Perfect! Here's a **full TensorFlow U-Net training pipeline** for:

- ✅ Grayscale **input images**
- ✅ Grayscale **index masks** (each pixel is a class ID: 0, 1, 2, ...)
- ✅ tf.data.Dataset with augmentation
- ✅ Loss: sparse_categorical_crossentropy

📁 Folder Structure (Expected)

```
dataset/
├── images/
│   ├── 0001.png  ← grayscale image
│   ├── 0002.png
├── masks/
│   ├── 0001.png  ← grayscale mask (0,1,...)
│   ├── 0002.png
```

✅ Full Training Pipeline in TensorFlow

```
import tensorflow as tf
import os
import cv2
import numpy as np
from tensorflow.keras import layers, models

# ----------------------------
# PARAMETERS
# ----------------------------
IMAGE_HEIGHT = 256
IMAGE_WIDTH = 256
NUM_CLASSES = 4  # update this
BATCH_SIZE = 8
EPOCHS = 25

# ----------------------------
# PATHS
# ----------------------------
IMAGE_DIR = 'dataset/images'
MASK_DIR = 'dataset/masks'

image_paths = sorted([os.path.join(IMAGE_DIR, fname) for fname in os.listdir(IMAGE_DIR)])
mask_paths = sorted([os.path.join(MASK_DIR, fname) for fname in os.listdir(MASK_DIR)])

# ----------------------------
# DATA LOADING FUNCTION
# ----------------------------
def load_image_and_mask(image_path, mask_path):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=1)  # grayscale input
    image = tf.image.resize(image, [IMAGE_HEIGHT, IMAGE_WIDTH])
    image = tf.cast(image, tf.float32) / 255.0
```

```
    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=1)
    mask = tf.image.resize(mask, [IMAGE_HEIGHT, IMAGE_WIDTH], method='nearest')
    mask = tf.squeeze(mask, axis=-1)  # shape: (H, W)
    mask = tf.cast(mask, tf.int32)

    return image, mask

# ----------------------------
# tf.data.Dataset Preparation
# ----------------------------
def data_generator(image_paths, mask_paths, augment=False):
    dataset = tf.data.Dataset.from_tensor_slices((image_paths, mask_paths))

    def _map_fn(img, msk):
        img, msk = load_image_and_mask(img, msk)
        if augment:
            if tf.random.uniform(()) > 0.5:
                img = tf.image.flip_left_right(img)
                msk = tf.image.flip_left_right(msk)
        return img, msk

    dataset = dataset.map(_map_fn, num_parallel_calls=tf.data.AUTOTUNE)
    return dataset

train_size = int(0.8 * len(image_paths))
train_dataset = data_generator(image_paths[:train_size], mask_paths[:train_size], augment=True)
val_dataset = data_generator(image_paths[train_size:], mask_paths[train_size:], augment=False)

train_dataset = train_dataset.shuffle(100).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
val_dataset = val_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

# ----------------------------
# U-Net Model
# ----------------------------
def unet(input_size=(IMAGE_HEIGHT, IMAGE_WIDTH, 1), num_classes=NUM_CLASSES):
    inputs = layers.Input(input_size)

    # Encoder
    c1 = layers.Conv2D(64, 3, activation='relu', padding='same')(inputs)
    c1 = layers.Conv2D(64, 3, activation='relu', padding='same')(c1)
    p1 = layers.MaxPooling2D((2, 2))(c1)

    c2 = layers.Conv2D(128, 3, activation='relu', padding='same')(p1)
    c2 = layers.Conv2D(128, 3, activation='relu', padding='same')(c2)
    p2 = layers.MaxPooling2D((2, 2))(c2)

    # Bottleneck
    b = layers.Conv2D(256, 3, activation='relu', padding='same')(p2)
    b = layers.Conv2D(256, 3, activation='relu', padding='same')(b)

    # Decoder
    u2 = layers.UpSampling2D((2, 2))(b)
    u2 = layers.concatenate([u2, c2])
    c3 = layers.Conv2D(128, 3, activation='relu', padding='same')(u2)
    c3 = layers.Conv2D(128, 3, activation='relu', padding='same')(c3)

    u1 = layers.UpSampling2D((2, 2))(c3)
    u1 = layers.concatenate([u1, c1])
    c4 = layers.Conv2D(64, 3, activation='relu', padding='same')(u1)
```

```
    c4 = layers.Conv2D(64, 3, activation='relu', padding='same')(c4)

    outputs = layers.Conv2D(num_classes, 1, activation='softmax')(c4)

    return models.Model(inputs, outputs)

model = unet()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

# ----------------------------
# TRAINING
# ----------------------------
model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=EPOCHS
)

# ----------------------------
# SAVE MODEL
# ----------------------------
model.save("unet_grayscale_tf.h5")
```

✅ What This Covers:

- Grayscale images and masks
- Masks contain integer values (class indices like 0,1,...)
- U-Net with `softmax` output
- `sparse_categorical_crossentropy` loss (does NOT require one-hot masks)
- Augmentation (horizontal flip)
- Uses `tf.data` for performance and scalability

CODE :

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.applications import EfficientNetB0

from tensorflow.keras.callbacks import ModelCheckpoint

import numpy as np

import os

import matplotlib.pyplot as plt
```

```
# Parameters
IMG_HEIGHT = 224
IMG_WIDTH = 224
NUM_CLASSES = 3
BATCH_SIZE = 4
EPOCHS = 50

# Helper function to get sorted PNG file paths without glob
def sorted_png_files(folder):
    files = [f for f in os.listdir(folder) if f.endswith('.png')]
    files = sorted(files)
    full_paths = [os.path.join(folder, f) for f in files]
    return full_paths

# Paths — adjust to your folders
image_paths = sorted_png_files('./images')
mask_paths = sorted_png_files('./masks')
split_idx = int(0.8 * len(image_paths))  # 80/20 split

# Load image + mask (grayscale, resize, normalize)
def load_image_mask(img_path, mask_path):
    img = tf.io.read_file(img_path)
    img = tf.image.decode_png(img, channels=1)
    img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
    img = tf.cast(img, tf.float32) / 255.0

    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=1)
    mask = tf.image.resize(mask, [IMG_HEIGHT, IMG_WIDTH], method='nearest')
    mask = tf.cast(mask, tf.uint8)
```

```
    return img, mask

# Offline data augmentation
def augment(img, mask):
    if tf.random.uniform(()) > 0.5:
        img = tf.image.flip_left_right(img)
        mask = tf.image.flip_left_right(mask)
    if tf.random.uniform(()) > 0.5:
        img = tf.image.flip_up_down(img)
        mask = tf.image.flip_up_down(mask)
    img = tf.image.random_brightness(img, max_delta=0.1)
    return img, mask

def load_and_augment(img_path, mask_path):
    img, mask = load_image_mask(img_path, mask_path)
    img, mask = augment(img, mask)
    return img, mask

def tf_dataset(img_paths, mask_paths, batch_size=4, augment=False):
    dataset = tf.data.Dataset.from_tensor_slices((img_paths, mask_paths))
    if augment:
        dataset = dataset.map(load_and_augment, num_parallel_calls=tf.data.AUTOTUNE)
    else:
        dataset = dataset.map(lambda x, y: load_image_mask(x, y),
num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.batch(batch_size).prefetch(tf.data.AUTOTUNE)
    return dataset

# Create datasets
train_dataset = tf_dataset(image_paths[:split_idx], mask_paths[:split_idx],
batch_size=BATCH_SIZE, augment=True)
val_dataset = tf_dataset(image_paths[split_idx:], mask_paths[split_idx:], batch_size=BATCH_SIZE,
augment=False)
```

```
# Build U-Net with EfficientNetB0 (no ImageNet weights)
def build_unet_no_imagenet(input_shape=(IMG_HEIGHT, IMG_WIDTH, 1), num_classes=NUM_CLASSES):
    inputs = layers.Input(shape=input_shape)
    x = layers.Concatenate()([inputs, inputs, inputs])  # grayscale → 3 channels

    base_model = EfficientNetB0(include_top=False, weights=None, input_tensor=x)

    skips = [
        base_model.get_layer("block2a_activation").output,
        base_model.get_layer("block3a_activation").output,
        base_model.get_layer("block4a_activation").output,
        base_model.get_layer("block6a_activation").output,
    ]
    x = base_model.output

    for skip in reversed(skips):
        x = layers.UpSampling2D()(x)
        x = layers.Concatenate()([x, skip])
        x = layers.Conv2D(256, 3, padding='same', activation='relu')(x)
        x = layers.Conv2D(256, 3, padding='same', activation='relu')(x)

    x = layers.UpSampling2D()(x)
    x = layers.Conv2D(128, 3, padding='same', activation='relu')(x)
    x = layers.Conv2D(64, 3, padding='same', activation='relu')(x)

    outputs = layers.Conv2D(num_classes, 1, activation='softmax')(x)
    return models.Model(inputs, outputs)

# Instantiate and compile model
model = build_unet_no_imagenet()
model.compile(
```

```
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Checkpoint callback
checkpoint_cb = ModelCheckpoint(
    "best_sar_unet_model.h5",
    save_best_only=True,
    monitor="val_loss",
    mode="min"
)

# Train model
model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=EPOCHS,
    callbacks=[checkpoint_cb]
)

# --- Prediction and save masks ---

output_pred_folder = './predicted_masks'
os.makedirs(output_pred_folder, exist_ok=True)

def predict_and_save_masks(model, image_paths, output_folder):
    for img_path in image_paths:
        img = tf.io.read_file(img_path)
        img = tf.image.decode_png(img, channels=1)
        img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
        img = tf.cast(img, tf.float32) / 255.0
```

```
        input_img = tf.expand_dims(img, axis=0)

        pred = model.predict(input_img)[0]
        pred_mask = tf.argmax(pred, axis=-1)
        pred_mask = tf.cast(pred_mask, tf.uint8).numpy()

        base_name = os.path.basename(img_path)
        name_wo_ext = os.path.splitext(base_name)[0]
        save_path = os.path.join(output_folder, f'{name_wo_ext}_pred.png')
        tf.keras.preprocessing.image.save_img(save_path, pred_mask[..., None], scale=False)
        print(f"Saved predicted mask: {save_path}")

# Run prediction on test set
test_image_paths = image_paths[split_idx:]
predict_and_save_masks(model, test_image_paths, output_pred_folder)

# --- Metrics computation ---

def compute_iou(y_true, y_pred, num_classes=3):
    ious = []
    for cls in range(num_classes):
        true_cls = (y_true == cls)
        pred_cls = (y_pred == cls)
        intersection = np.logical_and(true_cls, pred_cls).sum()
        union = np.logical_or(true_cls, pred_cls).sum()
        if union == 0:
            ious.append(np.nan)
        else:
            ious.append(intersection / union)
    return np.nanmean(ious)

def compute_dice(y_true, y_pred, num_classes=3):
```

```
    dices = []
    for cls in range(num_classes):
        true_cls = (y_true == cls)
        pred_cls = (y_pred == cls)
        intersection = 2 * np.logical_and(true_cls, pred_cls).sum()
        total = true_cls.sum() + pred_cls.sum()
        if total == 0:
            dices.append(np.nan)
        else:
            dices.append(intersection / total)
    return np.nanmean(dices)

# Load saved predicted and ground truth masks
pred_mask_paths = sorted_png_files(output_pred_folder)
true_mask_paths = sorted_png_files('./masks')

ious = []
dices = []

for pred_path, true_path in zip(pred_mask_paths, true_mask_paths):
    pred_mask = tf.io.read_file(pred_path)
    pred_mask = tf.image.decode_png(pred_mask, channels=1)
    pred_mask = tf.squeeze(pred_mask).numpy()

    true_mask = tf.io.read_file(true_path)
    true_mask = tf.image.decode_png(true_mask, channels=1)
    true_mask = tf.squeeze(true_mask).numpy()

    iou = compute_iou(true_mask, pred_mask, num_classes=NUM_CLASSES)
    dice = compute_dice(true_mask, pred_mask, num_classes=NUM_CLASSES)

    ious.append(iou)
```

```
    dices.append(dice)

print(f"Mean IoU over test set: {np.nanmean(ious):.4f}")
print(f"Mean Dice over test set: {np.nanmean(dices):.4f}")

# --- Optional: visualize example prediction ---

def visualize_prediction(model, img_path, mask_path):
    img = tf.io.read_file(img_path)
    img = tf.image.decode_png(img, channels=1)
    img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
    img = tf.cast(img, tf.float32) / 255.0
    input_img = tf.expand_dims(img, axis=0)

    pred = model.predict(input_img)[0]
    pred_mask = np.argmax(pred, axis=-1)

    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=1)
    mask = tf.image.resize(mask, [IMG_HEIGHT, IMG_WIDTH], method='nearest')
    mask = tf.squeeze(mask).numpy()

    plt.figure(figsize=(12,4))
    plt.subplot(1,3,1)
    plt.title('Input Image')
    plt.imshow(tf.squeeze(img), cmap='gray')
    plt.axis('off')

    plt.subplot(1,3,2)
    plt.title('True Mask')
    plt.imshow(mask, cmap='jet', vmin=0, vmax=NUM_CLASSES-1)
    plt.axis('off')
```

plt.subplot(1,3,3)

plt.title('Predicted Mask')

plt.imshow(pred_mask, cmap='jet', vmin=0, vmax=NUM_CLASSES-1)

plt.axis('off')

plt.show()

Visualize first test image prediction

visualize_prediction(model, test_image_paths[0], true_mask_paths[0])

PIXEL GEN CODE:

Awesome! Here's how to **modify your existing code** to include class region extraction and print region info when visualizing predictions.

Step 1: Add this helper function near the top of your script

```
import numpy as np

def get_class_regions(mask, num_classes):
    class_coords = {}
    class_bboxes = {}

    for cls in range(num_classes):
        coords = np.argwhere(mask == cls)
        class_coords[cls] = coords

        if coords.size == 0:
            class_bboxes[cls] = None
        else:
            min_row, min_col = coords.min(axis=0)
            max_row, max_col = coords.max(axis=0)
            class_bboxes[cls] = (min_row, min_col, max_row, max_col)

    return class_coords, class_bboxes
```

Step 2: Modify your `visualize_prediction` function to print region info:

Find your `visualize_prediction` function and **replace it with this updated version:**

```
def visualize_prediction(model, img_path, mask_path):
```

```
    img = tf.io.read_file(img_path)
    img = tf.image.decode_png(img, channels=1)
    img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
    img = tf.cast(img, tf.float32) / 255.0
    input_img = tf.expand_dims(img, axis=0)

    pred = model.predict(input_img)[0]
    pred_mask = np.argmax(pred, axis=-1)

    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=1)
    mask = tf.image.resize(mask, [IMG_HEIGHT, IMG_WIDTH], method='nearest')
    mask = tf.squeeze(mask).numpy()

    # Get class regions and bounding boxes for the true mask
    class_coords, class_bboxes = get_class_regions(mask,
num_classes=NUM_CLASSES)
    print("Class regions info for TRUE mask:")
    for cls in range(NUM_CLASSES):
        print(f"Class {cls}: pixels={len(class_coords[cls])},
bbox={class_bboxes[cls]}")

    # Also get for predicted mask if you want
    pred_class_coords, pred_class_bboxes = get_class_regions(pred_mask,
num_classes=NUM_CLASSES)
    print("Class regions info for PREDICTED mask:")
    for cls in range(NUM_CLASSES):
        print(f"Class {cls}: pixels={len(pred_class_coords[cls])},
bbox={pred_class_bboxes[cls]}")

    plt.figure(figsize=(12,4))
    plt.subplot(1,3,1)
    plt.title('Input Image')
    plt.imshow(tf.squeeze(img), cmap='gray')
    plt.axis('off')

    plt.subplot(1,3,2)
    plt.title('True Mask')
    plt.imshow(mask, cmap='jet', vmin=0, vmax=NUM_CLASSES-1)
    plt.axis('off')

    plt.subplot(1,3,3)
    plt.title('Predicted Mask')
    plt.imshow(pred_mask, cmap='jet', vmin=0, vmax=NUM_CLASSES-1)
    plt.axis('off')

    plt.show()
```

What this does:

- For both **true mask** and **predicted mask**, it finds:
  - How many pixels belong to each class
  - The bounding box (min_row, min_col, max_row, max_col) covering that class region
- Prints those details before showing the plots.

If you run this and visualize any sample, you'll see console output like:

```
Class regions info for TRUE mask:
Class 0: pixels=12345, bbox=(0, 0, 223, 223)
Class 1: pixels=3200, bbox=(50, 50, 100, 100)
Class 2: pixels=2800, bbox=(120, 120, 170, 170)
...
Class regions info for PREDICTED mask:
Class 0: pixels=...
...
```
