Code Implementation description

Setup & Installation (Colab Environment)

1. **Library Imports:**
   - Imports necessary Python libraries such as os, shutil, PIL (for image processing), torch (PyTorch), and torchvision's optical_flow models.
   - Includes optional Colab-specific tools like files for uploading/downloading.
2. **Change Working Directory:**
   - Ensures the script operates from the /content directory, the default workspace in Google Colab.
3. **Cloning GFPGAN Repository:**
   - Removes any existing GFPGAN directory to avoid conflicts.
   - Clones the official GFPGAN repository from GitHub, which provides tools for face restoration and enhancement.
4. **Installing Required Dependencies:**
   - Uses pip to install:
     - numpy==1.24.4 (compatible version).
     - Required deep learning libraries: basicsr, facexlib, realesrgan.
     - Dependencies listed in GFPGAN's requirements.txt.
   - Runs python setup.py develop to install GFPGAN in development mode (enables live code updates without reinstalling).
5. **Compatibility Fix for torchvision:**
   - Fixes a deprecated import statement in the installed basicsr package that is incompatible with recent versions of torchvision (e.g., Python 3.11).
   - Uses sed to update the import of rgb_to_grayscale to the correct path.
6. **Download Pre-trained GFPGAN Model:**
   - Creates a directory experiments/pretrained_models to store model weights.
   - Downloads the GFPGANv1.3.pth pre-trained model for inference use.

🖼️ GIF Upload and Frame Extraction

1. **Define Input Directory:**
    - gif_upload = 'inputs/upload': Specifies the directory where extracted frames will be saved.
    - shutil.rmtree(...): Removes any existing content in the directory to start clean.
    - os.makedirs(...): Recreates the directory to store new frames.
2. **Upload GIF File (Google Colab):**
    - files.upload(): Opens a file picker for the user to upload a GIF.
    - gif_file = list(uploaded.keys())[0]: Retrieves the uploaded filename from the returned dictionary.
3. **Extract Frames from the GIF:**
    - Image.open(gif_file): Opens the uploaded GIF using PIL.
    - ImageSequence.Iterator(img): Iterates through each frame of the animated GIF.
    - frame.convert("RGB").save(...): Converts each frame to RGB format and saves it as a PNG file in the gif_upload directory.
    - frame_paths.append(...): Tracks each saved frame's file path.
4. **Progress Output:**
    - Prints the number of frames extracted from the uploaded GIF, confirming successful preprocessing.

✨ Face Restoration with GFPGAN

1. **Clear Previous Results:**
   - !rm -rf results: Deletes any previous output directory to avoid mixing old and new results.
2. **Run Inference with GFPGAN:**
   - inference_gfpgan.py is the main inference script from the GFPGAN repository.
   - Parameters:
     - -i {gif_upload}: Input folder containing extracted frames from the GIF.
     - -o results: Output directory where enhanced images will be saved.
     - -v 1.3: Uses the GFPGAN version 1.3 model (GFPGANv1.3.pth).
     - -s 2: Sets the upscale factor to 2×.
     - --bg_upsampler realesrgan: Uses Real-ESRGAN for background upscaling to improve image quality beyond faces.
3. **Collect Restored Frame Paths:**
   - restored_dir = 'results/restored_imgs': Points to the directory where enhanced frames are saved.
   - restored_paths = sorted(...): Creates an ordered list of all restored frame file paths with .png extension.

🔄 Motion Estimation Using RAFT Optical Flow

1. **RAFt Model Setup:**
   - Uses raft_large model from torchvision.models.optical_flow, with pretrained weights (Raft_Large_Weights.DEFAULT).
   - Moves the model to GPU if available and sets it to evaluation mode (.eval()).
2. **Image Preprocessing:**
   - load_frame(...): Opens each restored image, resizes it to (384, 384) for consistent input shape, and converts it into a normalized PyTorch tensor with batch dimension [1, 3, H, W].
3. **Preparing Input Tensors:**
   - Loads and stores tensors for each restored frame from restored_paths using the load_frame() function.
4. **Debugging & Device Info:**
   - Prints whether GPU is used and displays PYTORCH_CUDA_ALLOC_CONF for potential debugging of memory issues.
5. **Optical Flow Computation:**
   - Iterates over each consecutive frame pair.
   - Applies the RAFT model's internal transform pipeline for proper preprocessing (e.g., normalization).
   - Inference is wrapped with torch.no_grad() to avoid gradient tracking and save memory.
   - Only the final refined flow result from RAFT is used.
   - Estimated flow tensors are moved to CPU to free up GPU memory.
6. **Output:**
   - Prints a success message indicating how many frame pairs had their motion vectors estimated.

🔁 Image Warping and Temporal Loss Calculation

This section aims to:

- **Warp** each frame toward the next frame using estimated optical flow.
- **Quantify temporal consistency** using Mean Squared Error (MSE) between warped and actual frames.

🔧 Implementation Details

1. **Setup:**
    - Initializes warped list to store warped frames and losses to store temporal losses.
    - Uses torch.nn.MSELoss() as the criterion for frame difference measurement.
2. **Per-Frame Warping Loop:**
   For each frame pair:
    - Moves both input (img) and target (target) frames to the computation device (GPU or CPU).
    - Moves optical flow (flow) to the same device (fixes mismatch errors).
    - Generates a 2D mesh grid (grid_x, grid_y) representing pixel coordinates.
    - Combines these into a grid and adds the flow to warp it: vgrid = grid + flow.
3. **Normalization for Grid Sampling:**
    - RAFT outputs flow in pixel units. F.grid_sample requires normalized coordinates in the range [-1, 1].
    - The code scales the vgrid accordingly and permutes its shape to match the input format [B, H, W, 2].
4. **Image Warping:**
    - F.grid_sample(...) warps the current frame (img) toward the next frame using the transformed grid.
    - This simulates how the current frame would look if moved using the predicted motion (optical flow).

5. **Temporal Loss Calculation:**
    - Calculates MSE between the warped image and the actual next frame (target).
    - Stores the warped image and its corresponding temporal loss.
6. **Output:**
    - Prints per-frame temporal loss values, which indicate how well the motion warping aligns with the next frame. Lower values imply better temporal coherence.

🖼️ GIF Reconstruction from Enhanced Frames

🔄 Purpose:

After performing face restoration and optional motion analysis (optical flow), this step reassembles the enhanced individual frames into a single animated GIF.

⚙️ Implementation Details

1. **Load Enhanced Frames:**
    - restored_images = [Image.open(p).convert("RGB") for p in restored_paths]:
        - Opens each restored PNG file.
        - Ensures all frames are in RGB format for consistency.
2. **Define Output Path:**
    - output = '/content/outputs/enhanced_output.gif': Specifies where the final GIF will be saved.
    - os.makedirs(...): Ensures the output directory exists.
3. **Save as Animated GIF:**
    - restored_images[0].save(...): Saves the first frame and appends the rest using:
        - save_all=True: Enables multiple frame saving.
        - append_images=...: Adds the remaining frames.

- duration=100: Sets frame duration (in milliseconds) — 10 frames per second.
- loop=0: Makes the animation loop infinitely.

4. **Output Confirmation:**
   - Prints the path where the enhanced GIF was saved.

📊 Objective Quality Assessment (PSNR & SSIM)

🧪 Purpose:

To compare the **restored (enhanced)** frames against the **original low-quality** frames and **quantify visual improvements** using widely accepted image quality metrics:

- **PSNR (Peak Signal-to-Noise Ratio)**: Measures the ratio between the maximum possible pixel value and the power of the noise (difference between the images). Higher is better.
- **SSIM (Structural Similarity Index)**: Evaluates perceived quality by considering luminance, contrast, and structural changes. Ranges from -1 to 1, where **1 indicates perfect similarity**.

⚙️ Implementation Details

1. **Frame Loading & Alignment:**
   - Loads each original and restored frame from:
     - inputs/upload: Original extracted frames.
     - results/restored_imgs: Enhanced frames after GFPGAN processing.
   - Resizes both to **384×384** pixels for consistent comparison.
2. **Metric Calculation:**
   - **PSNR** is calculated using skimage.metrics.peak_signal_noise_ratio.
   - **SSIM** is calculated using skimage.metrics.structural_similarity with channel_axis=2 for RGB images.
   - Metrics are computed for each frame pair and stored in lists.

3. **Averaged Output:**
   - The script prints the **mean PSNR** and **mean SSIM** across all frames:

python

CopyEdit

✅ PSNR (avg): 32.54 dB

✅ SSIM (avg): 0.9124

✨ Perceptual Quality & Temporal Consistency Evaluation

To assess the quality of the restored video frames, we used two important evaluation methods:

✅ 1. Temporal Loss (Smoothness Across Frames)

- **Purpose:** Measures how smoothly frames transition over time in the restored video.
- **Why it matters:** Flickering or inconsistent frames can reduce visual quality. A lower temporal loss means the restoration is temporally stable and pleasant to watch.
- **Output:** An average score showing how consistent the restored frames are. Lower is better.

✅ 2. LPIPS – Perceptual Image Quality

- **Purpose:** LPIPS (Learned Perceptual Image Patch Similarity) evaluates how *visually similar* the restored images are compared to the original ones.
- **Why it matters:** Traditional metrics like PSNR or SSIM don't always reflect human visual perception. LPIPS uses deep neural networks to model how people actually *see* differences.
- **Process:** Each restored frame is compared to its original version. The LPIPS model outputs a perceptual difference score for each pair.
- **Output:** An average LPIPS score across all frames. Lower values indicate that the restored images are closer to the original in terms of perceptual quality.