```
import numpy as np
import os
from PIL import Image
import matplotlib.pyplot as plt

import torch
import torchvision
from torchvision import transforms as T
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor

import cv2
import os
import numpy as np
import matplotlib.pyplot as plt

def process_and_save_images(input_dir, output_dir):
    # Create the output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    for img_file in os.listdir(input_dir):
        if img_file.endswith(('.jpg', '.jpeg', '.png', '.bmp',
'.tif')):
            img_path = os.path.join(input_dir, img_file)
            base_name = os.path.basename(img_path)

            # Load the image
            image = cv2.imread(img_path)
            if image is None:
                print(f"Failed to read image: {img_path}")
                continue

            # Convert the image to grayscale
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

            # Apply Gaussian Blur to reduce noise
            blurred = cv2.GaussianBlur(gray, (5, 5), 0)

            # Apply thresholding to create a binary image
            _, binary = cv2.threshold(blurred, 40, 255,
cv2.THRESH_BINARY_INV)

            # Perform morphological processing to remove small noise
and fill the NT area
            kernel = np.ones((5, 5), np.uint8)
            morph = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)
            morph = cv2.morphologyEx(morph, cv2.MORPH_OPEN, kernel)

            # Create a figure to display the results
            #fig, axs = plt.subplots(1, 3, figsize=(12, 6))
```

```
            #axs[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
            #axs[0].set_title('Original Image')
            #axs[0].axis('off')

            #axs[1].imshow(binary, cmap='gray')
            #axs[1].set_title('Binary Image')
            #axs[1].axis('off')

            #axs[2].imshow(morph, cmap='gray')
            #axs[2].set_title('Morphologically Processed Image')
            #axs[2].axis('off')

            # Save the figure
            #output_path = os.path.join(output_dir,
f'processed_{img_file}.png')
            #fig.savefig(output_path)
            #plt.close(fig)
            cv2.imwrite(os.path.join(output_dir, f'{base_name}'),
binary)
            #cv2.imwrite(os.path.join(output_dir,
f'mask_{base_name}'), binary)

            print(f"Processed and saved: {output_dir}")

# Paths
input_dir = '/kaggle/input/cropped/cropped'  # Update with your input
folder path
output_dir = '/kaggle/working/masks'  # Update with your output folder
path

# Process and save images
process_and_save_images(input_dir, output_dir)

print(f"Processed images saved to {output_dir}")

import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
Dropout, concatenate, Conv2DTranspose
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.metrics import MeanIoU
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.preprocessing.image import load_img,
img_to_array

# Define a function to load and preprocess images
```

```
def load_and_preprocess_image(image_path, mask_path, target_size=(256,
256)):
    # Load and resize image
    image = load_img(image_path, target_size=target_size)
    image = img_to_array(image) / 255.0  # Normalize to [0, 1]

    # Load and resize mask
    mask = load_img(mask_path, target_size=target_size,
color_mode='grayscale')
    mask = img_to_array(mask) / 255.0  # Normalize to [0, 1]

    return image, mask

# Example dataset loading and preprocessing
def load_dataset(image_paths, mask_paths, target_size=(256, 256)):
    images = []
    masks = []
    for i in range(len(image_paths)):
        image_path = image_paths[i]
        mask_path = mask_paths[i]

        # Load and preprocess image and mask
        image, mask = load_and_preprocess_image(image_path, mask_path,
target_size)

        images.append(image)
        masks.append(mask)

    return np.array(images), np.array(masks)

# Example paths (replace with your actual paths)
image_folder = '/kaggle/input/cropped/cropped'
mask_folder = '/kaggle/working/masks/'

image_paths = [os.path.join(image_folder, f) for f in
os.listdir(image_folder) if os.path.isfile(os.path.join(image_folder,
f))]
mask_paths = [os.path.join(mask_folder, f) for f in
os.listdir(mask_folder) if os.path.isfile(os.path.join(mask_folder,
f))]

# Load and preprocess the dataset
X_train, y_train = load_dataset(image_paths, mask_paths,
target_size=(256, 256))

# Define U-Net architecture
# Define U-Net architecture
def unet_model(input_shape=(256, 256, 3)):
    inputs = Input(input_shape)
```

```python
    # Encoder
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, 3, activation='relu', padding='same')(pool1)
    conv2 = Conv2D(128, 3, activation='relu', padding='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(256, 3, activation='relu', padding='same')(pool2)
    conv3 = Conv2D(256, 3, activation='relu', padding='same')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Conv2D(512, 3, activation='relu', padding='same')(pool3)
    conv4 = Conv2D(512, 3, activation='relu', padding='same')(conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(1024, 3, activation='relu', padding='same')(pool4)
    conv5 = Conv2D(1024, 3, activation='relu', padding='same')(conv5)
    drop5 = Dropout(0.5)(conv5)

    # Decoder
    up6 = Conv2DTranspose(512, 2, strides=(2, 2), padding='same')
(drop5)
    merge6 = concatenate([drop4, up6], axis=3)
    conv6 = Conv2D(512, 3, activation='relu', padding='same')(merge6)
    conv6 = Conv2D(512, 3, activation='relu', padding='same')(conv6)

    up7 = Conv2DTranspose(256, 2, strides=(2, 2), padding='same')
(conv6)
    merge7 = concatenate([conv3, up7], axis=3)
    conv7 = Conv2D(256, 3, activation='relu', padding='same')(merge7)
    conv7 = Conv2D(256, 3, activation='relu', padding='same')(conv7)

    up8 = Conv2DTranspose(128, 2, strides=(2, 2), padding='same')
(conv7)
    merge8 = concatenate([conv2, up8], axis=3)
    conv8 = Conv2D(128, 3, activation='relu', padding='same')(merge8)
    conv8 = Conv2D(128, 3, activation='relu', padding='same')(conv8)

    up9 = Conv2DTranspose(64, 2, strides=(2, 2), padding='same')
(conv8)
    merge9 = concatenate([conv1, up9], axis=3)
    conv9 = Conv2D(64, 3, activation='relu', padding='same')(merge9)
    conv9 = Conv2D(64, 3, activation='relu', padding='same')(conv9)

    outputs = Conv2D(1, 1, activation='sigmoid')(conv9)  # Output mask
```

```
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer=Adam(learning_rate=1e-4),
loss=BinaryCrossentropy(), metrics=[MeanIoU(num_classes=2)])

    return model


# Initialize and compile the model
model = unet_model(input_shape=(256, 256, 3))

# Define ModelCheckpoint callback
checkpoint = ModelCheckpoint('unet_model.keras', monitor='val_loss',
verbose=1, save_best_only=True, mode='min')

# Train the model
model.fit(X_train, y_train, batch_size=16, epochs=50,
validation_split=0.1, callbacks=[checkpoint])

# Once trained, you can use the model for prediction
# Example prediction
# pred_mask = model.predict(new_cropped_NT_image)

# Save the model if needed
model.save('final_unet_model.h5')

import cv2 as cv



import cv2
import numpy as np

# Load and resize the input image
img = cv2.imread('/kaggle/input/cropped/cropped/109.png')
img_resized = cv2.resize(img, (256, 256))

# Preprocess the image for prediction
input_image = img_resized / 255.0  # Normalize to [0, 1]
input_image = np.expand_dims(input_image, axis=0)  # Add batch
dimension

# Predict the mask
pred_mask = model.predict(input_image)

# Assuming you want to visualize the prediction or use it further
# pred_mask contains the predicted mask, process further as needed


# Plotting the results
plt.figure(figsize=(12, 6))
```

```python
# Original Image
plt.subplot(1, 3, 1)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

# Resized Image
plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(img_resized, cv2.COLOR_BGR2RGB))
plt.title('Resized Image (256x256)')
plt.axis('off')

# Predicted Mask
plt.subplot(1, 3, 3)
plt.imshow(pred_mask[0, :, :, 0], cmap='gray')
plt.title('Predicted Mask')
plt.axis('off')

plt.tight_layout()
plt.show()


# Resize the predicted mask to match the original image size
pred_mask_resized = cv2.resize(pred_mask[0], (img.shape[1],
img.shape[0]))

# Threshold the mask (if necessary)
threshold = 0.5  # Adjust as needed
pred_mask_resized = (pred_mask_resized > threshold).astype(np.uint8) *
255  # Convert to binary mask

# Overlay the mask on the original image
masked_img = cv2.bitwise_and(img, img, mask=pred_mask_resized)

# Plotting the results
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 3, 1)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

# Resized Image
plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(img_resized, cv2.COLOR_BGR2RGB))
plt.title('Resized Image (256x256)')
plt.axis('off')

# Masked Image
```

```python
plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(masked_img, cv2.COLOR_BGR2RGB))
plt.title('Masked Image')
plt.axis('off')

plt.tight_layout()
plt.show()

im = cv.imread('/kaggle/working/masks/109.png')

plt.imshow(im)

import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image_path = '/kaggle/working/masks/252.png'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Check if the image is loaded successfully
if image is None:
    raise ValueError(f"Image not found or could not be loaded at path:
{image_path}")

# Create a mask for the white region (ROI)
_, mask = cv2.threshold(image, 254, 255, cv2.THRESH_BINARY)  #
Assuming white is 255

# Calculate the number of white pixels in each column
column_pixel_counts = np.sum(mask == 255, axis=0)

# Plot the number of white pixels against the x-coordinate
plt.figure(figsize=(10, 6))
plt.plot(range(len(column_pixel_counts)), column_pixel_counts,
label='Pixel Count')
plt.xlabel('X-coordinate')
plt.ylabel('Number of White Pixels')
plt.title('Number of White Pixels in Each Column')
plt.legend()
plt.grid(True)
plt.show()

print("Plot generated successfully.")
```