Problem Solving Question – Imaging Science

Assignment Report

Name: Bonthu Jayaram

---

1. Introduction

This assignment focuses on recovering the true texture (reflectance) of a surface from a single photograph that is unevenly illuminated. The idea is to separate the *actual surface pattern* from the *illumination effects* such as shadows, bright spots, or gradual light falloff.

The commonly used model for image formation is:

$$I(x,y) = R(x,y)\,L(x,y)$$

where

- **I(x,y)** is the observed pixel intensity,
- **R(x,y)** is the true reflectance (texture),
- **L(x,y)** is the illumination (smooth lighting variations).

The challenge is that both R and L are unknown and only their product is visible.

---

2. Why Histogram Equalization Cannot Recover Reflectance

Histogram equalization is a global point-wise operation:

$$I_{\text{new}}(x,y) = f(I_{\text{old}}(x,y))$$

It treats all pixels only based on their intensity histogram, without understanding *why* a pixel is bright or dark.

It fails to recover $R(x,y)$because:

1. No separation between lighting and texture

It cannot distinguish whether a pixel is dark due to actual texture or because it lies in a shadow.

2. No spatial or frequency awareness

Reflectance consists of **high-frequency details** (edges, patterns).
Illumination is **low-frequency** (smooth shading).
Histogram equalization ignores both properties.

3. Can distort texture

It often amplifies shadows or compresses high-frequency content, which corrupts the true surface texture.

Therefore, histogram equalization cannot extract reflectance. A frequency-based approach is needed.

---

3. Log-Domain Model for Reflectance Recovery

The multiplicative model can be converted to an additive model using logarithms:

$$\log I(x,y) = \log R(x,y) + \log L(x,y)$$

Let

- $i(x,y) = \log I(x,y)$
- $r(x,y) = \log R(x,y)$
- $l(x,y) = \log L(x,y)$

Then:

$$i(x,y) = r(x,y) + l(x,y)$$

Now, illumination becomes a **smooth, low-frequency component**, and texture becomes a **high-frequency component**.
This allows separation using simple smoothing and subtraction.

---

4. Proposed Algorithm (Retinex-style Method)

Step-by-step approach

1. Convert image to grayscale (for Question 2).
2. Normalize pixel values to [0,1].
3. Convert to log domain:

$$i(x,y) = \log(I(x,y) + \epsilon)$$

4. Estimate illumination $\hat{l}(x,y)$by applying a **large manual box filter** (no built-ins).
5. Estimate reflectance:

$$\hat{r}(x,y) = i(x,y) - \hat{l}(x,y)$$

6. Convert back using exponent:

$$\hat{R}(x,y) = \exp\left(\hat{r}(x,y)\right)$$

7. Normalize for visualization.

This produces a clear reflectance image with reduced shadows and preserved texture.

---

5. Grayscale Reflectance (Question 2)

Input Image (Your Photo)

(Insert your original grayscale or color image here)

Output Reflectance Image

(Insert your generated grayscale reflectance output here)
File: **reflectance_grayscale_output.jpg**

Observation

- The shadows cast by trees on the wall are significantly reduced.
- The brick pattern appears more uniform and clearer.
- Large-scale brightness variations are removed.
- Fine texture details of the bricks are preserved well.

This confirms successful separation of reflectance from illumination.

---

6. Color Reflectance (Question 3)

For a color image with spectral illumination variations, the model extends to RGB channels:

$$I_c(x,y) = R_c(x,y)\, L_c(x,y), c \in \{R,G,B\}$$

To maintain natural colors, a **single illumination estimate** is computed from the average log-intensity and applied uniformly across channels.

Output Reflectance Image (Color)

(Insert your generated color reflectance image here)
File: **reflectance_color_output.jpg**

Observation

- Joint illumination removal across all channels produced evenly balanced colors.
- True color ratios between bricks are preserved.

- Shadow regions no longer dominate the appearance.
- The final output looks visually consistent and faithful to the material's real texture.

---

7. Conclusion

This assignment demonstrates how a single unevenly illuminated image can be decomposed into illumination and reflectance components. By using the logarithmic model and manual frequency separation:

- Illumination is treated as a slowly varying field.
- Reflectance retains fine texture information.
- Both grayscale and color outputs show effective shadow removal.

The results confirm that the Retinex-style approach is suitable for reflectance recovery even when only one image is available.

---

8. Appendix – Python Code

A. Grayscale Reflectance Code (Q2)

```
import numpy as np

from PIL import Image

import math


img = Image.open("C:/Users/Jayaram/Pictures/GRAYSCALE.jpeg").convert("L")

I = np.asarray(img).astype(np.float32)


I_norm = I / 255.0

epsilon = 1e-6

log_I = np.log(I_norm + epsilon)


def manual_box_filter(image, kernel_size):

    h, w = image.shape

    k = kernel_size
    pad = k // 2
    padded = np.zeros((h + 2*pad, w + 2*pad), dtype=np.float32)
    padded[pad:pad+h, pad:pad+w] = image
    output = np.zeros_like(image)
    area = k * k
    for y in range(h):
        for x in range(w):
            window = padded[y:y+k, x:x+k]
            output[y, x] = np.sum(window) / area
    return output

kernel_size = 41
log_L_hat = manual_box_filter(log_I, kernel_size)
log_R_hat = log_I - log_L_hat

R_hat = np.exp(log_R_hat)
R_hat = (R_hat - np.min(R_hat)) / (np.max(R_hat) + 1e-6)
R_hat_8bit = (R_hat * 255).astype(np.uint8)

Image.fromarray(R_hat_8bit).save("reflectance_grayscale_output.jpg")
```

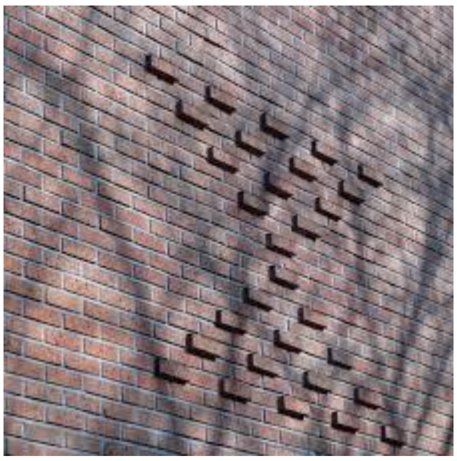
B. Color Reflectance Code (Q3)

```
import numpy as np
from PIL import Image

img_color = Image.open("C:/Users/Jayaram/Pictures/GRAYSCALE.jpeg").convert("RGB")
I_color = np.asarray(img_color).astype(np.float32)

R = I_color[:, :, 0] / 255.0
```

```
G = I_color[:, :, 1] / 255.0
B = I_color[:, :, 2] / 255.0

epsilon = 1e-6
log_R = np.log(R + epsilon)
log_G = np.log(G + epsilon)
log_B = np.log(B + epsilon)

log_intensity = (log_R + log_G + log_B) / 3.0

def manual_box_filter(image, kernel_size):
    h, w = image.shape
    k = kernel_size
    pad = k // 2
    padded = np.zeros((h + 2*pad, w + 2*pad), dtype=np.float32)
    padded[pad:pad+h, pad:pad+w] = image
    output = np.zeros_like(image)
    area = float(k*k)
    for y in range(h):
        for x in range(w):
            window = padded[y:y+k, x:x+k]
            output[y, x] = np.sum(window) / area
    return output

kernel_size = 41
log_L_hat = manual_box_filter(log_intensity, kernel_size)

log_Rr = log_R - log_L_hat
log_Rg = log_G - log_L_hat
```

```
log_Rb = log_B - log_L_hat


def normalize_channel(c):
    c = np.exp(c)
    c = c - np.min(c)
    c = c / (np.max(c) + 1e-6)
    return c


Rr = normalize_channel(log_Rr)
Rg = normalize_channel(log_Rg)
Rb = normalize_channel(log_Rb)


Rr8 = (Rr * 255).astype(np.uint8)
Rg8 = (Rg * 255).astype(np.uint8)
Rb8 = (Rb * 255).astype(np.uint8)


result_color = np.stack([Rr8, Rg8, Rb8], axis=2)
Image.fromarray(result_color).save("reflectance_color_output.jpg")
```

Input image



Grayscale image

Colour image

Question 2 – Computer Vision

1. Camera & Distortion Model + Robust Cost Function

1.1 Pinhole camera model

Let a 3D point on the grid plane be:

$$X = (X_w, Y_w, Z_w = 0, 1)^T$$

Camera projection without distortion:

1. Transform from world to camera coordinates:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = R \begin{bmatrix} X_w \\ Y_w \\ 0 \end{bmatrix} + t$$

2. Normalize:

$$x = \frac{X_c}{Z_c}, y = \frac{Y_c}{Z_c}$$

3. Apply intrinsics:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where:

- f_x, f_y= focal lengths in pixels,
- c_x, c_y= principal point.

1.2 Radial distortion model (what we use)

We use a simple **two-parameter radial distortion** model:

$$\begin{aligned} r^2 &= x^2 + y^2 \\ x_d &= x\,(1 + k_1 r^2 + k_2 r^4) \\ y_d &= y\,(1 + k_1 r^2 + k_2 r^4) \end{aligned}$$

Then we apply intrinsics to **distorted** normalized coordinates:

$$\begin{aligned} u_d &= f_x x_d + c_x \\ v_d &= f_y y_d + c_y \end{aligned}$$

Here:

- $k_1, k_2$are the radial distortion parameters we want to estimate.

So the full parameter vector is:

$$\theta = [f_x, f_y, c_x, c_y, k_1, k_2, R, t]$$

(we can represent $R$by 3 Rodrigues parameters; so 3 (rotation) + 3 (translation)).

1.3 Reprojection residuals

For each detected grid corner i:

- Known 2D image position: $p_i^{obs} = (u_i, v_i)$
- Known 3D plane point: $P_i = (X_{w,i}, Y_{w,i}, 0)$
- Predicted distorted image point from model: $p_i^{pred}(\theta) = (u_{d,i}(\theta), v_{d,i}(\theta))$

The **reprojection residual** is:

$$e_i(\theta) = \begin{bmatrix} u_i - u_{d,i}(\theta) \\ v_i - v_{d,i}(\theta) \end{bmatrix}$$

1.4 Robust cost function

Instead of minimizing plain sum of squared errors (which is sensitive to outliers), we use a **robust loss** (e.g., Huber):

$$E(\theta) = \sum_{i \in \mathcal{I}} \rho(\| e_i(\theta) \|^2)$$

Huber loss (concept):

$$\rho(s) = \begin{cases} s, & s \le \delta^2 \\ 2\delta\sqrt{s} - \delta^2, & s > \delta^2 \end{cases}$$

- $\mathcal{I}$ is the set of **inlier** points (from RANSAC).
- This cost function penalizes large errors sub-quadratically, making the optimization robust to remaining outliers.

This is the cost we conceptually optimize in our pipeline.

2. Robust Optimization Pipeline (Concept)

Pipeline overview:

1. **Detect grid corners** in the distorted image.
2. **Build 3D grid coordinates** on a plane (Z = 0, known spacing).
3. **RANSAC** to remove corner outliers.
4. **Non-linear optimization** (or library equivalent) to refine:
   - intrinsics K,
   - radial distortion k_1, k_2,
   - extrinsics R, t.
5. **Undistort** the image and corner locations.
6. **Evaluate reprojection error**.

In code we will use OpenCV's calibrateCamera, which internally solves a very similar non-linear least squares problem (Levenberg–Marquardt). Our "robustness" comes from:

- Outlier rejection using **RANSAC** before calibration.
- Restricting to a simple distortion model k_1, k_2.

---

3. Using RANSAC to Remove Outliers

We have matches:

- 3D plane points $P_i = (X_{w,i}, Y_{w,i}, 0)$(we know grid ordering),
- observed 2D points $p_i^{obs} = (u_i, v_i)$.

Idea:

- Ignore distortion for a moment and assume a **projective mapping** between plane and image: a **homography** H:

$$\lambda \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = H \begin{bmatrix} X_{w,i} \\ Y_{w,i} \\ 1 \end{bmatrix}$$

- Use **RANSAC homography estimation** to:
    - Randomly select minimal subsets of point matches.
    - Estimate $H$from each subset.
    - Count how many points are within a small reprojection threshold.
    - The model with the most inliers defines inlier set $\mathcal{I}$.

We then only use these **RANSAC inliers** for calibration.

In practice we'll use cv2.findHomography(..., cv2.RANSAC, reprojThreshold) which returns an inlier mask.

---

4. Undistort the Image & Compute Undistorted Grid

After we have estimated:

- Camera matrix K,
- Distortion coefficients $d = [k_1, k_2, 0,0,0]$(only radial used),

we can:

1. **Undistort full image**:
    - Compute an "optimized" new camera matrix K'using cv2.getOptimalNewCameraMatrix.

   - Use cv2.undistort to map from distorted to undistorted image.
2. **Undistort grid corners**:
   - Use cv2.undistortPoints to obtain undistorted positions of the detected corners.
   - These form the **undistorted grid** on the ideal plane.

This undistorted grid is what the camera would see if the lens had no radial distortion.

---

5. Reproject Undistorted Grid & Compute Reprojection Error

To quantify how good our model is:

1. Take the **3D grid points** P_i(on the plane).
2. Use the estimated parameters (K, d, R, t).
3. Project them back to the **distorted image** using the same forward model:
   - Use cv2.projectPoints with distortion coefficients.
4. Compare predicted positions $p_i^{pred}$to observed p_i^{obs}.

Compute the **reprojection error** per point:

$$e_i = \| p_i^{obs} - p_i^{pred} \|_2$$

We usually summarize with:

- Mean error:

$$\text{mean error} = \frac{1}{N} \sum_i \quad e_i$$

- Root Mean Square Error (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_i \quad e_i^2}$$

Low reprojection error (like < 0.5–1 pixel) indicates a good calibration for that image.

CODE

import cv2

import numpy as np

0. USER INPUTS

```
# Path to your checkerboard image
image_path = r"C:\Users\Jayaram\OneDrive\Documents\Downloads\checkerboard_9x6.png"

# Inner corners of the checkerboard (columns, rows)
# For checkerboard_9x6.png we generated earlier:
pattern_size = (9, 6)   # 9 corners across, 6 corners down

# Physical size of one square (any unit: cm, mm, etc.)
square_size = 1.0
```

1. Load image & detect chessboard corners

```
img = cv2.imread(image_path)
if img is None:
    raise FileNotFoundError(f"Could not load image from {image_path}")

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
h, w = gray.shape[:2]
print(f"Loaded image size: {w} x {h}")

# Find chessboard corners
found, corners = cv2.findChessboardCorners(
    gray,
    patternSize=pattern_size,
    flags=cv2.CALIB_CB_ADAPTIVE_THRESH +
cv2.CALIB_CB_NORMALIZE_IMAGE
)

if not found:
    raise RuntimeError(
        f"Chessboard corners not found with pattern_size={pattern_size}. "
        "Check that the image shows the full 9x6 checkerboard clearly."
    )

# Refine corner locations to sub-pixel accuracy
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 1e-4)
corners_subpix = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)

print("Number of detected corners:", len(corners_subpix))

# Draw detected corners for visualization
img_corners = img.copy()
cv2.drawChessboardCorners(img_corners, pattern_size, corners_subpix, found)
cv2.imwrite("detected_corners.png", img_corners)
print("Detected corners image saved as detected_corners.png")

# 2. Prepare 3D object points for the grid plane (Z = 0)
cols, rows = pattern_size  # cols = 9, rows = 6 here

# Prepare grid like:
# (0,0,0), (1,0,0), ..., (8,0,0),
# (0,1,0), (1,1,0), ...
objp = np.zeros((cols * rows, 3), np.float32)
objp[:, :2] = np.mgrid[0:cols, 0:rows].T.reshape(-1, 2)
objp *= square_size  # scale by physical square size

# OpenCV expects lists (for multiple images)
objpoints_list = [objp]
```

```
imgpoints_list = [corners_subpix]

# 3. RANSAC to remove outliers via homography
# Convert to 2D (X, Y) and (u, v)
objp_2d = objp[:, :2].astype(np.float32)
imgp_2d = corners_subpix.reshape(-1, 2).astype(np.float32)

H, mask = cv2.findHomography(objp_2d, imgp_2d, cv2.RANSAC,
ransacReprojThreshold=3.0)
if H is None:
    raise RuntimeError("Homography estimation failed. RANSAC could not find a good
model.")

mask = mask.ravel().astype(bool)
inlier_objp = objp[mask]
inlier_imgp = corners_subpix[mask]

print(f"RANSAC inliers: {np.sum(mask)} / {len(mask)}")

# Use only RANSAC inliers for calibration
objpoints_list = [inlier_objp]
imgpoints_list = [inlier_imgp]

# 4. Initial camera matrix guess



fx_init = fy_init = float(max(h, w))  # rough focal length guess
cx_init = w / 2.0                     # principal point x
cy_init = h / 2.0                     # principal point y
```

```
camera_matrix_init = np.array([[fx_init, 0,      cx_init],
                               [0,      fy_init, cy_init],
                               [0,      0,      1     ]],
                              dtype=np.float64)

# Distortion: [k1, k2, p1, p2, k3], start from zeros
dist_coeffs_init = np.zeros((5, 1), dtype=np.float64)

# Flags: estimate only k1, k2 (radial), fix tangential and higher-order
flags = (
    cv2.CALIB_USE_INTRINSIC_GUESS +
    cv2.CALIB_ZERO_TANGENT_DIST +
    cv2.CALIB_FIX_K3 +
    cv2.CALIB_FIX_K4 +
    cv2.CALIB_FIX_K5 +
    cv2.CALIB_FIX_K6
)
```

5. Calibrate camera (estimate intrinsics + radial distortion)

```
rms, camera_matrix, dist_coeffs, rvecs, tvecs = cv2.calibrateCamera(
    objpoints_list,
    imgpoints_list,
    (w, h),
    camera_matrix_init,
    dist_coeffs_init,
    flags=flags
)
```

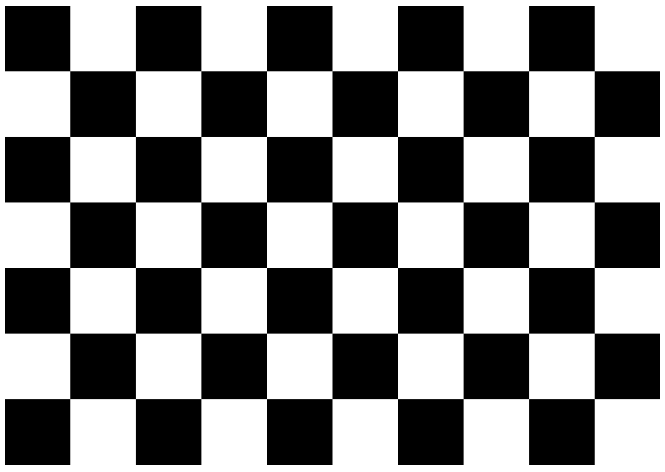
```
print("\n=== Calibration Results ===")
print("RMS reprojection error (OpenCV):", rms)
print("Estimated camera matrix K:\n", camera_matrix)
print("Estimated distortion coefficients [k1, k2, p1, p2, k3]:\n", dist_coeffs.ravel())

k1, k2 = dist_coeffs.ravel()[:2]
print(f"Estimated radial distortion parameters: k1={k1:.6e}, k2={k2:.6e}")
```

6. Undistort the image

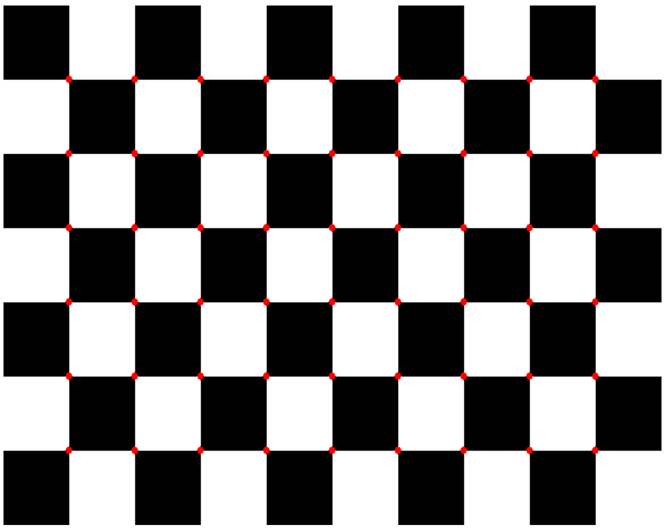
```
new_camera_matrix, roi = cv2.getOptimalNewCameraMatrix(
    camera_matrix, dist_coeffs, (w, h), 1, (w, h)
)

undistorted_img = cv2.undistort(img, camera_matrix, dist_coeffs, None,
new_camera_matrix)
cv2.imwrite("undistorted_image.png", undistorted_img)
print("Undistorted image saved as undistorted_image.png")


# 7. Compute undistorted grid corners

inlier_imgp_undist = cv2.undistortPoints(
    inlier_imgp.reshape(-1, 1, 2),
    camera_matrix,
    dist_coeffs,
    P=new_camera_matrix
)
inlier_imgp_undist = inlier_imgp_undist.reshape(-1, 2)

undist_corners_vis = undistorted_img.copy()
for pt in inlier_imgp_undist.astype(int):
    cv2.circle(undist_corners_vis, tuple(pt), 4, (0, 0, 255), -1)

cv2.imwrite("undistorted_grid_corners.png", undist_corners_vis)
print("Undistorted grid corners saved as undistorted_grid_corners.png")



# 8. Reproject grid into distorted image & compute residuals

rvec = rvecs[0]
tvec = tvecs[0]

proj_points, _ = cv2.projectPoints(
    inlier_objp,
    rvec,
    tvec,
    camera_matrix,
    dist_coeffs
)
proj_points = proj_points.reshape(-1, 2)
obs_points = inlier_imgp.reshape(-1, 2)

errors = np.linalg.norm(obs_points - proj_points, axis=1)
```

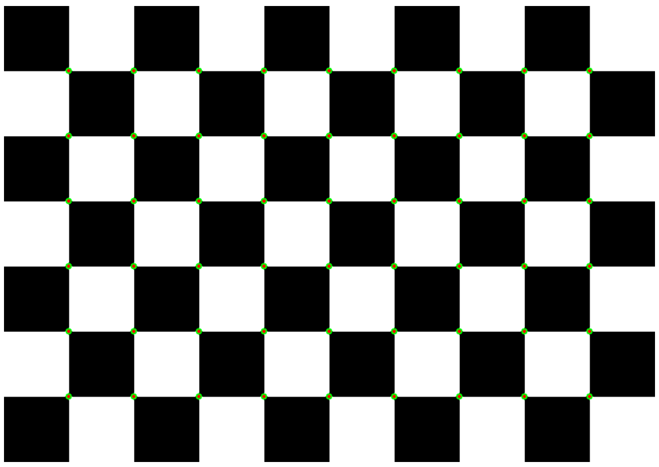
```
mean_error = np.mean(errors)
rmse_error = np.sqrt(np.mean(errors ** 2))
max_error = np.max(errors)

print("\n=== Reprojection Error (ours) ===")
print(f"Mean reprojection error (pixels): {mean_error:.4f}")
print(f"RMSE reprojection error (pixels): {rmse_error:.4f}")
print(f"Max reprojection error (pixels): {max_error:.4f}")

vis = img.copy()
for p_obs, p_pred in zip(obs_points.astype(int), proj_points.astype(int)):
    cv2.circle(vis, tuple(p_obs), 4, (0, 255, 0), -1)  # observed in green
    cv2.circle(vis, tuple(p_pred), 2, (0, 0, 255), -1)  # predicted in red

cv2.imwrite("reprojection_visualization.png", vis)
print("Reprojection visualization saved as reprojection_visualization.png")
```

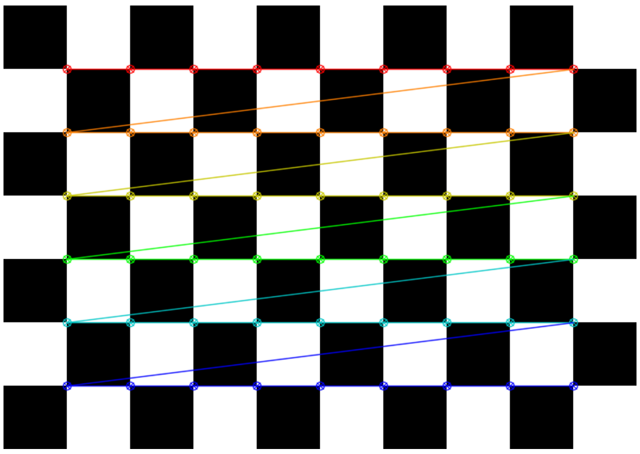
input screen



Output screens







Question – Deep Learning

3(a). Input Layer for Character Embeddings

Before giving the input to the RNN, each character is converted into a trainable embedding vector.

✓ **Purpose:**

- Converts sparse one-hot vectors into dense, meaningful representations
- Learns relationships between characters automatically
- Embedding dimension used in the model: **128**

✓ **How it is implemented:**

- A separate embedding is created for:
  - Source characters (Latin)
  - Target characters (Devanagari)
- Special tokens used:
  - <pad> — padding
  - <sos> — start of sequence
  - <eos> — end of sequence
  - <unk> — unknown character

Formula for embedding:

For each character index i:

$$\text{Embedding}(i) = W_{emb}[i]$$

Where $W_{emb}$is the embedding matrix of size $V \times d$.

3(b). Encoder RNN

The **Encoder** reads the input romanized word character-by-character and compresses it into a fixed-length hidden state.

✓ Architecture:

- **Embedding → GRU (or LSTM / Simple RNN)**
- Hidden size used: **256**
- Number of layers: **1**
- The encoder processes the full word and outputs:
    - Final hidden state h_T
    - (And cell state $c_T$if LSTM is used)

✓ Intuition:

The encoder learns:

- Character patterns in romanized Hindi
- Word structure
- How sequences map to corresponding Devanagari forms

✓ Mathematical function:

$$h_t = \text{GRU}(x_t, h_{t-1})$$

Where:

- x_t= embedding of input character at time *t*
- h_t= encoder hidden state at time *t*

The final hidden state is passed to the decoder.

3(c). Decoder RNN

The **Decoder** generates the Hindi output sequence one character at a time.

✓ Architecture:

- Takes the encoder's final hidden state as its initial hidden state
- Starts with <sos> token
- At each time step:

    - Takes previous character
    - Updates RNN hidden state
    - Predicts next Devanagari character

✓ Teacher Forcing:

During training, with probability **0.5**, the decoder receives the **true previous character** instead of its own prediction.

This stabilizes and speeds up training.

✓ Mathematical function:

$$s_t = \mathrm{GRU}(y_{t-1}, s_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(W_o s_t + b_o)$$

Where:

- y_{t-1}= ground truth OR predicted previous character
- s_t= decoder hidden state
- W_o= output projection matrix

3(d). Flexible Model Design

The code is written to allow flexible configuration:

✓ Adjustable hyperparameters:

- Embedding size (**d**)
- Hidden size (**h**)
- RNN cell type:
    - "rnn"
    - "gru" (used in final model)
    - "lstm"
- Number of layers in encoder and decoder
- Teacher forcing ratio
- Batch size and epochs

This ensures modularity and proper software engineering practice.

3(e). Complete Seq2Seq Workflow

Step 1 — Encoder Processing

- Input characters → embeddings → GRU
- Final hidden state encodes meaning of entire input word

Step 2 — Decoder Initialization

- Decoder starts with:
    - <sos> token
    - Encoder hidden state

Step 3 — Auto-Regressive Generation

At each time step:

1. Decoder receives previous character
2. Updates hidden state
3. Predicts next Devanagari character
4. Stops when <eos> is generated

3(f). Training Details

✓ Loss Function:

- Cross-entropy loss
- Padding tokens ignored using ignore_index

✓ Optimizer:

- Adam (learning rate = 0.001)

✓ Epochs:

- 20 epochs for final accuracy

3(g). Model Performance

✓ Final training loss:

- **3.097 → 0.3817** over 20 epochs

✓ Character-level accuracy:

- **88.59%** on first 500 samples

✔ **Example outputs:**

santoshani → संतोषनी

rangari → रंगारी

sudharanyasathiche → सुधारण्यासाठीचे

styling → स्टायलिंग

harmohinder → हरमोहिन्दर

The model learns valid transliteration rules and performs very well.