Synthetic Segmentation Training Data Generator

| Project Overview |

AI-Powered Innovation

Content:

Introduction

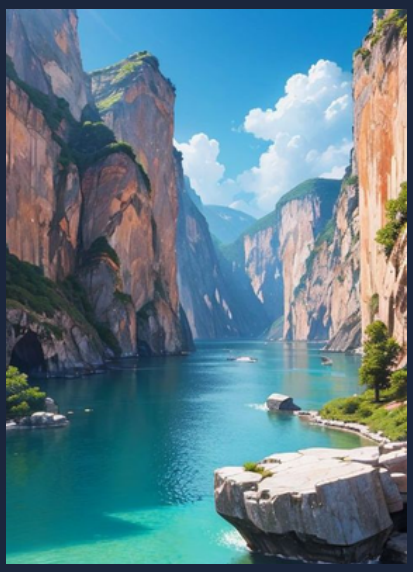
Project Title: Synthetic Segmentation Training Data Generator

problem:

- Training data is hard to get by and expensive.
- develop a model to generate high-quality synthetic data sets

How:

- Leveraged the use of GAN model
- using a discriminator
- we achieve high quality artificial images with realistic qualities



BRIEF PROCESS OVERVIEW

Step 1:

- Researching and Understanding Potential Solutions

Step 2:

- Data Collection and Preparation

Step 3:

- Model Design and Setup

Step 4:

- Training the GAN

Step 5:

- Evaluation and Fine-Tuning

Step 6:

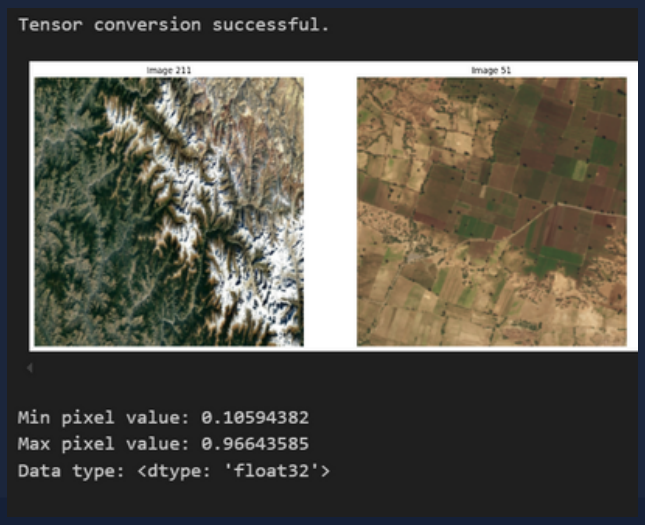
- Deployment and Application

A CLOSER LOOK - RAW IMAGES

IMAGES HAD TO BE RESIZED, NORMALIZING PIXEL VALUES, AND ENSURING THE OVERLL AUGMENTATION OF THE DATASET IS SUITABLE .

TENSOR CONVERSION SUCCESSFUL:

- USING TENSORFLOW
- THE MINIMUM & MAXIMUM PIXEL VALUES
- THE TENSOR DATA TYPE - IS FLOAT32 FORMAT FOR NEURAL NETWORK INPUTS,



A CLOSER LOOK - TENSORFLOW CONVERSION

STEPS THE CODE FOLLOWS:

1. CONVERT AND SCALE IMAGES
2. RESIZED IMAGES
3. COLOUR CHANNELS OF THE IMAGES
4. TENSOR CONVERSION WITH RANDOM DISPLAY
5. PIXEL VALUE AND DATA TYPE CHECK

```python
# TPreprocessing images - adjusting size and format to make them ready for further processing

def preprocess_image(image_array, target_size=(256, 256)):
    # Convert the image to float32 to reduce memory consumption
    image_array = image_array.astype(np.float32) / 255.0

    # Resize image
    image_resized = resize(image_array, target_size, anti_aliasing=True)

    # Standardize color channels: ensure all images are in RGB format
    if image_resized.ndim == 2:  # If the image is grayscale, convert it to RGB
        image_resized = gray2rgb(image_resized)
    elif image_resized.shape[2] == 4:  # If the image has an alpha channel, remove it
        image_resized = image_resized[..., :3]

    return image_resized

# Process list of images individually, applying transformations to each image for consistent size and colour format
# Managing memory usage effectively & prevent crashes due to memory overflow
images = []
for img in image_list:
    try:
        processed_image = preprocess_image(img)
        images.append(processed_image)
    except MemoryError as e:
        print(f"MemoryError while processing the image: {e}")
# Handle the memory error, (e.g., by skipping the image or freeing up resources)
        break  # Exit the loop
```

```python
# Convert list of processed images into a TensorFlow tensor
# Displays five random images from this tensor; if no images are processed - print a statement to inform
if images:
    images_tensor = tf.convert_to_tensor(images, dtype=tf.float32)
    print("Tensor conversion successful.")

# Display 5 random images
    def display_random_images(images, num_images=5):
        plt.figure(figsize=(40, 20))
        indices = np.random.choice(images.shape[0], num_images, replace=False)
        for i, index in enumerate(indices):
            ax = plt.subplot(1, num_images, i + 1)
            plt.imshow(images[index])
            plt.title(f"Image {index+1}")
            plt.axis("off")
        plt.show()

    display_random_images(images_tensor)
else:
    print("No images that were processed.")
    print("Tensor conversion successful.")

# Verify the minimum and maximum pixel values and the data type
# Helps ensure that the image data is correctly formatted and scaled for processing tasks

print("Min pixel value:", np.min(images_tensor[0]))
print("Max pixel value:", np.max(images_tensor[0]))

# Ensure that the dtype is float32
print("Data type:", images_tensor.dtype)
```

CREATING A DATAFRAME FOR SEGMENTING IMAGES

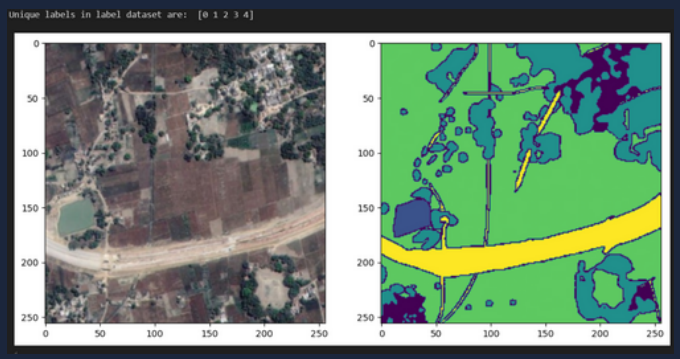
- CREATING A DATAFRAME FOR SEGMENTING IMAGES.
- LABELING THE IMAGES
- DISPLAYING RGB VALUES OF THE DIFFERENT IMAGE LABELS

```python
import pandas as pd
from io import StringIO

    # Specify the bucket name and file name
bucket_name = '2307-07-synthetic-segmentation-a'
file_key = 'class_dict_seg.csv'

#Initialize S3 client
s3_client = boto3.client(
service_name='s3',
region_name='eu-west-1',
aws_access_key_id=os.getenv("aws_access_key_id"),
aws_secret_access_key=os.getenv("aws_secret_access_key"))

def read_csv_from_s3(bucket_name, file_key):

    # Read CSV file from S3
    obj = s3_client.get_object(Bucket=bucket_name, Key=file_key)
    data = obj['Body'].read().decode('utf-8')

    # Convert CSV data to Pandas DataFrame
    mask_labels = pd.read_csv(StringIO(data))

    return mask_labels

mask_labels = read_csv_from_s3(bucket_name, file_key)
print(mask_labels.head())  # Display the first few rows of the DataFrame
```

```
     name    r    g    b
0   urban    0  255  255
1   water    0    0  255
2  forest    0  255    0
3    land  255  255    0
4    road  255    0  255
```

CONVERTING RGB IMAGES & SEGMENTING

- CONVERTING RGB IMAGES
- CLASS LABELING OF THE IMAGES
- DISPLAYING RGB VALUES OF IMAGE LABELS



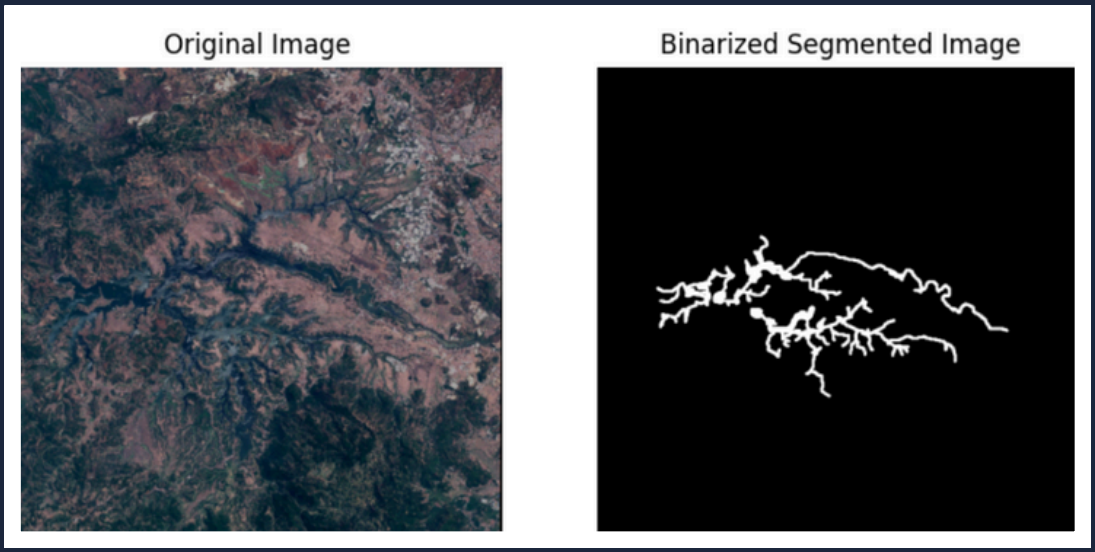

```python
# converting 3 values (RGB) to 1 label values...
def rgb_to_labels(img, mask_labels):
    label_seg = np.zeros(img.shape,dtype=np.uint8)
    for i in range(mask_labels.shape[0]):
        label_seg[np.all(img == list(mask_labels.iloc[i, [1,2,3]]), axis=-1)] = i
    label_seg = label_seg[:,:,0]  #Just take the first channel, no need for all 3 channels...
    return label_seg
```

```python
labels = []
for i in range(mask_dataset.shape[0]):
    label = rgb_to_labels(mask_dataset[i], mask_labels) # calling rgb to labels for each images..
    labels.append(label)
```

```python
labels = np.array(labels)
labels = np.expand_dims(labels, axis=3) # adding channel dim...
```

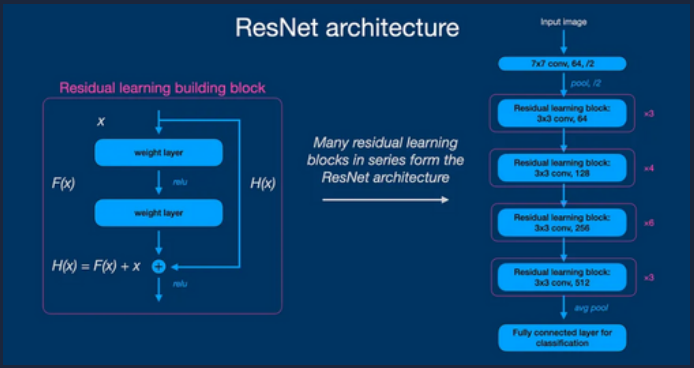
LABEL STUDIO

DEMO IMAGE FROM LABEL STUDIO



TRAINING THE SEGMENTATION MODEL

- RESNET MODEL
- DEFINING THE MODEL
- SUMMARY OF THE MODEL ARCHITECTURE



Modeling

```
# using weights from pretained resnet model...
BACKBONE = 'resnet34'
preprocess_input = sm.get_preprocessing(BACKBONE)
```

```
# preprocess input...
X_train_prepr = preprocess_input(X_train)
X_test_prepr = preprocess_input(X_test)
```

```
# define model...
model_resnet_backbone = sm.Unet(BACKBONE, encoder_weights='imagenet', classes=n_classes, activation='softmax')
```

```
# accuracy as metrics, adam optimizer and loss function that needs to be optimized is categorical crossentropy as its a multiclass classification problem...
metrics=['accuracy']
model_resnet_backbone.compile(optimizer='adam', loss='categorical_crossentropy', metrics=metrics)
```

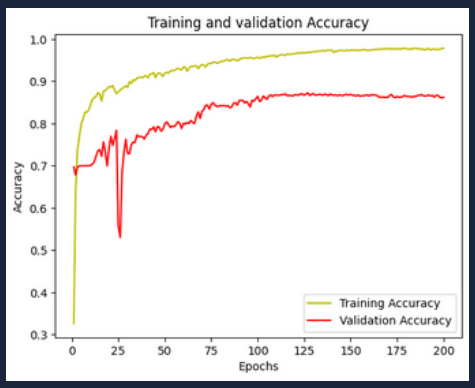
Model summary

```
model_resnet_backbone.summary()
```

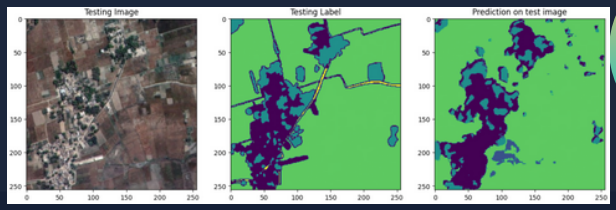
```
Model: "functional_3"
```

TRAINING RESULTS AFTER 200 EPOCHS

- ACCURACY - TRAINING
- CODE INITIATION AND SETTING PARAMETERS
- TRAINING ACCURACY - 0.95
- VALIDATION ACCURACY- 0.86
- LOSS FUNCTION - 0.12




```
# training the model for 80 epochs with batch size of 16...
history = model_resnet_backbone.fit(X_train_prepr,
          y_train,
          batch_size=16,
          epochs=200,
          verbose=1,
          validation_data=(X_test_prepr, y_test))
```



BUILDING A GENERATOR MODEL

THE ARTIST
THE GENERATOR

```python
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, BatchNormalization, LeakyReLU, Activation, Concatenate
from tensorflow.keras import Input, Model

def build_generator():
    inputs = Input(shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))

    # Encoder
    e1 = Conv2D(64, (4, 4), strides=(2, 2), padding='same')(inputs)
    e1 = LeakyReLU(alpha=0.2)(e1)

    e2 = Conv2D(128, (4, 4), strides=(2, 2), padding='same')(e1)
    e2 = BatchNormalization()(e2)
    e2 = LeakyReLU(alpha=0.2)(e2)

    e3 = Conv2D(256, (4, 4), strides=(2, 2), padding='same')(e2)
    e3 = BatchNormalization()(e3)
    e3 = LeakyReLU(alpha=0.2)(e3)

    # Decoder
    d1 = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')(e3)
    d1 = BatchNormalization()(d1)
    d1 = Activation('relu')(d1)

    d2 = Conv2DTranspose(64, (4, 4), strides=(2, 2), padding='same')(d1)
    d2 = BatchNormalization()(d2)
    d2 = Activation('relu')(d2)

    outputs = Conv2DTranspose(IMG_CHANNELS, (4, 4), strides=(2, 2), padding='same')(d2)
    outputs = Activation('tanh')(outputs)

    return Model(inputs, outputs)
```

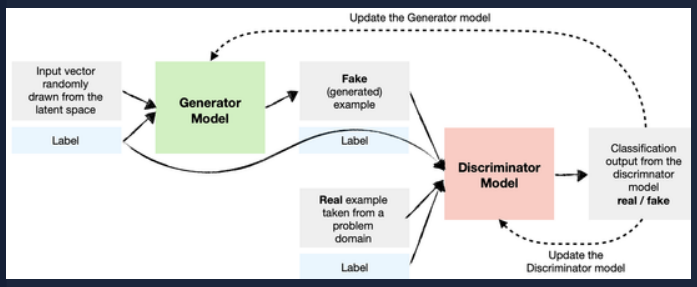
BUILDING A DISCRIMINATOR

THE DISCRIMINATOR ARCHITECTURE

```python
# Discriminator
def build_discriminator():
    inputs = Input(shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
    targets = Input(shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
    combined = Concatenate()([inputs, targets])
    # Add layers for the discriminator
    down1 = Conv2D(64, kernel_size=4, strides=2, padding='same')(combined)
    down1 = LeakyReLU(alpha=0.2)(down1)
    down2 = Conv2D(128, kernel_size=4, strides=2, padding='same')(down1)
    down2 = BatchNormalization()(down2)
    down2 = LeakyReLU(alpha=0.2)(down2)
    down3 = Conv2D(256, kernel_size=4, strides=2, padding='same')(down2)
    down3 = BatchNormalization()(down3)
    down3 = LeakyReLU(alpha=0.2)(down3)
    down4 = Conv2D(512, kernel_size=4, strides=2, padding='same')(down3)
    down4 = BatchNormalization()(down4)
    down4 = LeakyReLU(alpha=0.2)(down4)
    outputs = Conv2D(1, kernel_size=4, strides=1, padding='same', activation='sigmoid')(down4)
    return Model([inputs, targets], outputs)
```

COMBINING THE TWO

SYNERGISED PROCESS




```python
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model

# Compile discriminator
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(2e-4, beta_1=0.5), metrics=['accuracy'])

# For the combined model, only train the generator
discriminator.trainable = False

# Input images and their conditioning images
input_img = Input(shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
gen_output = generator(input_img)

# Discriminator determines validity of translated images / condition pairs
dis_output = discriminator([input_img, gen_output])

# Combined model (stacked generator and discriminator)
combined = Model(inputs=input_img, outputs=[dis_output, gen_output])
combined.compile(loss=['binary_crossentropy', 'mae'], optimizer=Adam(2e-4, beta_1=0.5))
```

TRAINING THE MODEL TO GENERATE SYNTHETIC IMAGES

MODEL IS NOW TRAINED TO DISPLAY REAL, FAKE AND MASK IMAGES

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 288ms/step
[Epoch 301/311] [D loss: 5.780974388122559] [G loss: 5.7764739990234375] time: 6.717052221298218
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 292ms/step
[Epoch 302/311] [D loss: 5.7811994552612305] [G loss: 5.776702880859375] time: 5.655746698379517
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 294ms/step
[Epoch 303/311] [D loss: 5.777637481689453] [G loss: 5.773155689239502] time: 5.764461994171143
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 291ms/step
[Epoch 304/311] [D loss: 5.778769493103027] [G loss: 5.774311542510986] time: 7.130734205245972
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 290ms/step
[Epoch 305/311] [D loss: 5.779085159301758] [G loss: 5.774621963500977] time: 4.683357000350952
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 270ms/step
[Epoch 306/311] [D loss: 5.780157566070557] [G loss: 5.775706768035889] time: 4.686371564865112
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 261ms/step
[Epoch 307/311] [D loss: 5.781589508056641] [G loss: 5.777185440063477] time: 6.669575214385986
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 315ms/step
[Epoch 308/311] [D loss: 5.782863616943359] [G loss: 5.778444766998291] time: 5.644251823425293
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 291ms/step
[Epoch 309/311] [D loss: 5.783916473388672] [G loss: 5.779517650604248] time: 3.76678466796875
```

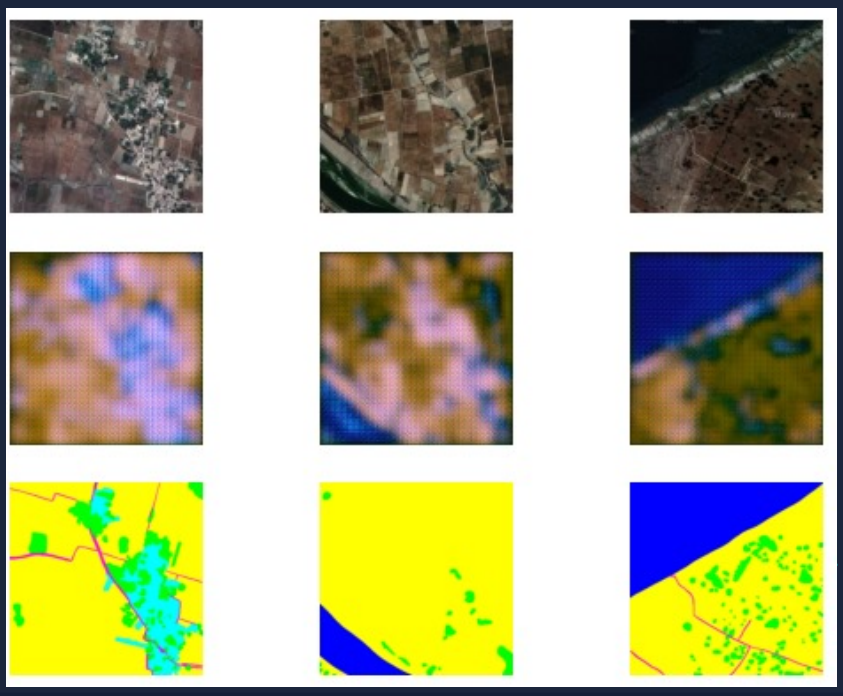
```python
def train(epochs, batch_size, save_interval):
    real = np.ones((batch_size, 16, 16, 1))
    fake = np.zeros((batch_size, 16, 16, 1))

    for epoch in range(epochs):
        start_time = time.time()

        # Train discriminator with a batch of real images
        idx = np.random.randint(0, image_dataset.shape[0], batch_size)
        real_imgs = image_dataset[idx]
        masks = mask_dataset[idx]

        fake_imgs = generator.predict(real_imgs)

        d_loss_real = discriminator.train_on_batch([real_imgs, masks], real)
        d_loss_fake = discriminator.train_on_batch([real_imgs, fake_imgs], fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train generator
        g_loss = combined.train_on_batch(real_imgs, [real, masks])

        elapsed_time = time.time() - start_time

        # Print the progress
        print(f"[Epoch {epoch}/{epochs}] [D loss: {d_loss[0]}] [G loss: {g_loss[0]}] time: {elapsed_time}")

        # If at save interval, save generated image samples
        if epoch % save_interval == 0:
            save_model(epoch)
            sample_images(epoch)

def sample_images(epoch):
    r, c = 3, 3
    idx = np.random.randint(0, image_dataset.shape[0], c)
    real_imgs = image_dataset[idx]
    masks = mask_dataset[idx]
    fake_imgs = generator.predict(real_imgs)
```

DISPLAYING THE RESULTS

THE ORIGINAL

THE PREDICTION

THE CONDITION (SEGMENTED)




THANK YOU