app.py

Import required libraries

 streamlit: For building the web interface.

 TrOCRProcessor: Preprocesses images for the TrOCR model.

 VisionEncoderDecoderModel: The TrOCR model (encoder-decoder for vision-text).

 PIL.Image: To handle image reading and manipulation.

 torch: For tensor operations and inference.

 re: For regex-based text cleanup.

 preprocess\_image, segment\_lines: Custom functions defined in your utils.py for preprocessing and line segmentation.

Load TrOCR model and processor (cached)

@st.cache\_resource is for **stateful objects** like:

* PyTorch/TensorFlow models
* Hugging Face models
* Database connections

**def load\_model():**

* This defines a Python function named load\_model.
* Purpose: to load the **TrOCR model** and **processor** required for handwritten text recognition.

**processor = TrOCRProcessor.from\_pretrained("microsoft/trocr-base-handwritten")**

* Loads a **pre-trained processor** from Hugging Face's model hub.
* Loads the **preprocessing and postprocessing logic**
* TrOCRProcessor handles:
  + Image preprocessing (e.g., resizing, normalization).
  + Tokenization for decoding model output.
* "microsoft/trocr-base-handwritten" is a model specifically fine-tuned on handwritten text.
* This processor ensures the input image is correctly formatted for the model.

**model = VisionEncoderDecoderModel.from\_pretrained("microsoft/trocr-base-handwritten")**

* Loads the **pre-trained model weights** for TrOCR from the same Hugging Face model.
* VisionEncoderDecoderModel is a transformer architecture with:
  + A **vision encoder** (like a CNN or ViT) for image understanding.
  + A **text decoder** (like GPT2 or BERT) for generating textual output.
* It connects image understanding and text generation in one end-to-end model.

**model.eval()**

* Sets the model to **evaluation mode**, not training mode.
* Turns off things like:
  + Dropout
  + Batch normalization updates
* This ensures consistent and deterministic behavior during inference.

**return processor, model**

* Returns the loaded processor and model as a tuple.
* These will be used later in your OCR pipeline for:
  + Processing images (processor)
  + Generating text predictions (model)

14–16: Set Streamlit page and UI headers

st.set\_page\_config(page\_title="Handwritten OCR", layout="centered")

**Purpose:**

This sets the configuration for your Streamlit app **before any UI is rendered**.

**🧠 Parameters:**

* page\_title="Handwritten OCR"  
  → Sets the title shown on the **browser tab** .

st.title("✍️ Handwritten Text to OCR using Hugging Face (TrOCR)")

Displays a **big bold title** at the top of your app UI.

st.write("Upload a handwritten image and extract text using Transformer-based OCR")

Displays a **text description or paragraph** under the title.

**🧠 Notes:**

* st.write() is flexible — it can print strings, numbers, dataframes, plots, etc.

uploaded\_file = st.file\_uploader("📤 Upload a handwritten image", type=["jpg", "jpeg", "png"])

20–23: Display uploaded image

**✅ Line 1: if uploaded\_file:**

* Checks if a file was **uploaded by the user** via st.file\_uploader(...)

**Line 2: image = Image.open(uploaded\_file).convert(“RGB”) red,green, blue**

* Loads the uploaded file as an image using the **Pillow (PIL)** library.
* .convert("RGB") ensures the image is in **RGB color mode**, even if the uploaded image was grayscale

**Line 3: st.subheader("🖼 Uploaded Image")**

* Displays a **subheading** in the app:  
  ➤ “🖼 Uploaded Image”

**Line 4: st.image(image, caption="Handwritten Input", use\_container\_width=True)**

Displays the uploaded image inside the app.

* caption="Handwritten Input" → Shows a label below the image.
* use\_container\_width=True → Scales the image to fit the width of the display area

**25–26: Load model once file is uploaded**

processor, model = load\_model()

* Retrieves the cached model and processor.

**28–30: Segment image into lines**

st.subheader("🔍 Line Segmentation")

lines = segment\_lines(image)

* Calls segment\_lines() to detect and extract lines from the image.
* Each line will be OCR-processed separately.

**32–33: Handle no lines detected**

if not lines:

st.warning("No text lines detected.")

* Shows a warning if no lines were found by the segment\_lines() function.

**34–36: Process each detected line**

else:

st.success(f"{len(lines)} line(s) detected")

st.subheader("📝 Predicted Text")

final\_output = ""

* Displays the number of lines detected.
* Prepares to show OCR output line-by-line.

37–46: Iterate over each line image and predict text

for idx, line\_img in enumerate(lines):

 Loops over all **segmented line images**.

 lines is a list of image crops (each representing one handwritten line).

 enumerate() gives both:

* idx → index (0, 1, 2, ...)
* line\_img → the image for each line

preprocessed = preprocess\_image(line\_img).convert("RGB")

 Calls your custom preprocess\_image() function to:

* Resize
* Pad
* Normalize (if needed)

 .convert("RGB") ensures the image is in standard RGB format.

 The result is a cleaned image, ready for model input.

pixel\_values = processor(images=preprocessed, return\_tensors="pt").pixel\_values

 Uses the **TrOCRProcessor** to:

* Resize and normalize the image (if needed)
* Convert it into a PyTorch tensor

 return\_tensors="pt" specifies PyTorch (pt) output

 pixel\_values will be of shape [1, 3, H, W] for model input

with torch.no\_grad():

Temporarily disables gradient tracking to:

* **Reduce memory usage**
* **Speed up** inference

generated\_ids = model.generate(pixel\_values)

 Feeds the preprocessed image tensor into the model.

 model.generate() uses **beam search or greedy decoding** to output the predicted text.

 generated\_ids are token IDs (integers representing characters or subwords).

raw\_text = processor.batch\_decode(generated\_ids, skip\_special\_tokens=True)[0]

Converts the predicted generated\_ids back into human-readable text.

**Final Purpose of This Block:**

* Takes a **single line image**, sends it through:
  + Preprocessing
  + Model inference
  + Decoding
* And returns the **recognized text** for that line.

**48–49: Clean up predicted text**

cleaned\_text = raw\_text.strip()

cleaned\_text = re.sub(r"[\d\W\_]+$", "", cleaned\_text)

* Removes extra whitespace and **trailing** digits/punctuation using regex.

**51–52: Display and accumulate results**

st.markdown(f"\*\*Line {idx + 1}:\*\* {cleaned\_text}")

final\_output += cleaned\_text + "\n"

* Displays each cleaned line as **Markdown** in Streamlit.
* Adds each line to a final output string (useful for later export or display).

This app performs:

* Uploading and displaying images
* Segmenting text into lines
* OCR using a TrOCR Transformer model
* Cleaning and displaying results line by line

**Workflow Summary:**

1. **🖼 Upload a Handwritten Image:**
   * User uploads an image (JPG/PNG) containing handwritten text.
2. **📐 Line Segmentation:**
   * The image is split into individual **text lines** using your custom segment\_lines() function (from utils.py).
3. **🧠 Load TrOCR Model:**
   * A Hugging Face **TrOCR model** (microsoft/trocr-base-handwritten) is loaded with:
     + TrOCRProcessor for preprocessing and decoding.
     + VisionEncoderDecoderModel for inference.
4. **🧼 Preprocess Each Line:**
   * Each segmented line is resized, padded, and converted into a format expected by the model.
5. **🔮 Predict Text:**
   * The TrOCR model **generates text token IDs** from the line image.
   * These are then **decoded into readable text** using the processor.
6. **🧹 Clean Output:**
   * You clean the predicted text using regex to remove trailing garbage like digits and special characters.
7. **📋 Display Results:**
   * Each line is displayed below the original image as **Line 1, Line 2, etc.**

**Conclusion:**

This app is a robust, Transformer-powered handwritten OCR system that combines **Hugging Face’s TrOCR model** with a clean **Streamlit interface** for real-time image-to-text conversion. It's modular, scalable, and perfect for expanding into projects like form readers, digitizers, or even exam sheet evaluation tools.

Utils.py

1. Import Required Libraries

 **PIL.Image**: For image reading, conversion, cropping, etc.

 **numpy**: Converts PIL image into an array so OpenCV can work with it.

 **cv2 (OpenCV)**: For image processing operations like thresholding, dilation, contour detection.

5-11 preprocess the input image

def preprocess\_image(img: Image.Image, resize\_height=384):

Defines a function to preprocess the input image.

Input:

img: a PIL image (usually a line of handwritten text).

resize\_height: Target height (default = 384) for resizing.

img = img.convert("L")

Converts the image to grayscale ("L" mode), reducing it to 1 channel.

Makes processing faster and simpler (no color information needed).

w, h = img.size

Gets the width and height of the image.

aspect\_ratio = w / h

Calculates the aspect ratio to preserve proportions during resizing.

new\_w = int(aspect\_ratio \* resize\_height)

Computes the new width based on the fixed resize\_height and aspect ratio.

This ensures that the image is resized proportionally.

img = img.resize((new\_w, resize\_height))

Resizes the image to the target size: (new width, fixed height).

return img

Returns the resized grayscale image.

13-16 **line segmentation**

def segment\_lines(pil\_image):

Defines a function that performs line segmentation on the given PIL image.

Returns a list of cropped line images.

**1.Convert to Grayscale**

gray = pil\_image.convert("L")

* Converts the input image to grayscale

**2: Convert PIL to NumPy Array**

img = np.array(gray)

* Converts the grayscale PIL image to a NumPy array so it can be processed with OpenCV.

**3:Binarize (Thresholding)**

\_, thresh = cv2.threshold(img, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

* Applies binary thresholding using Otsu's method.
* cv2.THRESH\_BINARY\_INV: White becomes black and vice versa (text becomes white on black).
* Result: A binary image with text as **white** on **black** background.

**4: Dilation (connect text)**

kernel = cv2.getStructuringElement(cv2.MORPH\_RECT, (img.shape[1] // 2, 5))

dilated = cv2.dilate(thresh, kernel, iterations=2)

* **Creates a rectangular kernel**: very wide (half the image width) and short (5 pixels).
* **Dilation** joins together connected components horizontally (lines of text).
* iterations=2: Strengthens the dilation effect.

**5: Find Contours (Lines)**

contours, \_ = cv2.findContours(dilated, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

* Finds **external contours** (connected white blobs) in the dilated image.
* Each contour represents a **line of text**.

**Sort Top to Bottom**

line\_images = []

for cnt in sorted(contours, key=lambda x: cv2.boundingRect(x)[1]):

* Sorts the contours by their **Y-position** (y value of bounding box).
* This ensures text lines appear in **top-to-bottom** order.

**Filter Small Boxes and Crop**

x, y, w, h = cv2.boundingRect(cnt)

if h > 15: # Filter small noise

line\_crop = pil\_image.crop((x, y, x + w, y + h))

line\_images.append(line\_crop)

* cv2.boundingRect(cnt) gets the bounding box for each contour (line).
* If height h > 15, it is treated as a valid line (filters out small noise blobs).
* Crops the original PIL image using crop() and adds it to line\_images

**8: Return Line Crops**

return line\_images

* Returns the list of cropped line images to be passed into the OCR model one by one.

| **Block** | **Purpose** |
| --- | --- |
| preprocess\_image | Resize line image to fixed height while keeping aspect ratio |
| Convert to grayscale | Simplifies image, reduces complexity |
| Threshold + Invert | Makes text white on black for easier contour detection |
| Dilation | Merges characters into line-like blobs |
| Contour detection | Finds text lines as blobs |
| Crop & return lines | Crops each detected line from the original image |

**Conclusion for Your Preprocessing and Line Segmentation Code**

Your preprocess\_image() and segment\_lines() functions form the **foundation of your OCR pipeline**, ensuring that input images are:

1. **Standardized in size and format**
2. **Cleanly separated into individual lines of text**

**🔧 preprocess\_image():**

* Converts images to **grayscale** for simplicity.
* Resizes images while **maintaining the original aspect ratio**.
* Ensures that every input to the OCR model is **uniform**, improving prediction accuracy.

**🧠 segment\_lines():**

* Uses **OpenCV techniques** like thresholding, dilation, and contour detection to:
  + Separate multi-line handwritten input into **individual line images**.
  + **Filter out noise** and return only meaningful lines.
  + Maintain **reading order** (top to bottom) using sorting.

**🚀 Overall:**

These functions prepare your handwritten images for **accurate OCR recognition** by TrOCR. The result is a clean, line-by-line prediction pipeline that works even on noisy, real-world handwriting.

They are:

* Efficient
* Modular
* Easy to integrate into any OCR system

**Overall Conclusion for Your Handwritten OCR Project using Hugging Face TrOCR and Streamlit**

Your project successfully delivers a **complete end-to-end handwritten OCR solution** using cutting-edge **Transformer-based deep learning** and a clean **Streamlit web interface**. It combines powerful backend AI models with an accessible frontend for real-world usage.

**🔍 What Your Project Does:**

1. **🖼 Image Upload & Visualization:**
   * Users can upload handwritten documents/images in JPG/PNG format.
2. **📐 Line Segmentation:**
   * Images are automatically segmented into individual **lines of text** using OpenCV.
   * Noise is filtered, and only valid text lines are processed.
3. **🧠 Text Recognition (OCR):**
   * Uses **Hugging Face’s TrOCR model** (microsoft/trocr-base-handwritten), a VisionEncoderDecoder Transformer model fine-tuned for handwriting recognition.
   * Each segmented line is processed and converted into accurate digital text.
4. **🖥 Streamlit UI:**
   * Intuitive web interface where users can:
     + Upload an image
     + View detected lines
     + See extracted text for each line
   * Caching and performance optimization included.

**🔬 Key Technologies Used:**

| **Component** | **Tool / Library** |
| --- | --- |
| OCR Model | Hugging Face TrOCR |
| Image Processing | OpenCV + NumPy |
| Web Interface | Streamlit |
| Image Handling | Pillow (PIL) |
| Backend Framework | PyTorch (via Hugging Face Transformers) |

**🧠 Strengths of the Project:**

* ✅ Uses **state-of-the-art Transformer OCR model**
* ✅ Supports **multi-line handwritten documents**
* ✅ Highly **modular and extensible** (easy to add word/paragraph detection)
* ✅ **Streamlit interface** makes it user-friendly and deployable
* ✅ Performs well on real-world handwritten inputs

**📌 Conclusion:**

Your project is a powerful and practical handwritten OCR tool that converts handwritten text into digital format using AI. It’s ideal for digitizing notes, exam papers, forms, and archival documents, and lays a solid foundation for future enhancements like word segmentation, real-time capture, multilingual support, or even text-to-speech.