```
import os
from IPython.display import Image, display
from keras.utils import load_img
from PIL import ImageOps
import keras
import numpy as np
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io
import random

# Directories
input_dir = "images/"
target_dir = "annotations/trimaps/"
img_size = (160, 160)
num_classes = 3
batch_size = 32

# Collect input and target file paths
input_img_paths = sorted(
    [
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ]
)
target_img_paths = sorted(
    [
        os.path.join(target_dir, fname)
        for fname in os.listdir(target_dir)
        if fname.endswith(".png") and not fname.startswith(".")
    ]
)

print("Number of samples:", len(input_img_paths))

# Display some input-target path pairs
for input_path, target_path in zip(input_img_paths[:10],
target_img_paths[:10]):
    print(input_path, "|", target_path)

# Display an example input image and its corresponding target
display(Image(filename=input_img_paths[9]))
img = ImageOps.autocontrast(load_img(target_img_paths[9]))
display(img)

# Dataset preparation function
def get_dataset(batch_size, img_size, input_img_paths,
target_img_paths, max_dataset_len=None):
    """
    Returns a TensorFlow Dataset for image segmentation.
    """
```

dataset.batch(batch_size):
Batches the dataset into groups of batch_size. This is important for model training, where each batch will be processed by the model during training.

dataset = dataset.map(load_img_masks, num_parallel_calls=tf_data.AUTOTUNE): Applies the load_img_masks function to each pair of input and target image paths in the dataset. This will load and preprocess the images and masks. The num_parallel_calls=tf_data.AUTOTUNE optimizes the number of parallel calls for faster processing.

```
    def load_img_masks(input_img_path, target_img_path):
        # Load and preprocess input image
        input_img = tf_io.read_file(input_img_path)
        input_img = tf_io.decode_png(input_img, channels=3)
        input_img = tf_image.resize(input_img, img_size)
        input_img = tf_image.convert_image_dtype(input_img,
"float32")

        # Load and preprocess target mask
        target_img = tf_io.read_file(target_img_path)
        target_img = tf_io.decode_png(target_img, channels=1)
        target_img = tf_image.resize(target_img, img_size,
method="nearest")
        target_img = tf_image.convert_image_dtype(target_img,
"uint8")

        # Adjust target mask to start from 0
        target_img -= 1

        return input_img, target_img

    # Limit dataset size for faster debugging
    if max_dataset_len:
        input_img_paths = input_img_paths[:max_dataset_len]
        target_img_paths = target_img_paths[:max_dataset_len]

    # Create TensorFlow Dataset
    dataset = tf_data.Dataset.from_tensor_slices((input_img_paths,
target_img_paths))
    dataset = dataset.map(load_img_masks,
num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

# Shuffle and split the dataset into training and validation sets
val_samples = 1000
random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_img_paths)

train_input_img_paths = input_img_paths[:-val_samples]
train_target_img_paths = target_img_paths[:-val_samples]
val_input_img_paths = input_img_paths[-val_samples:]
val_target_img_paths = target_img_paths[-val_samples:]

# Create training and validation datasets
train_dataset = get_dataset(
    batch_size, img_size, train_input_img_paths,
train_target_img_paths, max_dataset_len=1000
)
valid_dataset = get_dataset(batch_size, img_size,
val_input_img_paths, val_target_img_paths)
```

dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, target_img_paths)): Converts the lists of image paths and target paths into a TensorFlow Dataset. Each item in the dataset will contain a pair of image path and mask path.

COCO Dataset Processing Code

list1 = [1, 2, 3]
tuple1 = (4, 5)

list1.extend(tuple1)
print(list1)
Output: [1, 2, 3, 4, 5]

```
import os
import json
import numpy as np
from pycocotools.coco import COCO
from pycocotools.mask import decode
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io

# Paths
input_dir = "images/"  # Directory containing image files
annotation_dir = "annotations/json/"  # Directory containing multiple
annotation JSON files

img_size = (160, 160)
batch_size = 32

# Load COCO Annotations from Multiple Files
def load_coco_annotations(annotation_dir):
    """Load annotations from multiple COCO JSON files."""
    coco_objects = []
    for filename in os.listdir(annotation_dir):
        if filename.endswith(".json"):
            annotation_path = os.path.join(annotation_dir, filename)
            with open(annotation_path, 'r') as f:
                _ = json.load(f)  # Load and validate JSON
            coco = COCO(annotation_path)
            coco_objects.append(coco)
    return coco_objects
```

os.path.join(input_dir, fname)
for fname in os.listdir(input_dir)
if fname.endswith(".jpg")

```
# Extract image IDs and their paths from all annotation files
def get_image_paths_and_ids(coco_objects, input_dir):
    """Extract image paths and their IDs from multiple COCO objects."""
    image_ids = []
    input_img_paths = []
    for coco in coco_objects:
        ids = coco.getImgIds()
        image_ids.extend(ids)
        paths = [
            os.path.join(input_dir,
coco.loadImgs(img_id)[0]['file_name'])
            for img_id in ids
        ]
        input_img_paths.extend(paths)
    return image_ids, input_img_paths
```

Extract all image IDs using coco.getImgIds() and add them to image_ids using .extend(). This ensures the list contains all image IDs from all COCO objects.

Retrieve file names of the images corresponding to the IDs using coco.loadImgs(img_id)[0]['file_name'] and construct their full paths by combining them with input_dir. These paths are added to input_img_paths.

- ids থেকে file নেওয়া হবে। coco থেকে, input dir এ images join করা। এ ফাইলগুলো file input dir এ join করে list করা হবে।

```
# Preprocessing Functions
def preprocess_image(img_path):
    """Loads and resizes an image."""
    input_img = tf_io.read_file(img_path)
    input_img = tf_io.decode_jpeg(input_img, channels=3)
    input_img = tf_image.resize(input_img, img_size)
    input_img = tf_image.convert_image_dtype(input_img, "float32")
    return input_img

def generate_segmentation_mask(coco, image_id, img_height, img_width):
    """Generates a binary mask from COCO segmentation annotations."""
    ann_ids = coco.getAnnIds(imgIds=image_id, iscrowd=False)
    annotations = coco.loadAnns(ann_ids)
    mask = np.zeros((img_height, img_width), dtype=np.uint8)
    for ann in annotations:
        if 'segmentation' in ann:
            rle = coco.annToRLE(ann)
            mask += decode(rle)  # Decode RLE to a binary mask
    return tf_image.resize(mask[..., None], img_size, method="nearest")

def preprocess_image_and_mask(img_path, image_id):
    """Preprocesses the image and generates a corresponding segmentation
mask."""
    # Load image
    input_img = preprocess_image(img_path)
    # Identify which COCO object contains the image ID
    for coco in coco_objects:
        if image_id in coco.getImgIds():
            img_info = coco.loadImgs(image_id)[0]
            img_height, img_width = img_info['height'],
img_info['width']
            mask = generate_segmentation_mask(coco, image_id,
img_height, img_width)
            mask = tf_image.convert_image_dtype(mask, "uint8")
            return input_img, mask
    raise ValueError(f"Image ID {image_id} not found in any annotation
files.")

# Dataset Function
def get_dataset(batch_size, img_paths, image_ids, preprocess_fn,
max_dataset_len=None):
    """Creates a dataset for images and COCO segmentation masks."""
    if max_dataset_len:
        img_paths = img_paths[:max_dataset_len]
        image_ids = image_ids[:max_dataset_len]
    dataset = tf_data.Dataset.from_tensor_slices((img_paths, image_ids))
    dataset = dataset.map(preprocess_fn,
num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

# Main Execution
# Load all annotation files
coco_objects = load_coco_annotations(annotation_dir)

# Extract image IDs and paths
```

```
image_ids, input_img_paths = get_image_paths_and_ids(coco_objects,
input_dir)

# Validate that all referenced images exist
for img_path in input_img_paths:
    if not os.path.exists(img_path):
        raise FileNotFoundError(f"Image file not found: {img_path}")

# Random shuffling
val_samples = 1000
np.random.seed(1337)
shuffled_indices = np.random.permutation(len(image_ids))

# Split into train/val
train_indices = shuffled_indices[:-val_samples]
val_indices = shuffled_indices[-val_samples:]
train_img_paths = [input_img_paths[i] for i in train_indices]
train_image_ids = [image_ids[i] for i in train_indices]
val_img_paths = [input_img_paths[i] for i in val_indices]
val_image_ids = [image_ids[i] for i in val_indices]

# Training dataset
train_dataset = get_dataset(
    batch_size,
    train_img_paths,
    train_image_ids,
    preprocess_image_and_mask,
    max_dataset_len=1000
)

# Validation dataset
val_dataset = get_dataset(
    batch_size,
    val_img_paths,
    val_image_ids,
    preprocess_image_and_mask
)
```

load_img_masks(input_img_path, target_img_path)

This helper function processes a pair of input and target image file paths by performing the following:

Load the input image:

tf_io.read_file(input_img_path):

Reads the image file from the given path as a raw binary string.

tf_io.decode_png(input_img, channels=3):

Decodes the PNG file into a 3-channel (RGB) image tensor.

Resize the input image:

tf_image.resize(input_img, img_size):

Resizes the image to the dimensions specified in img_size.

Normalize the input image:

tf_image.convert_image_dtype(input_img, "float32"):

Converts the image's pixel values to the range [0, 1] by casting them to float32.

```
 import os
import random
import xml.etree.ElementTree as ET
import numpy as np
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io

# Directory paths
input_dir = "images/"
annotation_dir = "annotations/xmls/"
img_size = (160, 160)
batch_size = 32

# Load input image paths
input_img_paths = sorted(
    [
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ]
)

# Load annotation paths
annotation_paths = sorted(
    [
        os.path.join(annotation_dir, fname)
        for fname in os.listdir(annotation_dir)
        if fname.endswith(".xml")
    ]
)

print("Number of samples:", len(input_img_paths))
for input_path, annotation_path in zip(input_img_paths[:10], annotation_paths[:10]):
    print(input_path, "|", annotation_path)

# Parse XML Annotations
def parse_xml(annotation_path):
    tree = ET.parse(annotation_path)
    root = tree.getroot()
    bboxes = []
    for obj in root.findall("object"):
        bbox = obj.find("bndbox")
        class_name = obj.find("name").text
        xmin = int(bbox.find("xmin").text)
        ymin = int(bbox.find("ymin").text)
        xmax = int(bbox.find("xmax").text)
        ymax = int(bbox.find("ymax").text)
        bboxes.append({
            "class": class_name,
            "bbox": [xmin, ymin, xmax, ymax]
        })
    return bboxes

# Example: Normalize bounding boxes and extract class IDs
def preprocess_bboxes(annotation_path, img_width, img_height):
    parsed = parse_xml(annotation_path)
    bboxes = []
    class_ids = []
    for item in parsed:
        # Normalize bounding box coordinates
        xmin, ymin, xmax, ymax = item["bbox"]
        bboxes.append([
            xmin / img_width, ymin / img_height,
            xmax / img_width, ymax / img_height
        ])
        # Example: Map class name to integer ID
        class_id = {"class1": 0, "class2": 1, "class3": 2}.get(item["class"], -1)
        class_ids.append(class_id)
    return np.array(bboxes, dtype=np.float32), np.array(class_ids, dtype=np.int32)

# Preprocessing Function for Dataset
def preprocess_images_and_annotations(input_img_path, annotation_path):
    # Preprocess image
    input_img = tf_io.read_file(input_img_path)
    input_img = tf_io.decode_png(input_img, channels=3)
    input_img = tf_image.resize(input_img, img_size)
    input_img = tf_image.convert_image_dtype(input_img, "float32")

    # Get original dimensions
    img_shape = tf_io.read_file(input_img_path)
    img_shape = tf_image.decode_png(img_shape, channels=3).shape[:2]
    img_height, img_width = img_shape

    # Preprocess bounding boxes and class IDs
    bboxes, class_ids = preprocess_bboxes(annotation_path, img_width, img_height)
    return input_img, (bboxes, class_ids)

# General Dataset Function
def get_dataset(
    batch_size,
    input_img_paths,
    annotation_paths,
    preprocess_fn,
    max_dataset_len=None,
):
    """Creates a TF Dataset for images and annotations."""
    if max_dataset_len:
        input_img_paths = input_img_paths[:max_dataset_len]
        annotation_paths = annotation_paths[:max_dataset_len]

    dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, annotation_paths))
    dataset = dataset.map(preprocess_fn, num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

# Random shuffling
val_samples = 1000
random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(annotation_paths)

# Splitting into training and validation sets
train_input_img_paths = input_img_paths[:-val_samples]
train_annotation_paths = annotation_paths[:-val_samples]
val_input_img_paths = input_img_paths[-val_samples:]
val_annotation_paths = annotation_paths[-val_samples:]

# Create training dataset
train_dataset = get_dataset(
    batch_size,
    train_input_img_paths,
    train_annotation_paths,
    preprocess_images_and_annotations,
    max_dataset_len=1000,
)
```

```
# Create validation dataset
val_dataset = get_dataset(
    batch_size,
    val_input_img_paths,
    val_annotation_paths,
    preprocess_images_and_annotations,
    max_dataset_len=1000,
)
```