HYBRID THEORY

An AI-powered design tool that bridges Music, Art, and Code. Hybrid Theory takes any image, triangulates it using Delaunay triangulation, and recolors it using a neural network trained on the color palette of a second image (like an album cover).

Last Updated: February 2026

Table of Contents

---

1. Installation & Setup

Prerequisites

- Python 3.8+
- pip

Install Dependencies

```
# Clone the repository
git clone https://github.com/AhmadWali04/HybridTheory.git
cd HybridTheory

# Create virtual environment
python -m venv .venv
source .venv/bin/activate    # macOS/Linux
# .venv\Scripts\activate     # Windows

# Install all dependencies
pip install -r requirements.txt
```

Dependencies (requirements.txt)

```
matplotlib
numpy
opencv-python
pillow
plotly
python-dotenv
scikit-learn
scipy
tensorboard
torch
```

Verify Installation

```
python -c "import torch, cv2, plotly, sklearn; print('All imports successful')"
```

---

2. Configuration (.env)

All configuration is managed through the `.env` file in the project root. Create one if it doesn't exist:

Sample .env File

```
# ============================================================================
# IMAGE PATHS
```

```
# ============================================================================
TEMPLATE_IMAGE=templateImages/kengan.jpeg
PALETTE_IMAGE=paletteImages/hybridTheory.jpg

# ============================================================================
# CLUSTERING PARAMETERS
# ============================================================================
NUM_CLUSTERS=25
NUM_DISTINCT=10
DENSITY_REDUCTION=5

# ============================================================================
# CNN TRAINING PARAMETERS
# ============================================================================
EPOCHS=1000
FINE_TUNE_EPOCHS=150
TEMPERATURE=0.5

# ============================================================================
# DIRECTORY PATHS (optional - defaults work for most cases)
# ============================================================================
MODELS_DIR=models
FEEDBACK_DIR=feedback_data
TENSORBOARD_DIR=runs
OUTPUT_DIR=triangulatedImages
```

Key Parameters

Parameter	Default	Description
`TEMPLATE_IMAGE`	-	Image to triangulate (the subject)
`PALETTE_IMAGE`	-	Image to extract colors from (e.g. album cover)
`NUM_CLUSTERS`	25	K-Means clusters for color extraction
`NUM_DISTINCT`	10	Final palette size after greedy selection
`DENSITY_REDUCTION`	5	Higher = fewer/larger triangles, lower = finer detail
`EPOCHS`	1000	Training iterations for the CNN
`FINE_TUNE_EPOCHS`	150	Epochs for feedback-based fine-tuning
`TEMPERATURE`	0.5	Softmax sharpness (0.1 = sharp picks, 1.0+ = soft blending)

How to Use

1. Place your template image in `templateImages/`
2. Place your palette image in `paletteImages/`
3. Update `TEMPLATE_IMAGE` and `PALETTE_IMAGE` in `.env`
4. Run `python main.py`

The model will be saved to `models/{template_name}/{template_name}_{palette_name}.pth` automatically.

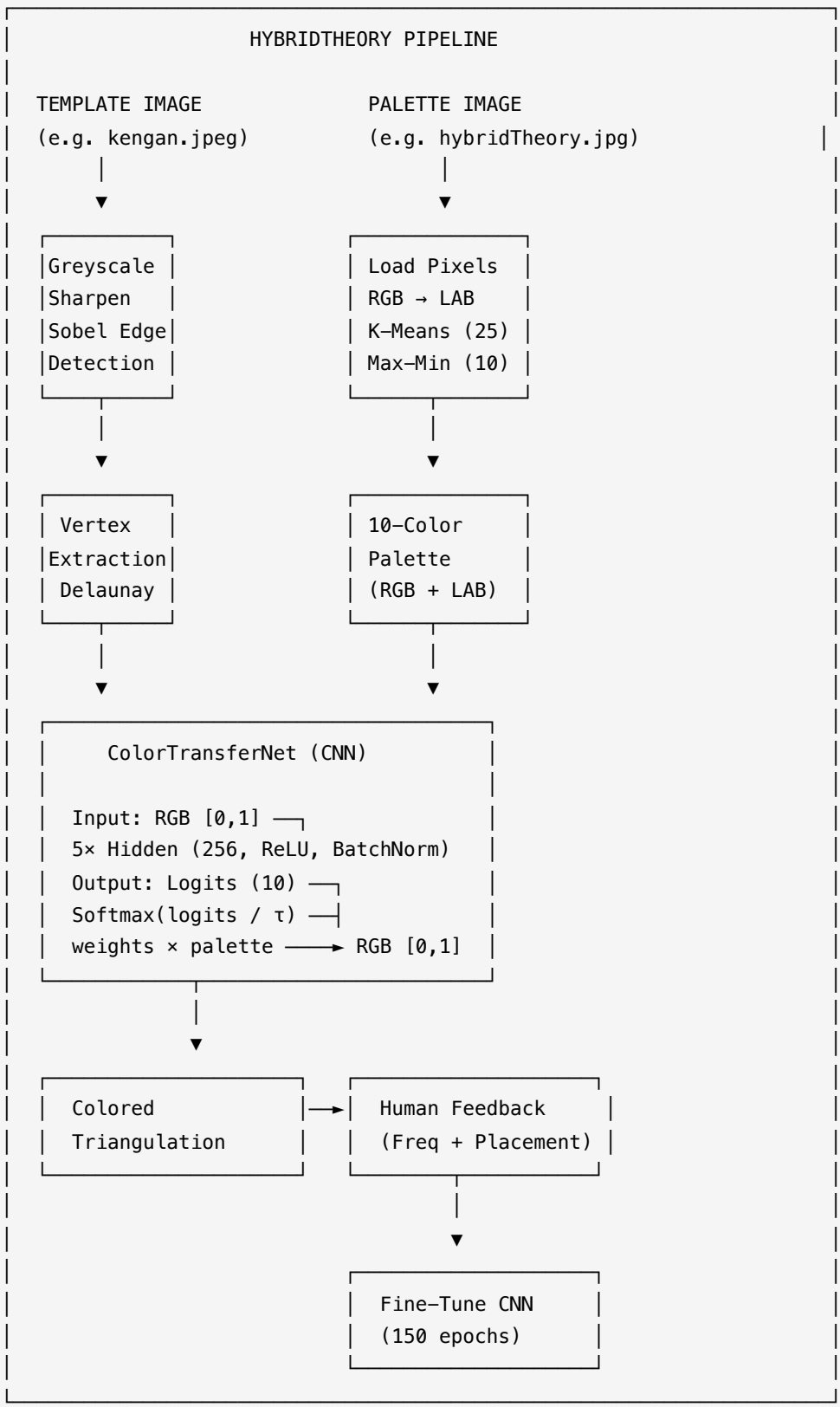
View Current Configuration

```
python config.py
```

3. System Overview

HybridTheory is a neural network-based artistic color transfer system. Given a **template image** (the subject, e.g., a character) and a **palette image** (the color source, e.g., an album cover), the system:

1. Decomposes the template image into geometric triangles using edge detection
2. Extracts a distinct color palette from the palette image
3. Trains a neural network to intelligently map the template's colors onto the extracted palette
4. Paints each triangle with the CNN's learned color mapping
5. Optionally incorporates human feedback to refine the result



Key Files

File	Role
`CNN.py`	Neural network, loss functions, training, application
`colour.py`	Color extraction (K-Means, LAB, distinct selection)
`imageTriangulation.py`	Edge detection, triangulation, full pipeline
`config.py`	`.env` loading and path generation
`tensorboard_feedback.py`	Feedback collection, TensorBoard logging
`plotting.py`	Interactive Plotly visualizations
`main.py`	Entry point with interactive menu

---

4. Image Processing Pipeline

Implemented in `imageTriangulation.py`

4.1 Greyscale Conversion

The first step converts the template image from RGB to greyscale using the **ITU-R BT.601 luminance** formula, which weights channels according to human perceptual sensitivity:

$$Y = 0.299R + 0.587G + 0.114B$$

where $R, G, B \in [0, 255]$.

Why these weights? The human eye is most sensitive to green light, less to red, and least to blue. These coefficients (derived from the NTSC television standard) ensure that the greyscale representation preserves perceived brightness.

For each pixel (i, j):

```
grey(i,j) = 0.299 × R(i,j) + 0.587 × G(i,j) + 0.114 × B(i,j)
```

The output pixel is set to $(grey, grey, grey)$ across all three channels.

4.2 Image Sharpening

A **discrete Laplacian sharpening kernel** is convolved with the greyscale image to enhance edges and fine details.

Kernel:

$$H = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This is the identity matrix plus the negative Laplacian:

$$H = I + (-\nabla^2) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Convolution operation for pixel (i, j):

$$G(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} H(m+1, n+1) \cdot I(i+m, j+n)$$

Expanding:

$$G(i,j) = -I(i,j-1) - I(i-1,j) + 5 \cdot I(i,j) - I(i+1,j) - I(i,j+1)$$

After computing all values, the output is **normalized** to $[0, 255]$:

$$I_{\text{sharp}}(i,j) = \frac{G(i,j)}{\max(G)} \times 255$$

Effect: Regions with rapid intensity changes (edges) get amplified, while uniform regions stay unchanged. The center weight of 5 preserves the original image while subtracting neighboring values enhances discontinuities.

4.3 Sobel Edge Detection

The Sobel operator estimates the **image gradient** at each pixel using two 3×3 kernels — one for horizontal changes and one for vertical changes.

Horizontal gradient kernel (∂I/∂x):

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Vertical gradient kernel (∂I/∂y):

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

For each pixel (i, j), the horizontal and vertical gradients are computed by convolution:

$$G_x(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} K_x(m+1, n+1) \cdot I(i+m, j+n)$$

$$G_y(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} K_y(m+1, n+1) \cdot I(i+m, j+n)$$

The **gradient magnitude** (edge strength) is:

$$G(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}$$

The **gradient direction** (edge orientation) is:

$$\theta(i,j) = \arctan\left(\frac{G_y(i,j)}{G_x(i,j)}\right)$$

The final edge map is normalized:

$$E(i,j) = \frac{G(i,j)}{\max(G)} \times 255$$

Intuition: G_x detects vertical edges (left-right intensity changes) and G_y detects horizontal edges (top-bottom changes). The magnitude combines both into a single edge strength value. Pixels near strong edges get high values;

flat regions get values near zero.

4.4 Vertex Extraction & Density Reduction

Vertices for triangulation are extracted from the edge map using a **thresholding** approach:

Step 1 — Threshold filter:

$$S_{\text{raw}} = \{(i, j) \mid E(i, j) > \tau\}$$

where τ is the threshold parameter (default: 50). Only pixels with edge strength above τ become vertex candidates.

Step 2 — Random subsampling (density reduction):

$$S = \text{random\_sample}\left(S_{\text{raw}}, \left\lfloor \frac{|S_{\text{raw}}|}{d} \right\rfloor\right)$$

where d is `DENSITY_REDUCTION`. Higher d = fewer vertices = fewer triangles = more abstract output.

Step 3 — Corner anchoring:
Four corner points are appended to ensure the triangulation covers the entire image:

$$S = S \cup \{(0,0),\ (0, H-1),\ (W-1, 0),\ (W, H)\}$$

4.5 Delaunay Triangulation

The vertex set S is triangulated using **Delaunay triangulation** (via `scipy.spatial.Delaunay`).

Definition: A Delaunay triangulation of a point set S is a triangulation $DT(S)$ such that no point in S is inside the circumcircle of any triangle in $DT(S)$.

Key property — Empty circumcircle: For every triangle $\triangle ABC$ in $DT(S)$, the circumscribed circle of $\triangle ABC$ contains no other points from S in its interior. This maximizes the minimum angle across all triangles, avoiding thin "sliver" triangles.

Output: An array of **simplices** — each simplex is a triple of indices (i, j, k) into the vertex array S, defining one triangle. If there are N vertices, there will be approximately $2N - 5$ triangles (by Euler's formula for planar graphs).

---

5. Color Extraction Pipeline

Implemented in `colour.py`

5.1 RGB Color Space

Each pixel is a 3D vector in RGB space:

$$\mathbf{c} = (R, G, B), \quad R, G, B \in [0, 255]$$

RGB is **not perceptually uniform** — equal Euclidean distances in RGB space do not correspond to equal perceived color differences. This is why LAB space is used for clustering.

5.2 LAB Color Space

5.2 LAB Color Space

CIE LAB (L*a*b*) is a **perceptually uniform** color space where Euclidean distance between two colors approximates the perceived visual difference.

Components:

- **L*** — Lightness (0 = black, 100 = white)
- **a*** — Green (–) to Red (+) axis
- **b*** — Blue (–) to Yellow (+) axis

Conversion from RGB to LAB:

1. **RGB → XYZ** (linear transformation via standard illuminant D65):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \cdot \begin{bmatrix} R_{\text{lin}} \\ G_{\text{lin}} \\ B_{\text{lin}} \end{bmatrix}$$

where $R_{\text{lin}} = \gamma^{-1}(R/255)$ applies inverse sRGB gamma correction:

$$\gamma^{-1}(c) = \begin{cases} \frac{c}{12.92} & \text{if } c \leq 0.04045 \\ \left(\frac{c+0.055}{1.055}\right)^{2.4} & \text{otherwise} \end{cases}$$

2. **XYZ → LAB:**

$$L^* = 116 \cdot f\left(\frac{Y}{Y_n}\right) - 16$$

$$a^* = 500 \cdot \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right]$$

$$b^* = 200 \cdot \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right]$$

where:

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

and (X_n, Y_n, Z_n) are the reference white point values for illuminant D65.

Why LAB? When we measure "how different are these two colors?" using Euclidean distance $\|\mathbf{c}_1 - \mathbf{c}_2\|_2$, the answer in LAB space closely matches human perception. In RGB, two colors can be mathematically far apart but look almost identical (or vice versa).

In the codebase, `cv2.cvtColor(image, cv2.COLOR_RGB2LAB)` performs this conversion using OpenCV's optimized implementation.

5.3 K-Means Clustering

K-Means groups all pixels into K clusters (default $K = 25$) to find the **dominant colors** in the palette image.

Algorithm:

Given pixel set $P = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N\}$ in LAB space and desired K clusters:

1. **Initialize** K cluster centroids $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$ randomly
2. **Repeat** until convergence:

E-step (Assignment): Assign each pixel to its nearest centroid:

$$c_i = \arg\min_k \|\mathbf{p}_i - \boldsymbol{\mu}_k\|_2$$

M-step (Update): Recompute each centroid as the mean of its assigned pixels:

$$\boldsymbol{\mu}_k = \frac{1}{|C_k|} \sum_{\mathbf{p}_i \in C_k} \mathbf{p}_i$$

where $C_k = \{\mathbf{p}_i \mid c_i = k\}$

3. **Output:** K cluster centers (dominant colors) and the percentage of pixels in each cluster:

$$\text{pct}_k = \frac{|C_k|}{N} \times 100\%$$

Objective function (minimized implicitly):

$$J = \sum_{k=1}^{K} \sum_{\mathbf{p}_i \in C_k} \|\mathbf{p}_i - \boldsymbol{\mu}_k\|_2^2$$

This is the **within-cluster sum of squares (WCSS)**. K-Means converges to a local minimum of J.

In the codebase, `sklearn.cluster.KMeans(n_clusters=25, n_init=10)` runs the algorithm 10 times with different random initializations and keeps the best result (lowest J).

5.4 Greedy Max-Min Distinct Color Selection

From the $K = 25$ cluster centers, we select $M = 10$ **maximally distinct** colors using a greedy algorithm. This ensures the final palette has diverse, well-separated colors rather than 10 similar shades.

Algorithm (Greedy Farthest-Point Sampling):

Given cluster centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$ in LAB space:

1. **Initialization:** Select the color with the highest **lightness** value:

$$s_1 = \arg\max_k L^*(\boldsymbol{\mu}_k)$$

This anchors the palette to the brightest color, providing consistent ordering.

2. **Iterative selection:** For $j = 2, 3, \ldots, M$:
For each unselected candidate $\boldsymbol{\mu}_c$, compute its **minimum distance** to all already-selected colors:

$$d_{\min}(c) = \min_{s \in \text{Selected}} \|\boldsymbol{\mu}_c - \boldsymbol{\mu}_s\|_2$$

Select the candidate that **maximizes** this minimum distance:

$$s_j = \arg\max_{c \notin \text{Selected}} d_{\min}(c)$$

Why this works: At each step, we pick the color that is farthest from everything already chosen. This is a greedy approximation of maximizing the **minimum pairwise distance** in the selected set — a well-known problem in computational geometry called the **k-center problem** or **dispersion problem**.

Time complexity: $O(M \cdot K)$ for M selections from K candidates, with $O(M)$ distance computations per candidate.

Example: If we have 25 cluster centers and the brightest is white, the second pick will be the color farthest from white (likely a deep dark color). The third pick will be the color farthest from both white and the dark color (likely a vivid mid-

(likely a deep dark color). The third pick will be the color farthest from both white and the dark color (likely a vivid mid-tone). This continues until we have 10 well-spread colors.

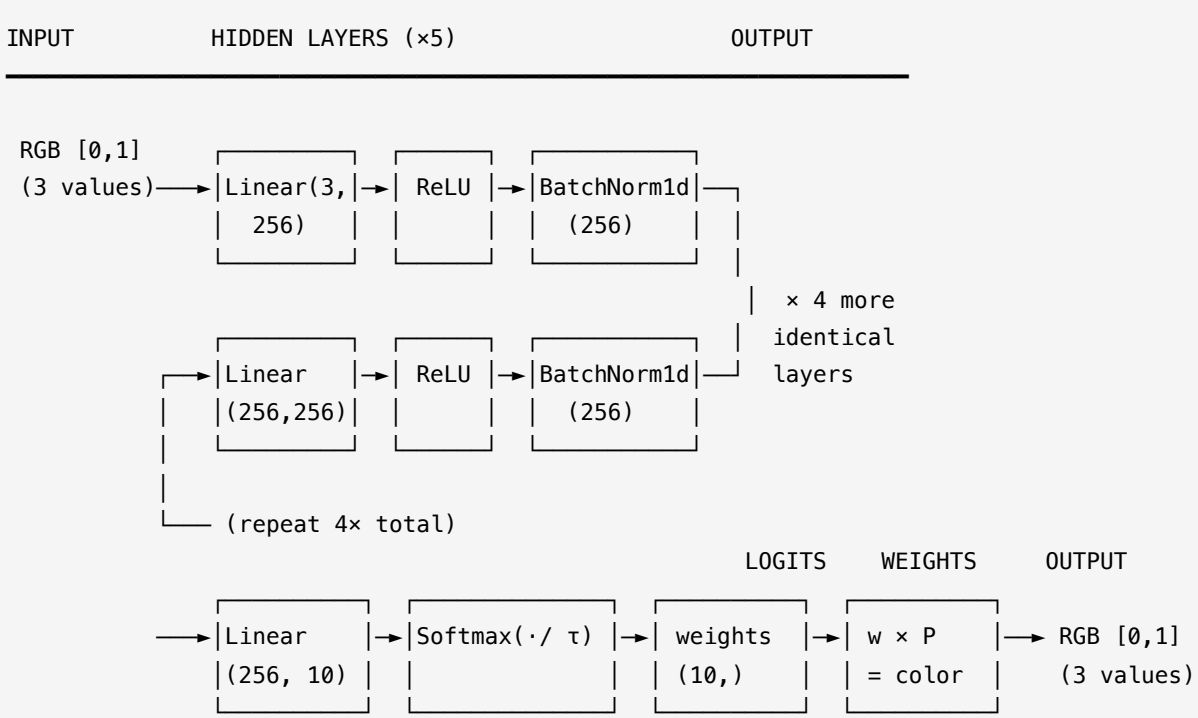
---

6. CNN Architecture — ColorTransferNet

Implemented as `class ColorTransferNet` *in* `CNN.py`

6.1 Network Structure

ColorTransferNet is a **fully connected feed-forward neural network** that learns to map individual pixel colors from the template image to the target palette.



Layer breakdown:

Layer	Input Dim	Output Dim	Parameters
Linear 1	3	256	3 × 256 + 256 = 1,024
ReLU + BatchNorm	256	256	512 (BN params)
Linear 2–5	256	256	4 × (256 × 256 + 256) = 263,168
ReLU + BatchNorm (×4)	256	256	4 × 512 = 2,048
Linear Out	256	10	256 × 10 + 10 = 2,570
Total			**~269,000 parameters**

6.2 Forward Pass Mathematics

For a single input pixel $\mathbf{x} \in \mathbb{R}^3$ (normalized RGB in $[0, 1]$):

Layer 1:

$$\mathbf{h}_1 = \text{BN}(\text{ReLU}(W_1\mathbf{x} + \mathbf{b}_1))$$

Layers 2–5 (identical structure):

$$\mathbf{h}_l = \mathrm{BN}(\mathrm{ReLU}(W_l \mathbf{h}_{l-1} + \mathbf{b}_l)), \quad l = 2, \ldots, 5$$

Output layer:

$$\mathbf{z} = W_6 \mathbf{h}_5 + \mathbf{b}_6 \in \mathbb{R}^{10}$$

where:

- W_l are weight matrices, $\mathbf{b}_l$ are bias vectors
- $\mathrm{ReLU}(x) = \max(0, x)$
- BN is Batch Normalization (described below)

ReLU activation:

$$\mathrm{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Introduces non-linearity, enabling the network to learn complex color mappings. Without it, the entire network would collapse to a single linear transformation.

Batch Normalization:

For a mini-batch $\{h_i\}_{i=1}^{B}$ of hidden activations:

$$\hat{h}_i = \frac{h_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\mathrm{BN}(h_i) = \gamma \hat{h}_i + \beta$$

where μ_B and σ_B^2 are the batch mean and variance, γ and β are learned scale and shift parameters, and $\epsilon = 10^{-5}$ prevents division by zero. BatchNorm stabilizes training by normalizing internal activations, allowing higher learning rates.

6.3 Temperature-Scaled Softmax

The logit vector $\mathbf{z} \in \mathbb{R}^{10}$ is converted to **palette assignment weights** using temperature-scaled softmax:

$$w_k = \frac{\exp(z_k / \tau)}{\sum_{j=1}^{10} \exp(z_j / \tau)}, \quad k = 1, \ldots, 10$$

where τ is the **temperature** parameter (default: 0.5).

Temperature effects:

Temperature τ	Behavior	Output Character
$\tau \to 0$	$\mathbf{w}$ approaches one-hot (argmax)	Pure palette colors, hard boundaries
$\tau = 0.1$ – 0.5	Sharp distribution, 1–2 dominant weights	Vivid, colorful, uses full palette
$\tau = 1.0$	Standard softmax	Moderate blending
$\tau \to \infty$	Uniform $w_k = 1/10$	Average of all palette colors (muddy grey)

Why $\tau = 0.5$? This is a sweet spot — low enough to make decisive color picks (each pixel gets mapped to primarily one palette color), but high enough to maintain differentiability for gradient-based training.

Mathematical intuition: Dividing logits by $\tau < 1$ amplifies the differences between them. If $z_1 = 2.0$ and $z_2 = 1.0$:

- At $\tau = 1.0$: $w_1/w_2 = e^{2-1} = e \approx 2.7$
- At $\tau = 0.5$: $w_1/w_2 = e^{(2-1)/0.5} = e^2 \approx 7.4$

Lower temperature makes the "winning" color win by a wider margin.

6.4 Palette-Constrained Output

The final output color is a **weighted sum** of the palette colors:

$$\hat{\mathbf{c}} = \sum_{k=1}^{10} w_k \cdot \mathbf{p}_k = \mathbf{w}^T P$$

where:

- $\mathbf{w} = (w_1, \ldots, w_{10})$ are the softmax weights
- $P \in \mathbb{R}^{10 \times 3}$ is the palette matrix (each row is an RGB color in $[0, 1]$)
- $\hat{\mathbf{c}} \in \mathbb{R}^3$ is the output RGB color

Critical design decision: The output is always a convex combination of palette colors ($w_k \geq 0, \sum w_k = 1$). This means the output is **constrained to the convex hull** of the palette. The network cannot invent colors outside the palette — it can only choose (or blend between) existing palette colors.

This prevents the common neural network failure mode of producing "grey averages" — when a naive regression network tries to satisfy all targets simultaneously and outputs the mean of everything.

In matrix form for a batch of N pixels:

$$\hat{C} = \text{softmax}\left(\frac{f_\theta(X)}{\tau}\right) \cdot P \in \mathbb{R}^{N \times 3}$$

where $X \in \mathbb{R}^{N \times 3}$ is the input batch and f_θ is the neural network (all layers before the final softmax).

---

7. Loss Functions (The Heuristics)

The training loss is a weighted combination of four carefully designed loss functions, each addressing a specific failure mode.

7.1 PaletteUsageLoss — Preventing Mode Collapse

Class: `PaletteUsageLoss` *in* `CNN.py`

Problem it solves: Without this loss, the network might learn to map everything to just 1–2 "safe" palette colors, ignoring the other 8. This is called **mode collapse**.

Mechanism: Compute the average softmax weight for each palette color across the batch, then penalize deviation from a uniform distribution.

Equation:

Given a batch of N pixels and their logits from the network:

1. Compute weights with **fixed temperature** $\tau = 1.0$ (not the model's temperature):

$$w_{ik} = \frac{\exp(z_{ik}/1.0)}{\sum_{j=1}^{10} \exp(z_{ij}/1.0)}$$

2. Average usage across the batch:

$$\bar{u}_k = \frac{1}{N} \sum_{i=1}^{N} w_{ik}, \quad k = 1, \ldots, 10$$

3. Target is uniform usage:

$$u_k^* = \frac{1}{10} = 0.1$$

4. Loss is **Mean Squared Error** from uniform:

$$\mathcal{L}_{\text{usage}} = \frac{1}{10} \sum_{k=1}^{10} (\bar{u}_k - u_k^*)^2$$

Why fixed $\tau = 1.0$? The model might use a very low temperature (e.g., 0.1) for sharp color picks. At such low temperatures, softmax outputs are nearly one-hot vectors, and gradients become vanishingly small for the non-dominant palette colors. Using $\tau = 1.0$ internally ensures that gradient signals flow to all palette color logits, even when the model's actual output temperature makes sharp picks.

Intuition: If the network only uses 3 colors, their average usage would be ~0.33 while the unused 7 colors would be ~0.0. The MSE from the uniform target (0.1 each) would be large, pushing the network to diversify.

7.2 NearestColorDistanceLoss — Palette Fidelity

Class: `NearestColorDistanceLoss` *in* `CNN.py`

Problem it solves: Even with palette-constrained output, blending between two distant palette colors can produce intermediate colors that don't exist in the palette. This loss encourages the network to output colors that are **close to at least one actual palette color**.

Equation:

Given output colors $\hat{C} = \{\hat{\mathbf{c}}_1, \ldots, \hat{\mathbf{c}}_N\}$ and palette $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_{10}\}$:

1. Compute pairwise distances via broadcasting:

$$D_{ik} = \|\hat{\mathbf{c}}_i - \mathbf{p}_k\|_2 = \sqrt{\sum_{j=1}^{3} (\hat{c}_{ij} - p_{kj})^2}$$

   This produces a distance matrix $D \in \mathbb{R}^{N \times 10}$.
2. For each output pixel, find the distance to its **nearest** palette color:

$$d_i^{\min} = \min_{k \in \{1, \ldots, 10\}} D_{ik}$$

3. Loss is the mean of these minimum distances:

$$\mathcal{L}_{\text{nearest}} = \frac{1}{N} \sum_{i=1}^{N} d_i^{\min}$$

Intuition: If every output color sits exactly on a palette color, $d_i^{\min} = 0$ for all i and the loss is zero. The further outputs drift from the palette (e.g., blending red and blue to get purple when purple isn't in the palette), the higher the penalty.

7.3 SmoothnessLoss — Lipschitz Regularization

Function: `compute_smoothness_loss` *in* `CNN.py`

Problem it solves: Without smoothness regularization, the network might learn discontinuous mappings — two nearly identical input colors could map to completely different palette colors, causing jarring visual artifacts in the triangulation.

Equation:

1. Sample n source pixels: $\{\mathbf{x}_i\}_{i=1}^{n}$, $n = 1000$
2. Create perturbed versions by adding Gaussian noise:

$$\tilde{\mathbf{x}}_i = \text{clamp}(\mathbf{x}_i + \boldsymbol{\epsilon}_i, 0, 1), \quad \boldsymbol{\epsilon}_i \sim \mathcal{N}(0, \sigma^2 I_3)$$

   where $\sigma = 0.01$ (1% of the color range)
3. Pass both through the model:

$$\hat{\mathbf{c}}_i = f_\theta(\mathbf{x}_i), \quad \tilde{\mathbf{c}}_i = f_\theta(\tilde{\mathbf{x}}_i)$$

4. Measure output difference:

$$\mathcal{L}_{\text{smooth}} = \frac{1}{n} \sum_{i=1}^{n} \|\hat{\mathbf{c}}_i - \tilde{\mathbf{c}}_i\|_2$$

Connection to Lipschitz continuity: This is a stochastic approximation of the **Lipschitz constant** of the network. A function f is L-Lipschitz if:

$$\|f(\mathbf{x}) - f(\mathbf{y})\|_2 \leq L \cdot \|\mathbf{x} - \mathbf{y}\|_2 \quad \forall \mathbf{x}, \mathbf{y}$$

The smoothness loss penalizes large output differences for small input perturbations, effectively bounding the local Lipschitz constant and ensuring smooth color transitions.

7.4 PaletteEntropyLoss — Decisive Color Picks

Class: `PaletteEntropyLoss` *in* `CNN.py`

Problem it solves: The softmax might produce diffuse weight distributions (e.g., $w = [0.1, 0.1, \ldots, 0.1]$), which results in every pixel being painted with the same muddy average of all palette colors. We want **decisive** picks — each pixel should strongly prefer 1–2 palette colors.

Equation:

Using the **Shannon entropy** of the softmax weight distribution:

1. Compute softmax weights at the model's actual temperature:

$$w_{ik} = \frac{\exp(z_{ik}/\tau)}{\sum_{j=1}^{10} \exp(z_{ij}/\tau)}$$

2. Compute entropy for each pixel:

$$H_i = -\sum_{k=1}^{10} w_{ik} \log(w_{ik} + \epsilon)$$

   where $\epsilon = 10^{-8}$ prevents $\log(0)$.

3. Loss is the mean entropy:

$$\mathcal{L}_{\text{entropy}} = \frac{1}{N}\sum_{i=1}^{N} H_i$$

Entropy values:

- **Minimum** ($H = 0$): One weight is 1.0, rest are 0.0. Perfect one-hot pick of a single palette color.
- **Maximum** ($H = \log(10) \approx 2.30$): All weights equal $1/10$. Complete uncertainty — muddy average.

Minimizing entropy pushes the network toward sharp, decisive color assignments.

Note: This loss may seem to conflict with `PaletteUsageLoss` (which wants uniform global usage). They operate at different levels — entropy wants each **individual pixel** to pick one color decisively, while usage wants the **aggregate** across all pixels to be balanced. Both can be satisfied simultaneously if different pixels pick different colors.

7.5 Composite Loss Function

The total training loss is a weighted sum:

$$\mathcal{L}_{\text{total}} = \underbrace{1.0 \cdot \mathcal{L}_{\text{usage}}}_{\text{diversity}} + \underbrace{0.2 \cdot \mathcal{L}_{\text{nearest}}}_{\text{fidelity}} + \underbrace{0.05 \cdot \mathcal{L}_{\text{smooth}}}_{\text{continuity}} + \underbrace{0.3 \cdot \mathcal{L}_{\text{entropy}}}_{\text{decisiveness}}$$

Weight rationale:

Loss	Weight	Why
Usage	1.0	Most critical — without diversity, the output is monotone
Entropy	0.3	Important — prevents muddy blending
Nearest Color	0.2	Moderate — soft constraint toward actual palette colors
Smoothness	0.05	Light touch — too much smoothness prevents the network from making color boundaries

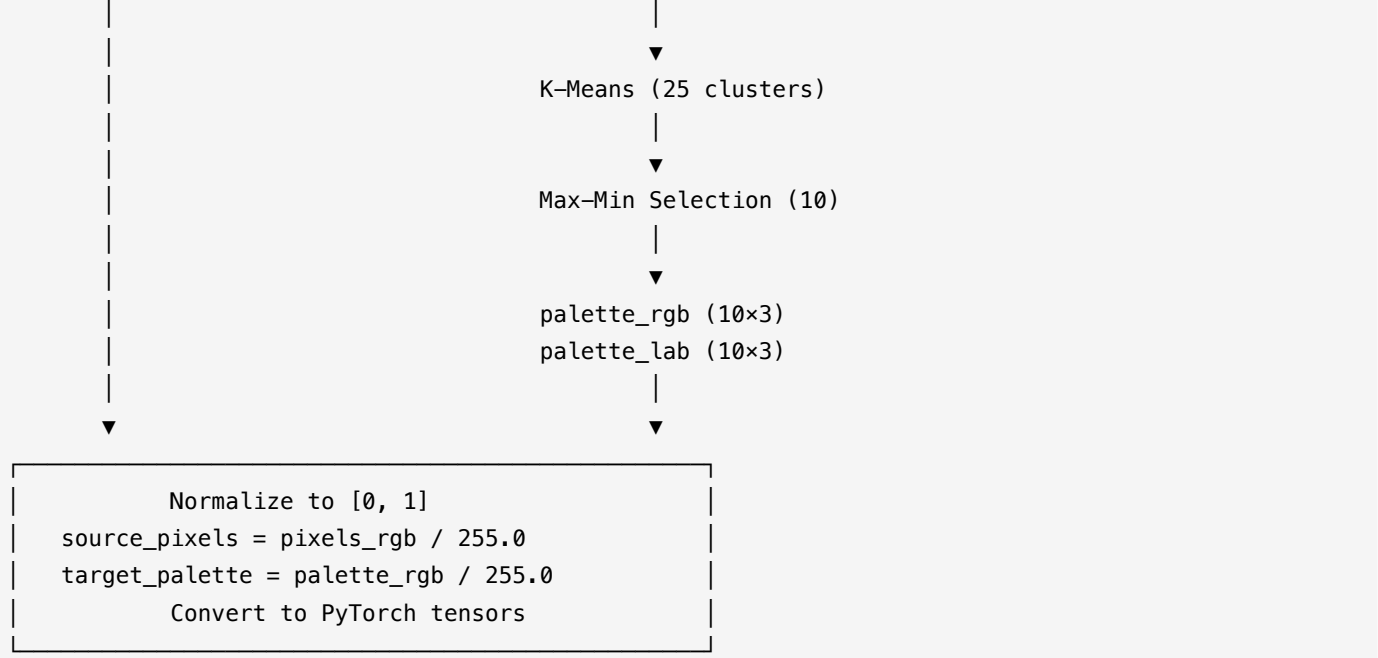
---

8. Training Pipeline

8.1 Data Preparation

Function: `prepare_training_data` *in* `CNN.py`

```
Template Image (kengan.jpeg)          Palette Image (hybridTheory.jpg)
       │                                     │
       ▼                                     ▼
load_image_pixels()                   load_image_pixels()
       │                                     │
       ▼                                     ▼
pixels_rgb (N×3)                      pixels_rgb (M×3)
[0–255] uint8                         [0–255] uint8
       │                                     │
       ├──► convert to LAB                   ├──► convert to LAB
       │    pixels_lab (N×3)                 │    pixels_lab (M×3)
```



The source image provides **all of its pixels** as training data. Each pixel is an independent sample — the network learns a color-to-color mapping, not a spatial mapping. The palette provides the **10 target colors** the network should map toward.

8.2 Training Loop

For each epoch $t = 1, \ldots, 1000$:

1. **Sample batch:** Randomly select $B = 512$ pixels from the source:

$$\text{batch} = \text{source\_pixels}[\text{randperm}(N)[:B]]$$

2. **Forward pass:** Compute output colors:

$$\hat{C} = \text{model}(\text{batch})$$

3. **Compute losses:** Evaluate all four loss components on the batch
4. **Backward pass:** Compute gradients via backpropagation:

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial \theta}$$

   where θ represents all network parameters (weights, biases, BN parameters)
5. **Optimizer step:** Update parameters using Adam (see next section)
6. **Log:** Record loss values for visualization

8.3 Adam Optimizer

Adam (**Ada**ptive **M**oment Estimation) is used with learning rate $\eta = 0.001$.

For each parameter θ:

1. **First moment (mean of gradients):**

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

2. **Second moment (mean of squared gradients):**

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. **Bias correction:**

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. **Parameter update:**

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Default hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Why Adam? It adapts the learning rate per-parameter based on gradient history. Parameters with consistently large gradients get smaller effective learning rates (preventing overshooting), while parameters with small gradients get larger rates (accelerating convergence). This is well-suited for the heterogeneous loss landscape of our composite objective.

---

9. Human-in-the-Loop Feedback System

Implemented in `CNN.py` *(loss computation) and* `tensorboard_feedback.py` *(collection)*

After the initial 1000-epoch training, the user can rate the result. The feedback drives a **fine-tuning phase** (150 epochs) that adjusts the model to match the user's preferences.

9.1 Scoring System

For each of the 10 palette colors, the user provides two ratings on a 0–9 scale:

Frequency Score — "How often does this color appear?"

Score	Meaning
0	Never appears (needs to be added)
1	Appears but rarely (too little)
5	Perfect amount
9	Overused (too much, reduce it)

Placement Score — "Where does this color appear?"

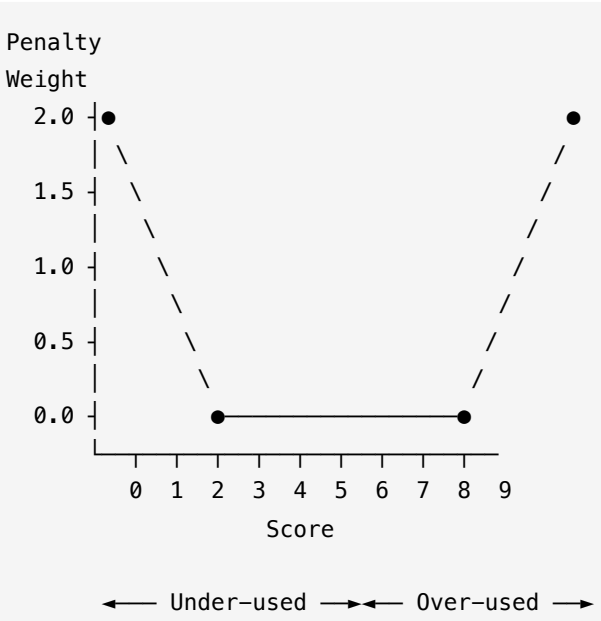
Score	Meaning
0	Not used (skip placement evaluation)
1	Used in wrong places
5	Good placement
9	Excellent placement

9.2 V-Shaped Penalty Curve

Both frequency and placement scores are converted to **penalty weights** using a symmetric V-shaped curve centered

Both frequency and placement scores are converted to **penalty weights** using a symmetric V-shaped curve centered at 5 (perfect):

$$\text{penalty}(s) = \begin{cases} 2.0 - \frac{s}{5.0} \times 2.0 & \text{if } s < 5 \quad \text{(under-used)} \\ \frac{(s-5)}{4.0} \times 2.0 & \text{if } s \geq 5 \quad \text{(over-used)} \end{cases}$$



Key values:

Score	Penalty	Interpretation
0	2.0	Maximum — color completely missing
1	1.6	High — barely appears
3	0.8	Moderate — somewhat underused
5	0.0	Perfect — no penalty
7	1.0	Moderate — somewhat overused
9	2.0	Maximum — way too dominant

Design principle: Both under-use and over-use are penalized equally, creating a **symmetric** incentive. The model is driven toward a score of 5 for every color.

9.3 Frequency Loss

Measures whether the network's output **includes** each palette color.

For each palette color $\mathbf{p}_k$:

1. Denormalize outputs and palette to $[0, 255]$:

$$\hat{C}_{255} = \hat{C} \times 255, \quad P_{255} = P \times 255$$

2. Compute distance from each output pixel to this palette color:

$$d_{ik} = \|\hat{\mathbf{c}}_{i,255} - \mathbf{p}_{k,255}\|_2$$

3. Find the closest output pixel:

$$d_k^{\min} = \min_i d_{ik}$$

4. Apply frequency penalty weight:

$$\mathcal{L}_{\text{freq}} = \frac{1}{10} \sum_{k=1}^{10} \text{penalty}(s_k^{\text{freq}}) \cdot d_k^{\min}$$

Interpretation: If a color got a low frequency score (user said "never appears"), its penalty weight is high (up to 2.0), so the loss strongly penalizes the network for not producing that color. If the color scored 5 (perfect), the penalty is 0 and the loss ignores that color.

9.4 Placement Loss

Measures whether colors are used **consistently** — placed in appropriate spatial regions rather than scattered randomly.

For each palette color $\mathbf{p}_k$ with placement score $s_k^{\text{place}} > 0$:

1. Compute distances from all outputs to this palette color:

$$d_{ik} = \|\hat{\mathbf{c}}_{i,255} - \mathbf{p}_{k,255}\|_2$$

2. Compute the **standard deviation** of these distances:

$$\sigma_k = \text{std}(\{d_{ik}\}_{i=1}^{N})$$

3. Apply placement penalty weight:

$$\mathcal{L}_{\text{place}} = \frac{1}{10} \sum_{k=1}^{10} \text{penalty}(s_k^{\text{place}}) \cdot \sigma_k$$

Intuition: If a color is well-placed, the distances to that color will have **low variance** — the pixels that use it are consistently close, and the pixels that don't are consistently far. High variance means the color is scattered incoherently.

9.5 Recency Weighting

When multiple feedback sessions exist, more recent feedback is weighted more heavily:

$$w_i^{\text{recency}} = 0.3 + \frac{i}{\max(1, n-1)} \times 0.7$$

where i is the session index (0 = oldest, $n-1$ = newest).

Session	Recency Weight
Oldest (i=0)	0.30
Middle	0.65
Newest (i=n-1)	1.00

Weighted average scores:

$$\bar{s}_k = \frac{\sum_{i=1}^{n} w_i^{\text{recency}} \cdot s_k^{(i)}}{\sum_{i=1}^{n} w_i^{\text{recency}}}$$

This applies separately to both frequency and placement scores.

This applies separately to both frequency and placement scores.

Rationale: The user's most recent feedback reflects their current preferences. Older sessions may have been before the model improved. The 0.3 floor ensures old feedback isn't completely ignored.

9.6 Combined Feedback Loss

$$\mathcal{L}_{\text{feedback}} = 0.7 \cdot \mathcal{L}_{\text{freq}} + 0.3 \cdot \mathcal{L}_{\text{place}}$$

Frequency receives 70% weight because it addresses the user's primary concern ("this color isn't appearing"). Placement receives 30% because it's a more subtle refinement.

9.7 Fine-Tuning Composite Loss

During the 150-epoch fine-tuning phase, the feedback loss is combined with the original loss components:

$$\mathcal{L}_{\text{fine-tune}} = \mathcal{L}_{\text{feedback}} + 0.3 \cdot \mathcal{L}_{\text{usage}} + 0.2 \cdot \mathcal{L}_{\text{nearest}} + 0.2 \cdot \mathcal{L}_{\text{entropy}}$$

Key differences from initial training:

- Feedback loss leads (weight 1.0) — user preferences are the priority
- Usage and entropy weights reduced (0.3, 0.2) — the model is already trained, gentle regularization suffices
- Smoothness loss dropped — the model already learned smooth transitions
- Learning rate lowered: $\eta = 0.0005$ (half of training rate) — fine adjustments, not major restructuring

---

10. Application to Triangulation

Function: `apply_cnn_to_triangulation` *in* `CNN.py`

After training, the CNN is applied to color each triangle:

```
For each triangle T in triangulation:
  │
  ├─ 1. Get vertices: v₁, v₂, v₃ = S[simplices[T]]
  │
  ├─ 2. Compute centroid:
  │     cx = (v₁ₓ + v₂ₓ + v₃ₓ) / 3
  │     cy = (v₁ᵧ + v₂ᵧ + v₃ᵧ) / 3
  │
  ├─ 3. Sample original image at centroid:
  │     rgb_original = image.getpixel(cx, cy)
  │
  ├─ 4. Normalize to [0,1]:
  │     input = rgb_original / 255.0
  │
  ├─ 5. Pass through trained CNN:
  │     output = model(input)
  │     (internally: logits → softmax/τ → weighted palette sum)
  │
  ├─ 6. Denormalize to [0,255]:
  │     rgb_output = output × 255
  │
  └─ 7. Fill triangle with rgb_output
```

Centroid formula for triangle with vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$:

$$\mathbf{c} = \frac{\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3}{3}$$

The centroid is chosen as the sampling point because it represents the "center of mass" of the triangle — a single point that best represents the triangle's spatial position in the original image.

---

11. End-to-End Data Flow

A complete trace of a single pixel through the entire system:

```
ORIGINAL IMAGE PIXEL at position (247, 183)
    Color: RGB(142, 67, 35) — a brownish skin tone
         │
         ▼
    NORMALIZE: (142/255, 67/255, 35/255) = (0.557, 0.263, 0.137)
         │
         ▼
    LINEAR LAYER 1: W₁ × [0.557, 0.263, 0.137]ᵀ + b₁
    → 256 values
         │
         ▼
    ReLU → BatchNorm → ... (4 more hidden layers)
         │
         ▼
    LINEAR OUTPUT: W₆ × h₅ + b₆
    → 10 logits: [2.1, −0.3, 0.8, −1.2, 3.4, 0.1, −0.5, 1.7, −0.8, 0.4]
         │
         ▼
    DIVIDE BY τ=0.5:
    → [4.2, −0.6, 1.6, −2.4, 6.8, 0.2, −1.0, 3.4, −1.6, 0.8]
         │
         ▼
    SOFTMAX:
    → [0.061, 0.001, 0.005, 0.000, 0.831, 0.001, 0.000, 0.027, 0.000, 0.002]
         │                            ▲
         │                    Color 5 dominates (83.1%)
         ▼
    WEIGHTED SUM with palette:
    0.061 × p₁ + 0.001 × p₂ + ... + 0.831 × p₅ + ... + 0.002 × p₁₀
         │
         │  (If p₅ = [0.82, 0.24, 0.15] — a warm red-orange)
         ▼
    OUTPUT ≈ [0.80, 0.24, 0.15] — mostly palette color 5
         │
         ▼
    DENORMALIZE: (204, 61, 38) — the triangle gets painted this warm red
```

The brownish skin tone (142, 67, 35) was mapped to a warm red (204, 61, 38) from the palette — the network learned that warm earth tones in the source should map to the warm reds in the target palette.

---

12. Glossary

Term	Definition
Batch Normalization	Technique that normalizes layer activations to zero mean and unit variance, stabilizing training
Convex hull	The smallest convex set containing all points; for colors, the set of all possible blends
Delaunay triangulation	Triangulation maximizing minimum angles; no point lies inside any triangle's circumcircle
Entropy	Measure of uncertainty/disorder in a probability distribution; $H = -\sum p \log p$
K-Means	Clustering algorithm that partitions data into K groups by minimizing within-cluster variance
LAB color space	Perceptually uniform color space where Euclidean distance ≈ perceived color difference
Lipschitz continuity	Function property bounding how fast outputs can change relative to inputs
Mode collapse	When a model ignores parts of the output space and only produces a few distinct outputs
ReLU	Rectified Linear Unit activation: $\max(0, x)$
Sobel operator	Discrete differentiation operator for estimating image intensity gradients
Softmax	Function converting logits to a probability distribution: $\sigma(z)_k = e^{z_k} / \sum e^{z_j}$
Temperature	Scalar dividing logits before softmax; controls distribution sharpness

---

Built by WALL-E