction_bias = b_opt
249         print("校正完成; 矩阵行列式: ", np.linalg.det(M_opt))
250         print("偏置: ", b_opt)
251         return M_opt, b_opt
252
253
254     def apply_correction(self, input_rgb):
255         """应用带偏置的线性校正"""
256         lin = self.apply_gamma_correction(input_rgb.astype(np.float32),
257             ↪ inverse=True)/255.0
258         flat = lin.reshape(-1,3)
259         corr = np.dot(flat, self.correction_matrix.T) + self.correction_bias
260         corr = np.clip(corr, 0.0, 1.0).reshape(input_rgb.shape)
261         out = (corr * 255.0).astype(np.float32)
262         final = self.apply_gamma_correction(out, inverse=False)
263         return final.astype(np.uint8)
264
265     def evaluate_correction(self):
266         """评估校正效果"""
267         corrected = self.apply_correction(self.measured_data.astype(np.float32)
268             ↪ )
269
270         measured_xyz = self.rgb_to_xyz(self.measured_data.astype(np.float32))
271         corrected_xyz = self.rgb_to_xyz(corrected.astype(np.float32))
272         target_xyz = self.rgb_to_xyz(self.target_data.astype(np.float32))
273
274         measured_lab = self.xyz_to_lab(measured_xyz)
275         corrected_lab = self.xyz_to_lab(corrected_xyz)
276         target_lab = self.xyz_to_lab(target_xyz)
277
278         diff_before = self.calculate_color_difference(measured_lab, target_lab
279             ↪ )
280         diff_after = self.calculate_color_difference(corrected_lab, target_lab
281             ↪ )
282
283         print("="*50)
284         print("校正效果评估报告")
285         print("="*50)
286         print(f"校正前平均色差: {np.mean(diff_before):.3f}")
287         print(f"校正后平均色差: {np.mean(diff_after):.3f}")
288         print(f"色差改善: {np.mean(diff_before) - np.mean(diff_after):.3f}")
289         print(f"改善百分比: {((np.mean(diff_before) - np.mean(diff_after)) /
290             ↪ np.mean(diff_before) * 100):.1f}%")
291         print(f"校正前最大色差: {np.max(diff_before):.3f}")
292         print(f"校正后最大色差: {np.max(diff_after):.3f}")
293         print(f"色差<1.0的像素比例: 校正前{np.mean(diff_before < 1.0)*100:.1f}

```

```

    ↪ }%, 校正后{np.mean(diff_after < 1.0)*100:.1f}%")
289 print("="*50)
290
291 return corrected, diff_before, diff_after
292
293 def visualize_results(self):
294     """可视化校正结果"""
295     corrected_data = self.apply_correction(self.measured_data.astype(np.
    ↪ float32))
296
297     fig, axes = plt.subplots(3, 4, figsize=(20, 15))
298
299     # 第一行: 测量数据
300     for i, (channel, color) in enumerate(zip(['R', 'G', 'B'], ['Reds', '
    ↪ Greens', 'Blues'])):
301         im = axes[0, i].imshow(self.measured_data[:, :, i], cmap=color,
    ↪ vmin=0, vmax=255)
302         axes[0, i].set_title(f'测量值 - {channel} 通道')
303         axes[0, i].axis('off')
304         plt.colorbar(im, ax=axes[0, i], fraction=0.046, pad=0.04)
305
306     measured_rgb = np.clip(self.measured_data / 255.0, 0, 1)
307     axes[0, 3].imshow(measured_rgb)
308     axes[0, 3].set_title('测量值 - RGB合成')
309     axes[0, 3].axis('off')
310
311     # 第二行: 目标数据
312     for i, (channel, color) in enumerate(zip(['R', 'G', 'B'], ['Reds', '
    ↪ Greens', 'Blues'])):
313         im = axes[1, i].imshow(self.target_data[:, :, i], cmap=color, vmin
    ↪ =0, vmax=255)
314         axes[1, i].set_title(f'目标值 - {channel} 通道')
315         axes[1, i].axis('off')
316         plt.colorbar(im, ax=axes[1, i], fraction=0.046, pad=0.04)
317
318     target_rgb = np.clip(self.target_data / 255.0, 0, 1)
319     axes[1, 3].imshow(target_rgb)
320     axes[1, 3].set_title('目标值 - RGB合成')
321     axes[1, 3].axis('off')
322
323     # 第三行: 校正后数据
324     for i, (channel, color) in enumerate(zip(['R', 'G', 'B'], ['Reds', '
    ↪ Greens', 'Blues'])):
325         im = axes[2, i].imshow(corrected_data[:, :, i], cmap=color, vmin
    ↪ =0, vmax=255)
326         axes[2, i].set_title(f'校正后 - {channel} 通道')
327         axes[2, i].axis('off')
328         plt.colorbar(im, ax=axes[2, i], fraction=0.046, pad=0.04)

```

```
329
330     corrected_rgb = np.clip(corrected_data / 255.0, 0, 1)
331     axes[2, 3].imshow(corrected_rgb)
332     axes[2, 3].set_title('校正后 - RGB合成')
333     axes[2, 3].axis('off')
334
335     plt.tight_layout()
336     plt.show()
337
338
339 # 主函数
340 if __name__ == "__main__":
341     files = ["MathModel_Code\\data\\preprocess\\p3\\RedPicture.xlsx", "
        ↪ MathModel_Code\\data\\preprocess\\p3\\GreenPicture.xlsx", "
        ↪ MathModel_Code\\data\\preprocess\\p3\\BluePicture.xlsx"]
342
343     corrector = LEDColorCorrection()
344
345     for filepath in files:
346         corrector.load_excel_data(filepath)
347         correction_matrix = corrector.calibrate_correction_matrix()
348
349         print("\n评估校正效果:")
350         corrected_display, diff_before, diff_after = corrector.
            ↪ evaluate_correction()
351
352         corrector.visualize_results()
353
354         print("\n校正完成!")
```

D. 像素数据集