电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

实验报告

EXPERIMENT REPORT

STUDENT NAME:	JAHID SHAHIDUL ISLAM
STUDENT ID:	202324090107
COURSE NAME:	PYTHON PRACTICAL PROGRAMMING
TEACHER NAME:	PROF. RAO YUNBO
EXPERIMENT NO:	FIVE
DATE:	3rd June 2024

1. Experiment title：Install Python Platform

2. Experiment hours：4h Experiment location: Software Building 400

3. Objectives

   At the end of this experiment, you will be able to:

   - At the end of this experiment, you will be able to:
   - How to use Jupyter for SAM.

4. Experimental contents & step

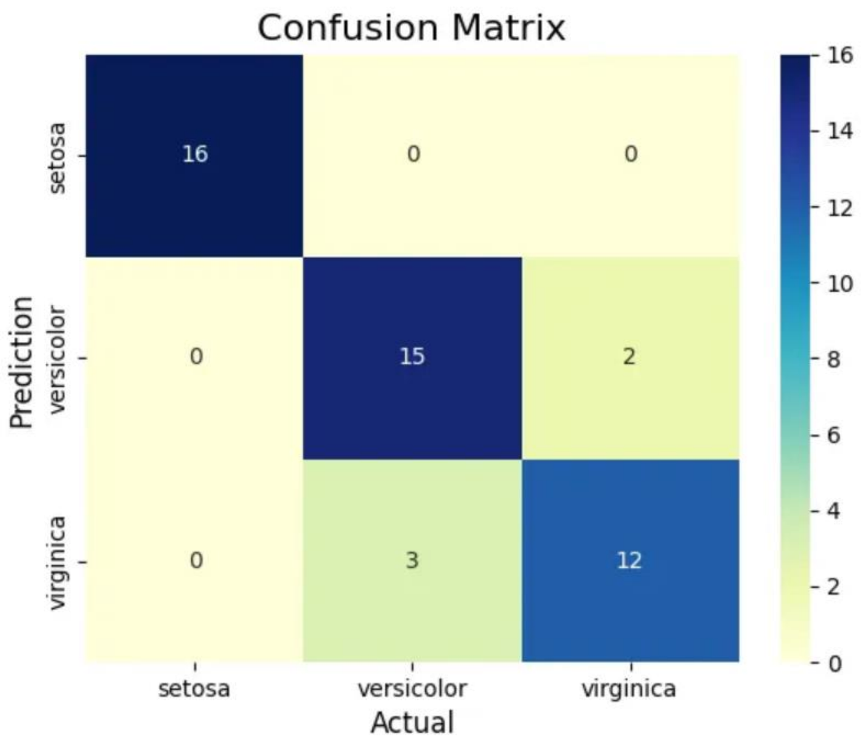
   1) Exercise 1: Triple classification confusion matrix
   2) Exercise 2: Use SegmentationMetric.py to calculate evaluation metrics
   3) Exercise 3: Semantic segmentation of football
   4) Exercise 4: Using the SAM model for image segmentation
   5) Exercise 5: Installing and configuring the SAM environment
   6) Exercise 6: Using SAM auto generate image mask
   7) Exercise 7: Using prompts to choose objects
   8) Exercise 8: Specify a specific goal through additional points
   9) Exercise 9: Specify an object through a box
   10) Exercise 10: Combining points and boxes
   11) Exercise 11: Batch prompts input
   12) Exercise 12: End to end batch processing

5. Experimental analysis

Exercise 1: Triple Classification Confusion Matrix

A "triple classification confusion matrix" typically refers to a confusion matrix generated for a classification problem with three classes. A confusion matrix is a table that is often used to describe the performance of a classification model.

In a triple classification confusion matrix, each axis represents the predicted classes and the actual classes, and the cells in the table represent the counts or proportions of instances that were predicted or misclassified into each class.



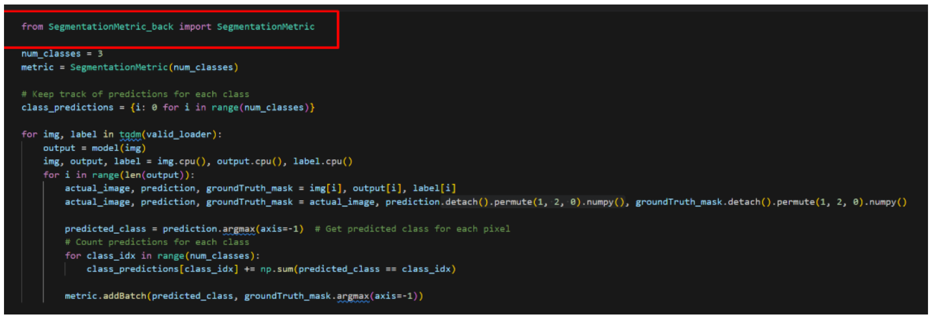
This matrix visualizes the performance of the multi-class classification model by comparing predicted and actual class labels for the Iris dataset.

The Decision Tree classifier correctly identified all 16 serosa examples. It misclassified 2 versicolor examples as virginica, while correctly classifying 15 versicolor instances. Additionally, the classifier misclassified 3 virginica as versicolor, but accurately classified 12 virginica instances. In summary, the Decision Tree classifier performed well on the Iris dataset, with only a few misclassifications between the versicolor and virginica classes.

Exercise 2: Use Segmentationmetric.Py To Calculate Evaluation Metrics

Step 1: Import SegmentationMetric Class

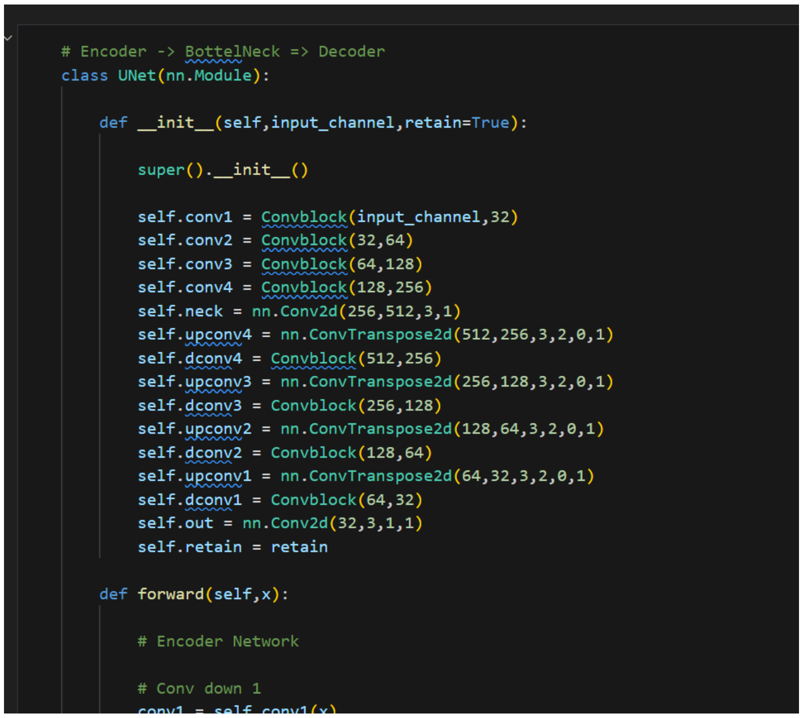
Imported the SegmentationMetric class from the SegmentationMetric_back.py script into my unitV1 model file. This class contains functions for calculating various segmentation evaluation metrics.



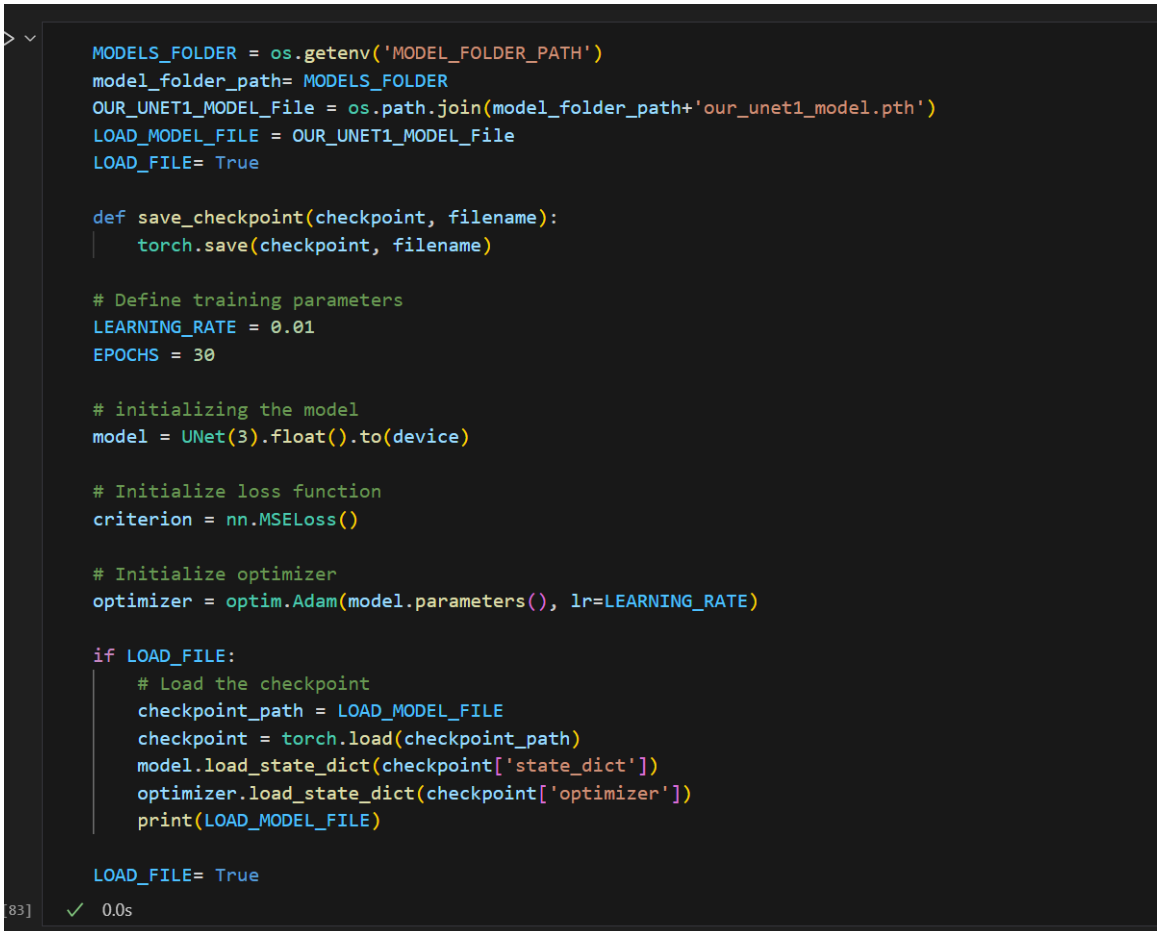
```python
from SegmentationMetric_back import SegmentationMetric

num_classes = 3
metric = SegmentationMetric(num_classes)

# Keep track of predictions for each class
class_predictions = {i: 0 for i in range(num_classes)}

for img, label in tqdm(valid_loader):
    output = model(img)
    img, output, label = img.cpu(), output.cpu(), label.cpu()
    for i in range(len(output)):
        actual_image, prediction, groundTruth_mask = img[i], output[i], label[i]
        actual_image, prediction, groundTruth_mask = actual_image, prediction.detach().permute(1, 2, 0).numpy(), groundTruth_mask.detach().permute(1, 2, 0).numpy()

        predicted_class = prediction.argmax(axis=-1)  # Get predicted class for each pixel
        # Count predictions for each class
        for class_idx in range(num_classes):
            class_predictions[class_idx] += np.sum(predicted_class == class_idx)

        metric.addBatch(predicted_class, groundTruth_mask.argmax(axis=-1))
```

Step 2: Train and Save the unitV1 Model

Trained the unitV1 model on the desired dataset and saved the trained model weights.



```python
# Encoder -> BottelNeck => Decoder
class UNet(nn.Module):

    def __init__(self,input_channel,retain=True):

        super().__init__()

        self.conv1 = Convblock(input_channel,32)
        self.conv2 = Convblock(32,64)
        self.conv3 = Convblock(64,128)
        self.conv4 = Convblock(128,256)
        self.neck = nn.Conv2d(256,512,3,1)
        self.upconv4 = nn.ConvTranspose2d(512,256,3,2,0,1)
        self.dconv4 = Convblock(512,256)
        self.upconv3 = nn.ConvTranspose2d(256,128,3,2,0,1)
        self.dconv3 = Convblock(256,128)
        self.upconv2 = nn.ConvTranspose2d(128,64,3,2,0,1)
        self.dconv2 = Convblock(128,64)
        self.upconv1 = nn.ConvTranspose2d(64,32,3,2,0,1)
        self.dconv1 = Convblock(64,32)
        self.out = nn.Conv2d(32,3,1,1)
        self.retain = retain

    def forward(self,x):

        # Encoder Network

        # Conv down 1
        conv1 = self.conv1(x)
```



```
MODELS_FOLDER = os.getenv('MODEL_FOLDER_PATH')
model_folder_path= MODELS_FOLDER
OUR_UNET1_MODEL_File = os.path.join(model_folder_path+'our_unet1_model.pth')
LOAD_MODEL_FILE = OUR_UNET1_MODEL_File
LOAD_FILE= True

def save_checkpoint(checkpoint, filename):
    torch.save(checkpoint, filename)

# Define training parameters
LEARNING_RATE = 0.01
EPOCHS = 30

# initializing the model
model = UNet(3).float().to(device)

# Initialize loss function
criterion = nn.MSELoss()

# Initialize optimizer
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

if LOAD_FILE:
    # Load the checkpoint
    checkpoint_path = LOAD_MODEL_FILE
    checkpoint = torch.load(checkpoint_path)
    model.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    print(LOAD_MODEL_FILE)

LOAD_FILE= True
```

Step 3: Instantiate Segmentation Metric and Evaluate

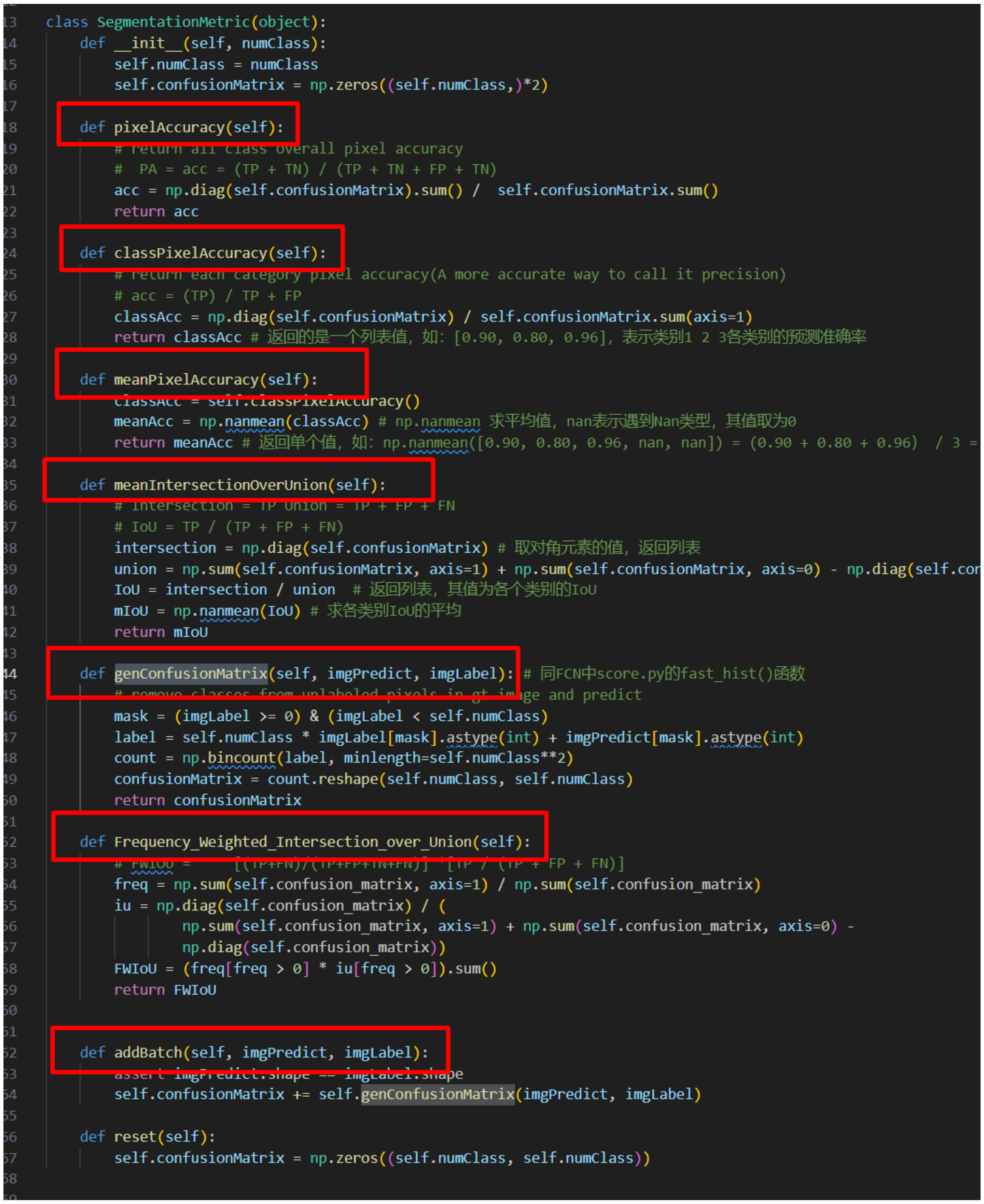
Created an instance of the SegmentationMetric class, configured for the specific number of classes in my segmentation task. I then used the following functions from the class to evaluate the model's performance:

pixelAccuracy(): Calculated the overall pixel accuracy (PA) of the model.

classPixelAccuracy(): Calculated the per-class pixel accuracy (CPA).

meanPixelAccuracy(): Calculated the mean pixel accuracy (MPA) across all classes.

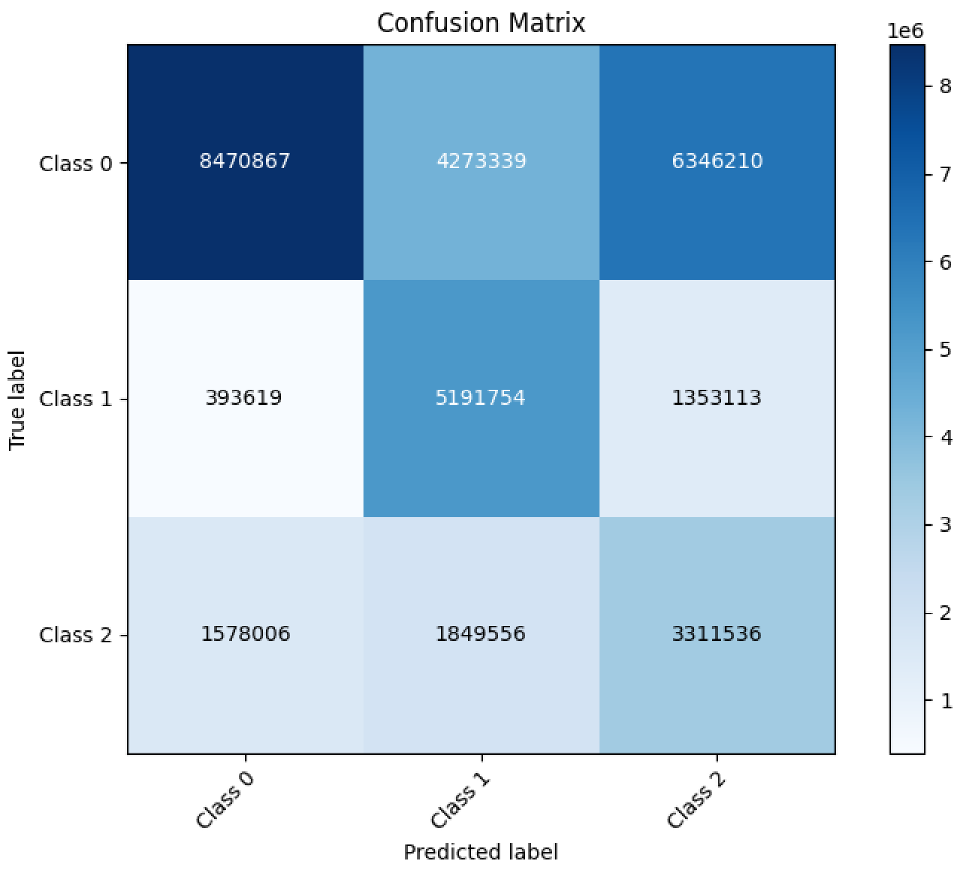
meanIntersectionOverUnion(): Calculated the mean Intersection over Union (mIoU), a widely used segmentation metric.



```python
class SegmentationMetric(object):
    def __init__(self, numClass):
        self.numClass = numClass
        self.confusionMatrix = np.zeros((self.numClass,)*2)

    def pixelAccuracy(self):
        # return all class overall pixel accuracy
        #  PA = acc = (TP + TN) / (TP + TN + FP + TN)
        acc = np.diag(self.confusionMatrix).sum() /  self.confusionMatrix.sum()
        return acc

    def classPixelAccuracy(self):
        # return each category pixel accuracy(A more accurate way to call it precision)
        # acc = (TP) / TP + FP
        classAcc = np.diag(self.confusionMatrix) / self.confusionMatrix.sum(axis=1)
        return classAcc # 返回的是一个列表值，如：[0.90, 0.80, 0.96]，表示类别1 2 3各类别的预测准确率

    def meanPixelAccuracy(self):
        classAcc = self.classPixelAccuracy()
        meanAcc = np.nanmean(classAcc) # np.nanmean 求平均值，nan表示遇到Nan类型，其值取为0
        return meanAcc # 返回单个值，如：np.nanmean([0.90, 0.80, 0.96, nan, nan]) = (0.90 + 0.80 + 0.96)  / 3 =

    def meanIntersectionOverUnion(self):
        # Intersection = TP Union = TP + FP + FN
        # IoU = TP / (TP + FP + FN)
        intersection = np.diag(self.confusionMatrix) # 取对角元素的值，返回列表
        union = np.sum(self.confusionMatrix, axis=1) + np.sum(self.confusionMatrix, axis=0) - np.diag(self.con
        IoU = intersection / union  # 返回列表，其值为各个类别的IoU
        mIoU = np.nanmean(IoU) # 求各类别IoU的平均
        return mIoU

    def genConfusionMatrix(self, imgPredict, imgLabel): # 同FCN中score.py的fast_hist()函数
        # remove classes from unlabeled pixels in gt image and predict
        mask = (imgLabel >= 0) & (imgLabel < self.numClass)
        label = self.numClass * imgLabel[mask].astype(int) + imgPredict[mask].astype(int)
        count = np.bincount(label, minlength=self.numClass**2)
        confusionMatrix = count.reshape(self.numClass, self.numClass)
        return confusionMatrix

    def Frequency_Weighted_Intersection_over_Union(self):
        # FWIOU =     [(TP+FN)/(TP+FP+TN+FN)] *[TP / (TP + FP + FN)]
        freq = np.sum(self.confusion_matrix, axis=1) / np.sum(self.confusion_matrix)
        iu = np.diag(self.confusion_matrix) / (
                np.sum(self.confusion_matrix, axis=1) + np.sum(self.confusion_matrix, axis=0) -
                np.diag(self.confusion_matrix))
        FWIoU = (freq[freq > 0] * iu[freq > 0]).sum()
        return FWIoU


    def addBatch(self, imgPredict, imgLabel):
        assert imgPredict.shape == imgLabel.shape
        self.confusionMatrix += self.genConfusionMatrix(imgPredict, imgLabel)

    def reset(self):
        self.confusionMatrix = np.zeros((self.numClass, self.numClass))
```

Step 4: Generate Confusion Matrix

Utilized the genConfusionMatrix() function from the SegmentationMetric class to generate a confusion matrix. This matrix provides a detailed breakdown of the model's predictions by pixel



Step 5: Analyze Results

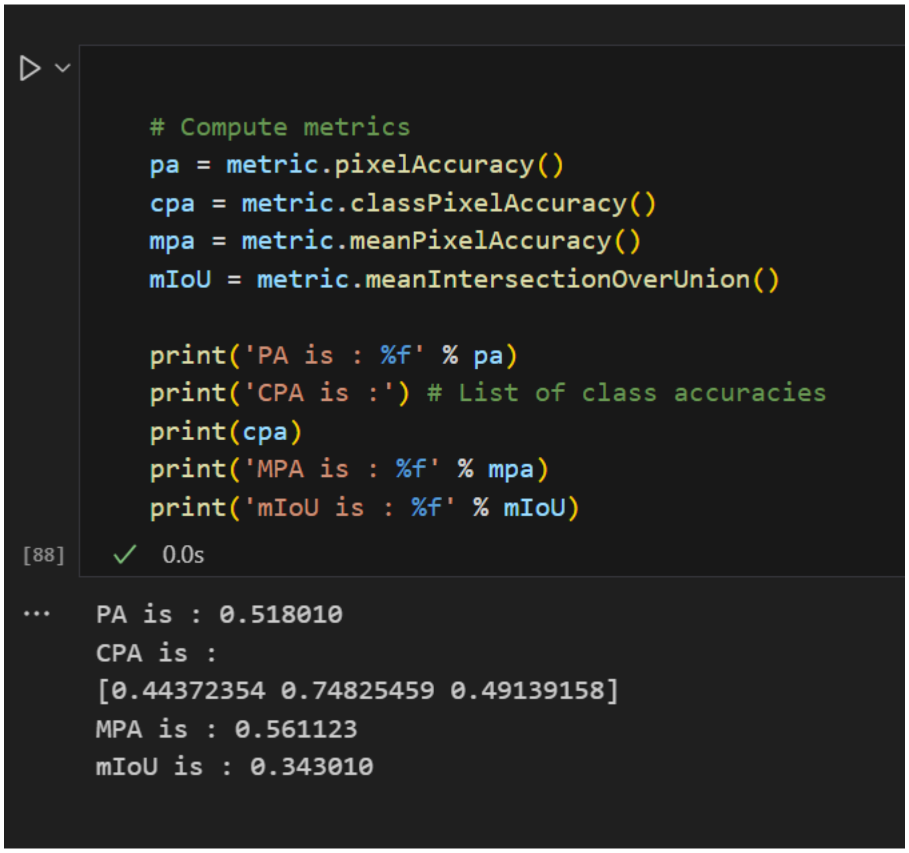
The evaluation process yielded the following results:

PA: 0.518010

CPA: [0.44372354 0.74825459 0.49139158]

MPA: 0.561123

mIoU: 0.343010

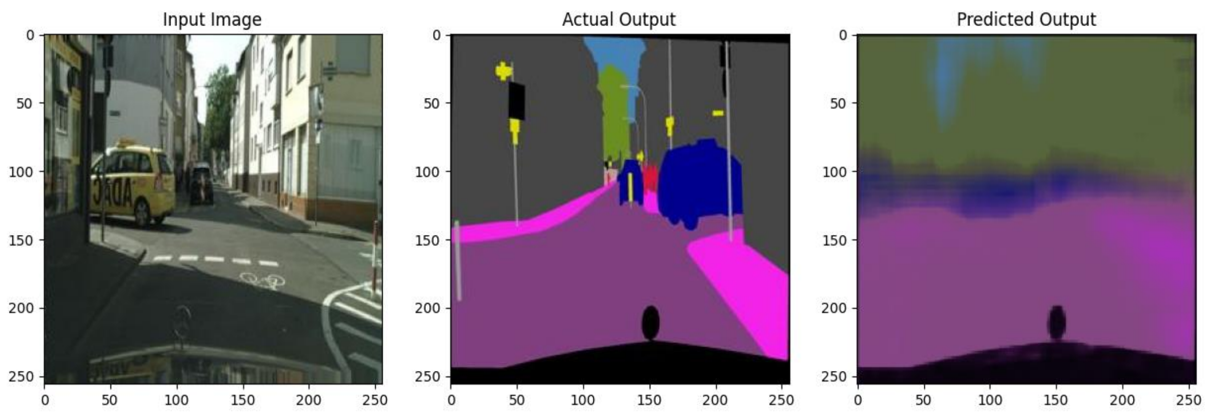


```python
# Compute metrics
pa = metric.pixelAccuracy()
cpa = metric.classPixelAccuracy()
mpa = metric.meanPixelAccuracy()
mIoU = metric.meanIntersectionOverUnion()

print('PA is : %f' % pa)
print('CPA is :') # List of class accuracies
print(cpa)
print('MPA is : %f' % mpa)
print('mIoU is : %f' % mIoU)
```

[88] ✓ 0.0s

```
PA is : 0.518010
CPA is :
[0.44372354 0.74825459 0.49139158]
MPA is : 0.561123
mIoU is : 0.343010
```

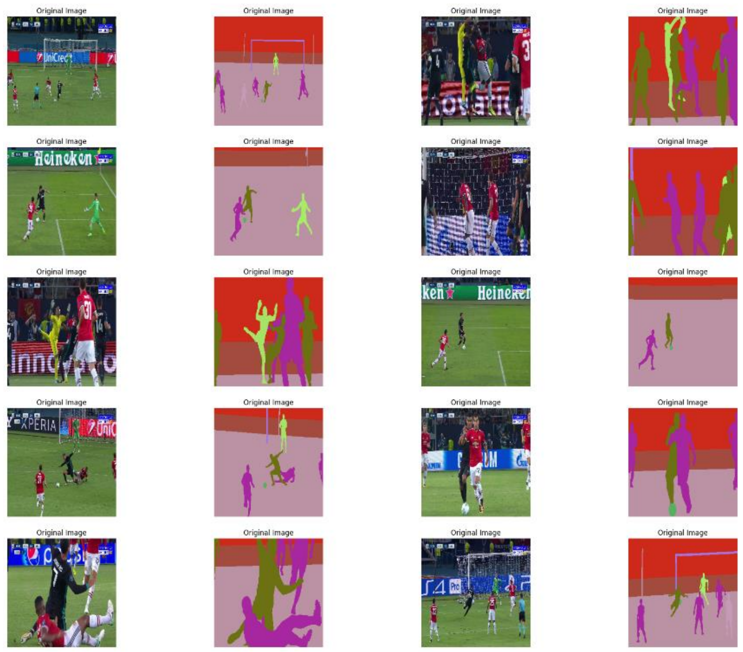
These metrics, along with the confusion matrix, provide a comprehensive assessment of the unitV1 model's segmentation performance. I used these insights to analyze the model's strengths and weaknesses and identify areas for potential improvement.



Exercise 3: Semantic Segmentation of Football

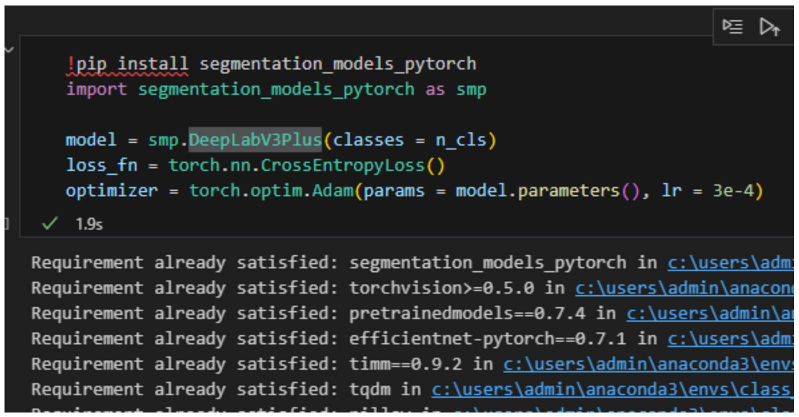
Step 1: Prepare the Dataset

Acquired a dataset of football images with corresponding segmentation masks. Each mask delineated the regions within the image corresponding to the "football" class.



Step 2: Import DeepLabV3+ Model

Utilized the powerful DeepLabV3+ model architecture, a state-of-the-art semantic segmentation model, from the PyTorch Hub. This provided a pre-trained model, allowing for faster training and potentially better performance.



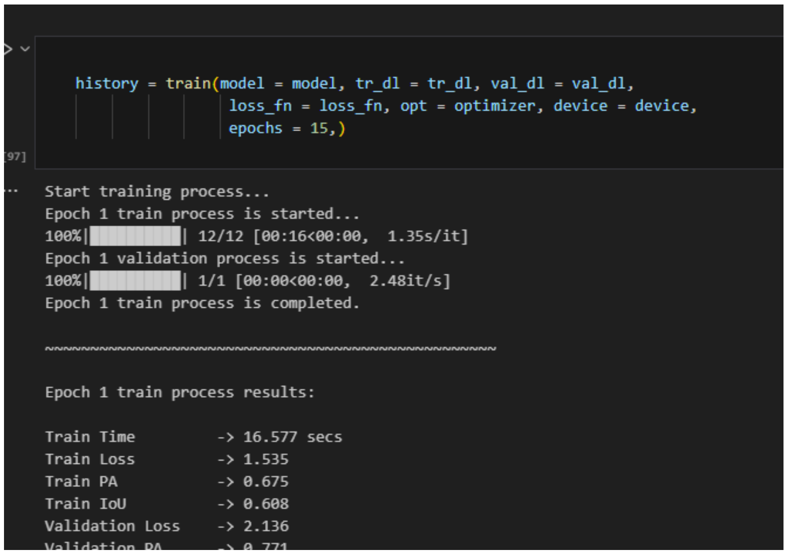
```
!pip install segmentation_models_pytorch
import segmentation_models_pytorch as smp

model = smp.DeepLabV3Plus(classes = n_cls)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params = model.parameters(), lr = 3e-4)
```

✓ 1.9s

```
Requirement already satisfied: segmentation_models_pytorch in c:\users\adm
Requirement already satisfied: torchvision>=0.5.0 in c:\users\admin\anacon
Requirement already satisfied: pretrainedmodels==0.7.4 in c:\users\admin\a
Requirement already satisfied: efficientnet-pytorch==0.7.1 in c:\users\adm
Requirement already satisfied: timm==0.9.2 in c:\users\admin\anaconda3\env
Requirement already satisfied: tqdm in c:\users\admin\anaconda3\envs\class
```

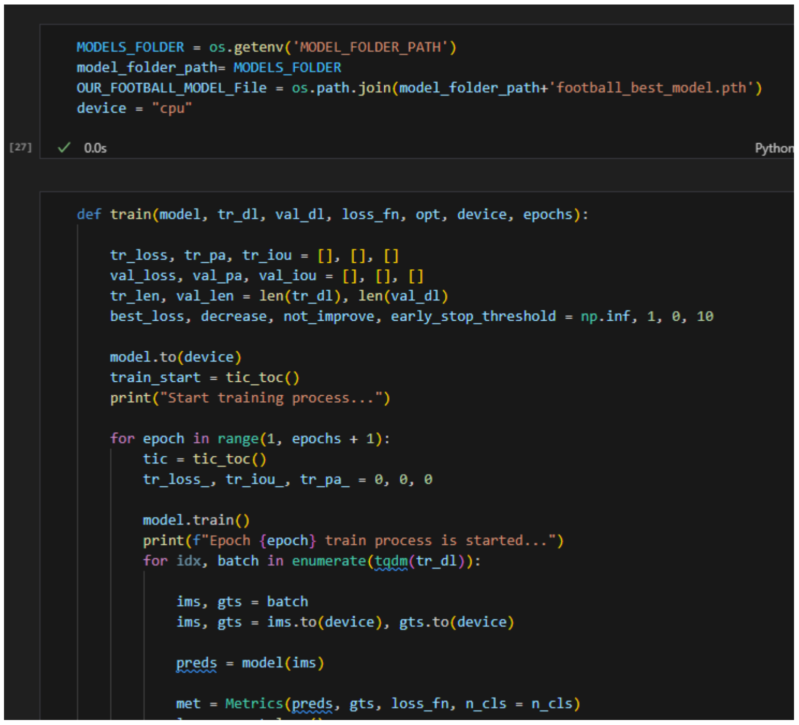
Step 3: Train the Model

Fine-tuned the DeepLabV3+ model on the football image dataset. During training, monitored the model's performance using metrics like Intersection over Union (IoU), Pixel Accuracy (PA), and loss. Visualized the training progress by plotting the learning curves for these metrics (iou_learning_curve, pa_learning_curve, loss_learning_curve).



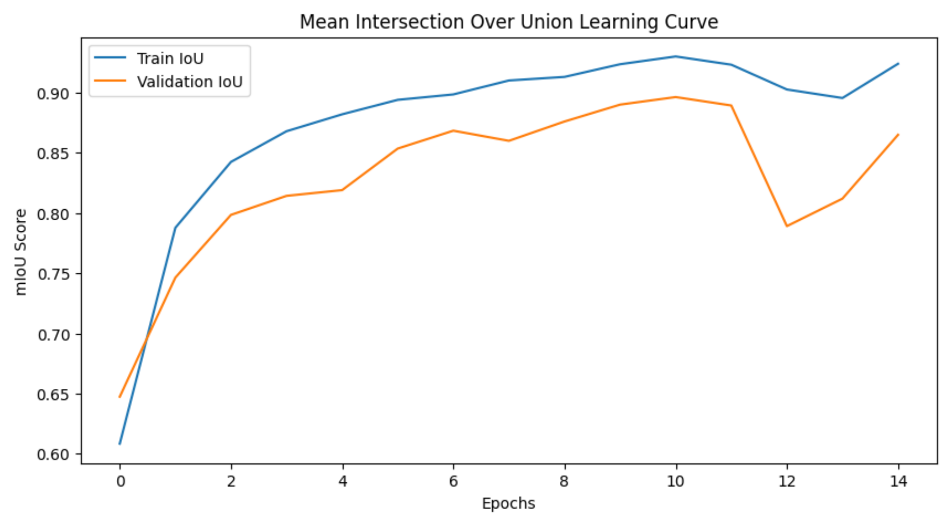
```python
history = train(model = model, tr_dl = tr_dl, val_dl = val_dl,
                loss_fn = loss_fn, opt = optimizer, device = device,
                epochs = 15,)
```

```
Start training process...
Epoch 1 train process is started...
100%|██████████| 12/12 [00:16<00:00,  1.35s/it]
Epoch 1 validation process is started...
100%|██████████| 1/1 [00:00<00:00,  2.48it/s]
Epoch 1 train process is completed.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Epoch 1 train process results:

Train Time         -> 16.577 secs
Train Loss         -> 1.535
Train PA           -> 0.675
Train IoU          -> 0.608
Validation Loss    -> 2.136
Validation PA      -> 0.771
```

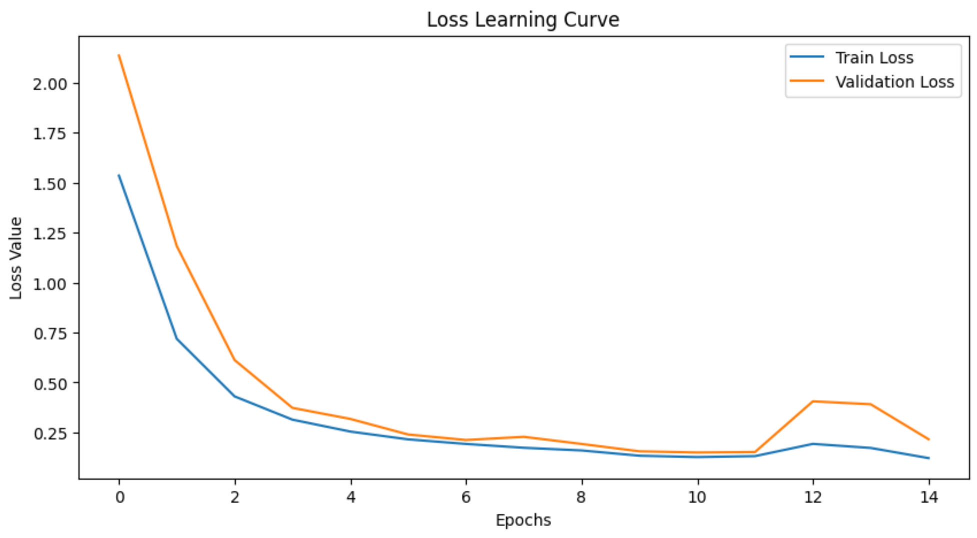


```python
MODELS_FOLDER = os.getenv('MODEL_FOLDER_PATH')
model_folder_path= MODELS_FOLDER
OUR_FOOTBALL_MODEL_File = os.path.join(model_folder_path+'football_best_model.pth')
device = "cpu"
```

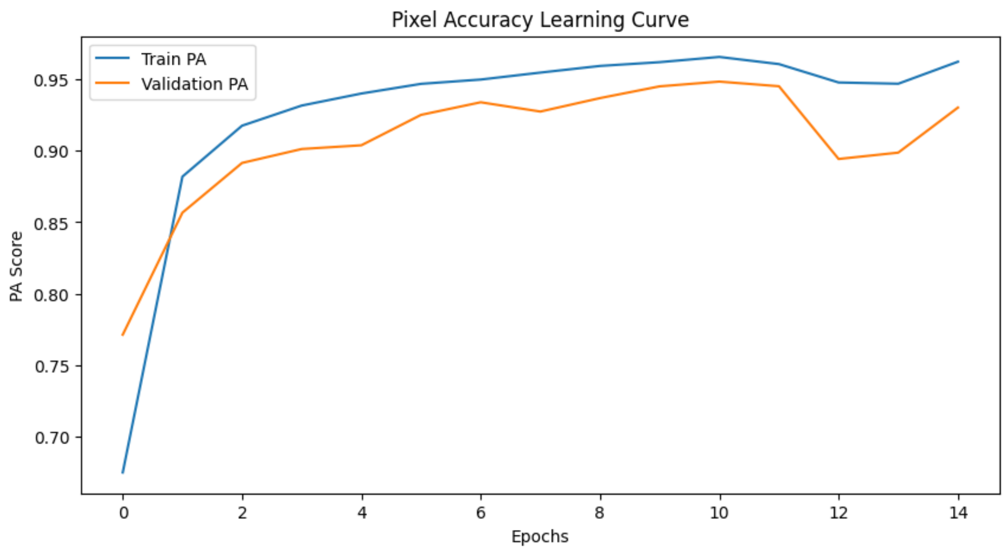
```python
def train(model, tr_dl, val_dl, loss_fn, opt, device, epochs):

    tr_loss, tr_pa, tr_iou = [], [], []
    val_loss, val_pa, val_iou = [], [], []
    tr_len, val_len = len(tr_dl), len(val_dl)
    best_loss, decrease, not_improve, early_stop_threshold = np.inf, 1, 0, 10

    model.to(device)
    train_start = tic_toc()
    print("Start training process...")

    for epoch in range(1, epochs + 1):
        tic = tic_toc()
        tr_loss_, tr_iou_, tr_pa_ = 0, 0, 0

        model.train()
        print(f"Epoch {epoch} train process is started...")
        for idx, batch in enumerate(tqdm(tr_dl)):

            ims, gts = batch
            ims, gts = ims.to(device), gts.to(device)

            preds = model(ims)

            met = Metrics(preds, gts, loss_fn, n_cls = n_cls)
```



Mean Intersection Over Union Learning Curve
Train IoU
Validation IoU
mIoU Score
Epochs



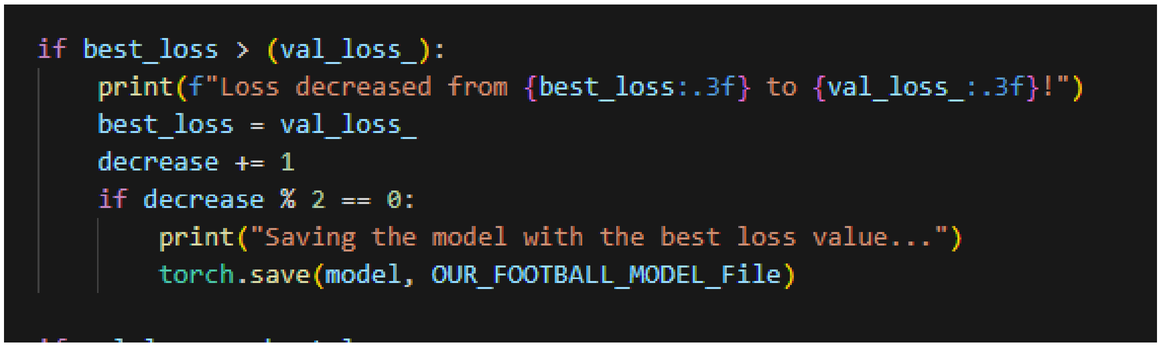
Loss Learning Curve
Train Loss
Validation Loss
Loss Value
Epochs



Pixel Accuracy Learning Curve
Train PA
Validation PA
PA Score
Epochs

Step 4: Save the Trained Model

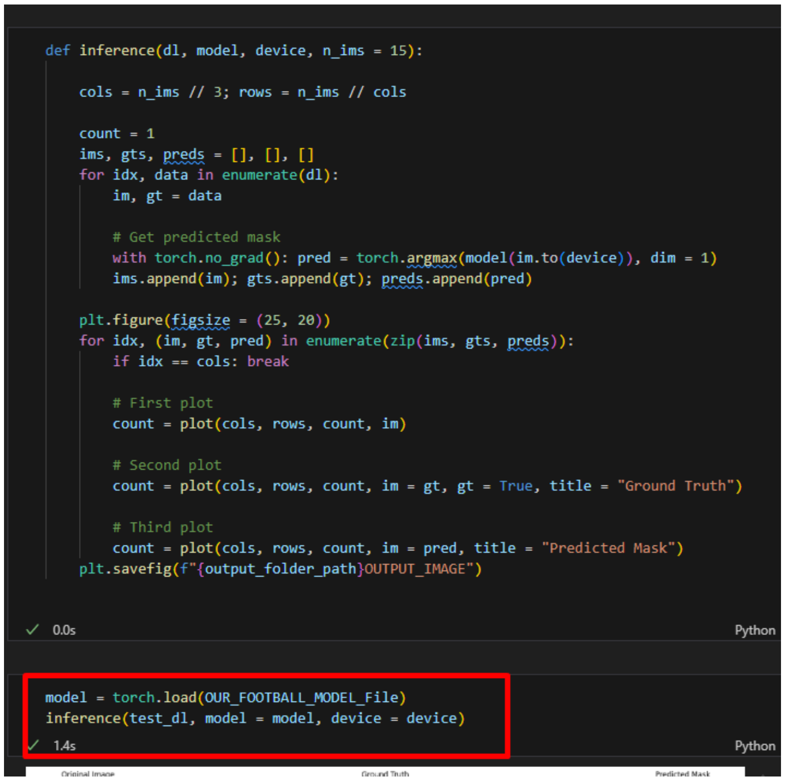
Saved the trained DeepLabV3+ model weights for later use. This preserved the learned parameters and allowed for the model to be loaded and used without retraining.



```python
if best_loss > (val_loss_):
    print(f"Loss decreased from {best_loss:.3f} to {val_loss_:.3f}!")
    best_loss = val_loss_
    decrease += 1
    if decrease % 2 == 0:
        print("Saving the model with the best loss value...")
        torch.save(model, OUR_FOOTBALL_MODEL_File)
```

Step 5: Load and Predict

Loaded the saved model weights and used the model to predict segmentation masks on new football images. The model effectively identified and segmented the "football" regions within these images, demonstrating its ability to generalize to unseen data.



```python
def inference(dl, model, device, n_ims = 15):

    cols = n_ims // 3; rows = n_ims // cols

    count = 1
    ims, gts, preds = [], [], []
    for idx, data in enumerate(dl):
        im, gt = data

        # Get predicted mask
        with torch.no_grad(): pred = torch.argmax(model(im.to(device)), dim = 1)
        ims.append(im); gts.append(gt); preds.append(pred)

    plt.figure(figsize = (25, 20))
    for idx, (im, gt, pred) in enumerate(zip(ims, gts, preds)):
        if idx == cols: break

        # First plot
        count = plot(cols, rows, count, im)

        # Second plot
        count = plot(cols, rows, count, im = gt, gt = True, title = "Ground Truth")

        # Third plot
        count = plot(cols, rows, count, im = pred, title = "Predicted Mask")
    plt.savefig(f"{output_folder_path}OUTPUT_IMAGE")
```

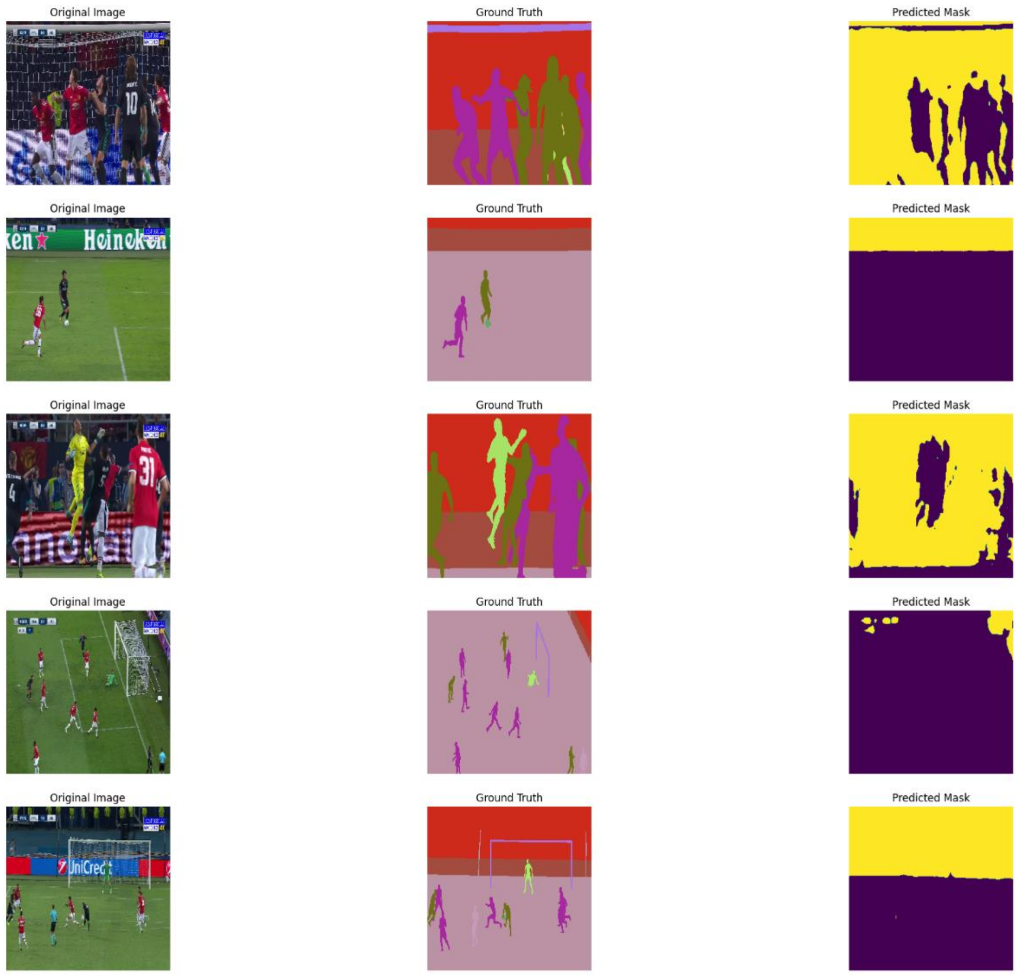
```
✓ 0.0s                                                    Python
```

```python
model = torch.load(OUR_FOOTBALL_MODEL_File)
inference(test_dl, model = model, device = device)
```

```
✓ 1.4s                                                    Python
```

Step 6: Analyze Results

Visually compared the predicted segmentation masks with the ground truth masks. This analysis revealed the model's effectiveness in accurately segmenting footballs within various image contexts.



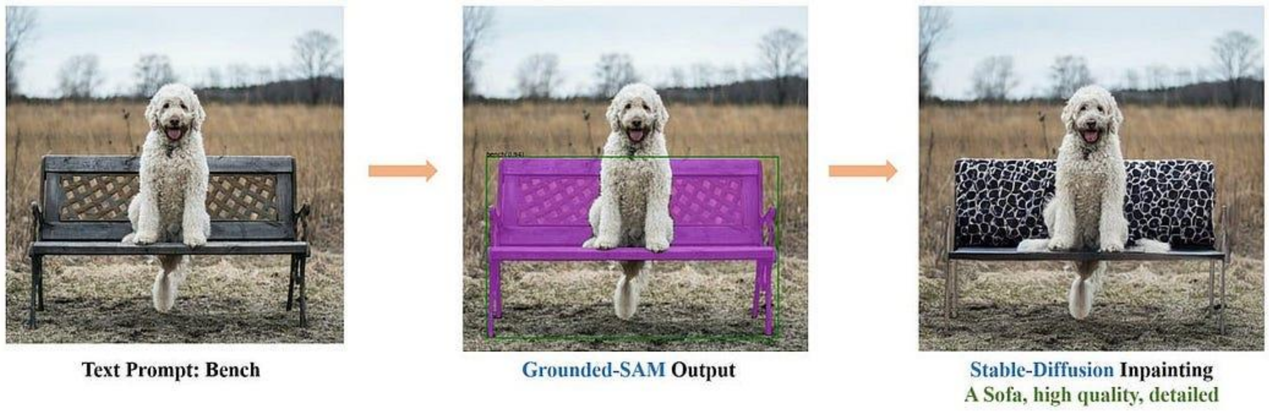
Through this process, successfully implemented a semantic segmentation pipeline for identifying footballs in images using the DeepLabV3+ model, highlighting the model's capabilities in accurately segmenting objects of interest within complex scenes.

Exercise 4: Using the SAM Model for Image Segmentation

In simple words SAM is a foundational model in computer vision. Below are the examples when you ask the model to provide a mask for everything.

original and masked image

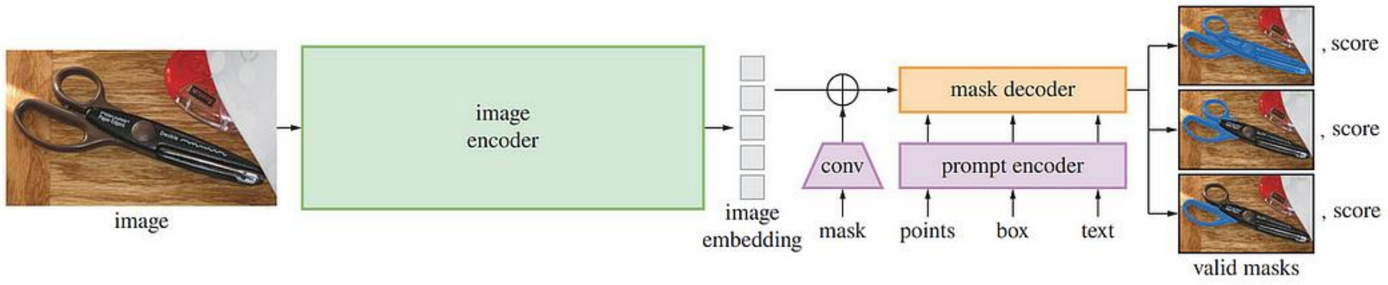
Thanks to open source people are now using it in numerous ways, one which stands out for me is shown in the above image. Write a text prompt and use image generation models to inpaint it.



Modified version of SAM being used along with stable diffusion for inpainting.

In the following sections, I will delve into the architecture of the model, the process of creating the dataset, a bit on process of building the dataset and a bit on its zero-shot capabilities.

Architecture:



In a nutshell model consists of:

1. Image encoder (masked autoencoder) to extract the image embedding,
2. Prompt encoder that takes in different types of prompt and
3. Mask decoder to build a mask.

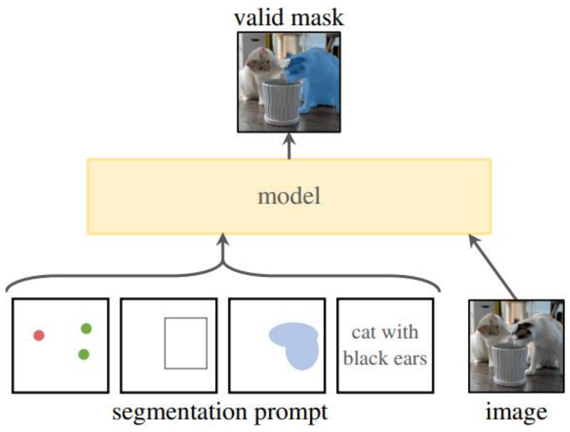
Large Language Models (LLMs) pre-trained on extensive text data from the web, are revolutionizing **NLP** with their impressive zero-shot and few-shot generalization. These models are usually referred to as "**foundational models**". However, in the computer vision domain, there has been limited exploration in developing foundational models. CLIP and ALIGN are among the few existing models in this space. So the main motivation for this paper was to develop a foundational **model** for image segmentation, a model that can take a prompt of different styles and which has powerful generalization abilities.

Image Encoder:

The core of the model is a **Masked autoencoder** which utilizes a vision transformer to achieve high scalability. They employ a ViT-H/16 which is a huge vision transformer model that handles a 16x16 patch size. It features a 14×14 windowed attention and four equally-spaced global attention blocks.

The output from the above encoder is feature embedding that is a 16x downscaled version of the original image. This downsizing process is crucial for efficient processing while retaining essential image features. The model takes an input resolution of 1024x1024x3, typical for high-resolution images, and transforms it into a dense embedding of size 64x64x256.

Prompt Encoder:



Prompt encoder

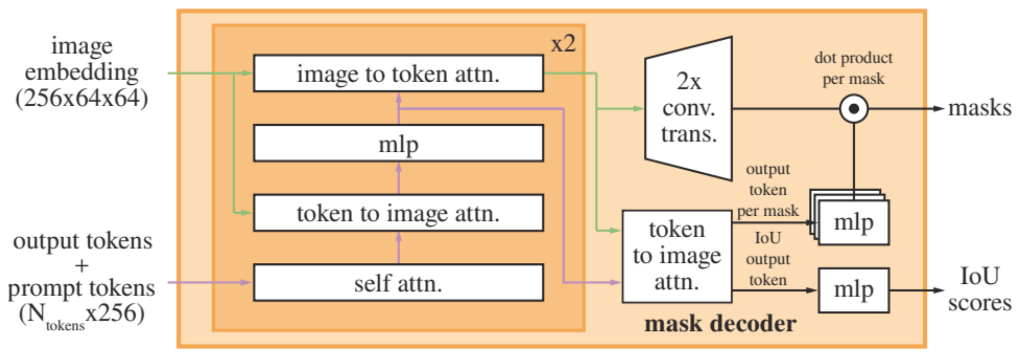
Different types of prompts namely: points, boxes, masks and text. There are two types of prompts:

1. **sparse** which include **points**, **boxes** and **text** and

2. **dense** which includes a masks

Point is represented as a sum of positional encoding of the points's location and one of the two learned embeddings to indicate either a **foreground point** or **background point**. Boxes are represented by an **embedding pair**. The positional encoding of its top-left corner is summed with a learned embedding representing "top-left corner" and similarly for bottom right corner. For the **text** they use **CLIP**, they don't modify this and therefore any text encoder can be used. **Dense prompts** are masks therefore they have spatial correspondence with the images. These masks are downscaled by a factor of 4 before inputting them into the model. Inside, they are again downscaled by the factor of 4, achieved through two 2x2, stride-2 convolutions, with the output channels as 4 and 16. Subsequently a final 1x1 convolution is applied that maps these channels to 256 channels. Each layer is enhanced with Gelu activation and layer normalization. The mask embedding is then added element-wise to the image embedding. In cases where no mask prompt is provided, a learned embedding, representing 'no mask,' is added to each image embedding location.

Light Weight Mask Decoder:



output mask decoder

This is the place where all the magic happens, here image embeddings and prompt embeddings are mapped to the final mask. To build this they take some inspiration from Transformer segmentation models and accordingly modify the Transformer decoder.

A key aspect of this process involves the introduction of a learned output token embedding into the prompt embedding before it is processed by the decoder. This output token embedding plays a pivotal role in the decoder's function, containing essential information required for the overall image segmentation task. This concept is similar to the use of class tokens in Vision Transformers for image classification. In image classification, these class tokens are crucial as they encapsulate information about the overall image. Similarly, in our model, the output token embedding serves as a critical element that guides the decoding process towards effective image segmentation.

Each decoder layer performs 4 steps (as can be seem from the figure above):

1) self-attention on the tokens,
2) cross-attention from tokens (as queries) to the image embedding,
3) a point-wise MLP updates each token, and
4) cross-attention from the image embedding (as queries) to tokens.

This last step updates the image embedding with prompt information. During cross-attention, the image embedding is treated as a set of 64^2 256-dimensional vectors.

Building the dataset:

Building the dataset for this model was a tough task as they described it in paper. Typical approach to the training foundation model that is taken while training **llms** is by taking the dataset from the internet, but masks are not easily available like who would make a mask of their images? So they come up with this: -

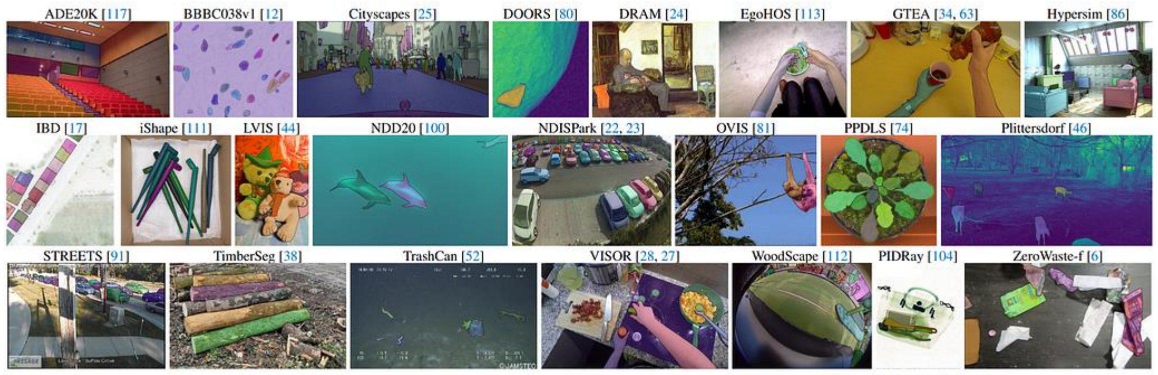
Data Engine: They develop the model with model-in-the-loop dataset annotation. Data engine has three stages: assisted-manual, semi-automatic, and fully automatic.

1. In the first stage, SAM assists annotators in annotating masks, similar to I would say a roboflow setup.
2. In the second phase, SAM has the capability to automatically generate masks for specific objects once prompted with their locations. This allows annotators to focus on creating masks for other objects that SAM cannot handle automatically.
3. At this final stage, SAM is prompted with a regular grid of foreground points. This method efficiently produces approximately 100 high-quality masks per image.

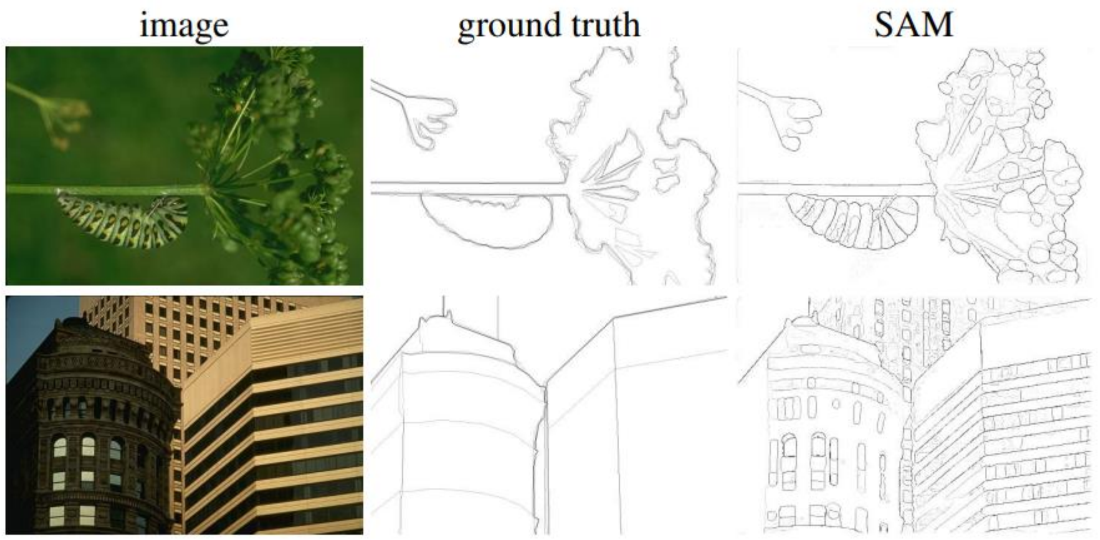
One of the contributions of the paper is **SA-1B** (name for the above dataset) which has over a billion masks for 11 million licensed privacy-preserving images. Final model is the model trained on the masks generated form the third step.

Zero shot transfer results:

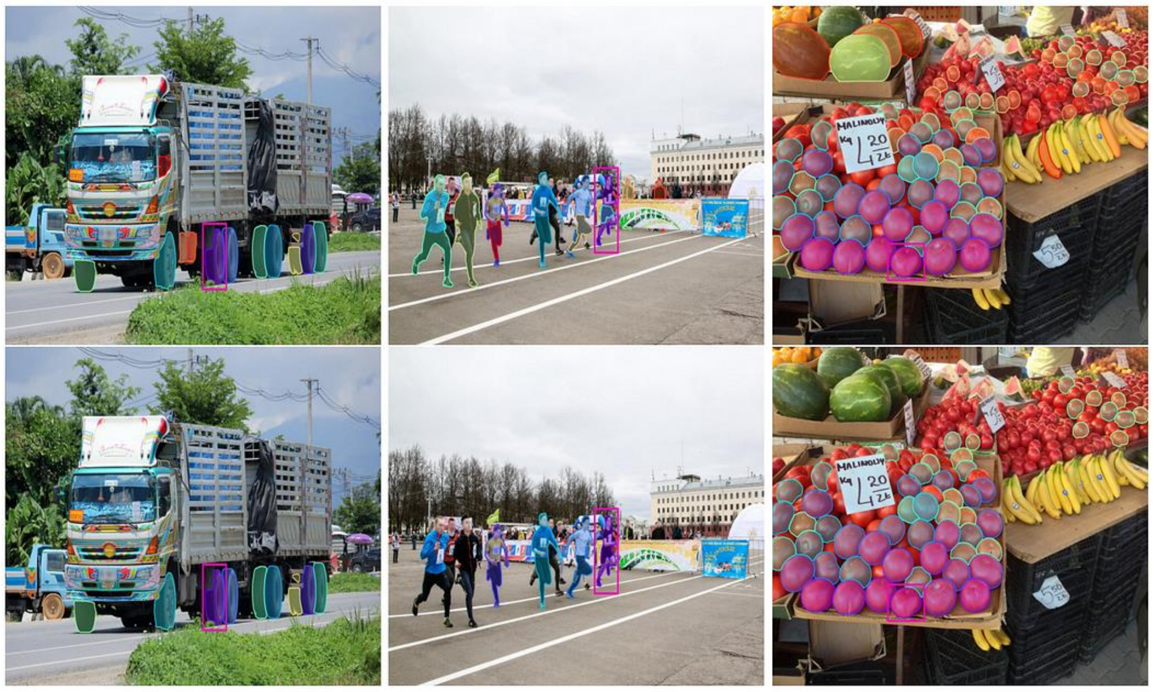
Some examples of zero shot transfers where SAM excels. These are directly taken from the paper.



Samples from the 23 diverse segmentation datasets used to evaluate SAM’s zero-shot transfer capabilities.



Zero-shot edge prediction on BSDS500. SAM was not trained to predict edge maps, nor did it have access to BSDS images or annotations during training.



Visualization of thresholding the similarities of mask embeddings from SAM's latent space. A query is indicated by the magenta box; top row shows matches at a low threshold, bottom row at a high threshold. The most similar mask embeddings in the same image can often be semantically similar to the query mask embedding, even though SAM is not trained with explicit semantic supervision.

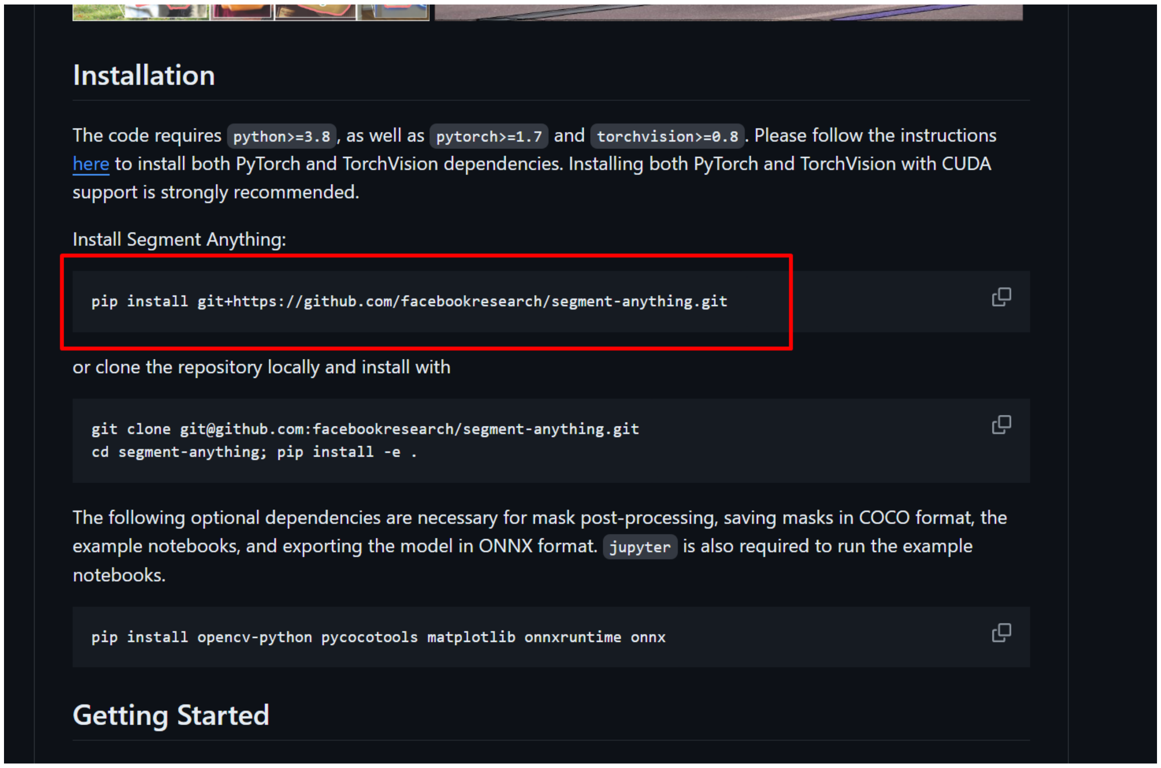
Exercise 5: Installing and Configuring the SAM Environment

Step 1: Install the Segment Anything Model (SAM)

Navigated to the official SAM GitHub repository and followed the installation instructions. Installed SAM using the following pip command:

pip install git+https://github.com/facebookresearch/segment-anything.git

This command fetched the latest code from the repository and installed the necessary dependencies.



Installation

The code requires `python>=3.8`, as well as `pytorch>=1.7` and `torchvision>=0.8`. Please follow the instructions here to install both PyTorch and TorchVision dependencies. Installing both PyTorch and TorchVision with CUDA support is strongly recommended.

Install Segment Anything:

```
pip install git+https://github.com/facebookresearch/segment-anything.git
```

or clone the repository locally and install with

```
git clone git@github.com:facebookresearch/segment-anything.git
cd segment-anything; pip install -e .
```

The following optional dependencies are necessary for mask post-processing, saving masks in COCO format, the example notebooks, and exporting the model in ONNX format. `jupyter` is also required to run the example notebooks.

```
pip install opencv-python pycocotools matplotlib onnxruntime onnx
```

Getting Started

Step 2: Download SAM Model Checkpoints

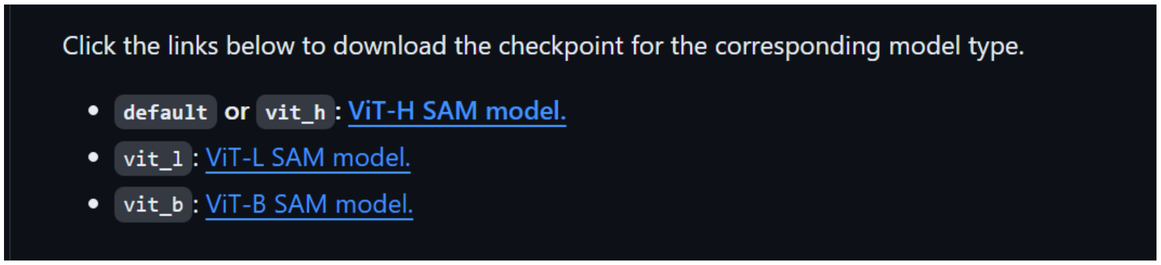
Downloaded the saved checkpoints for the three available SAM model variants:

I. vit_h: ViT-H SAM model (default)

II. vit_l: ViT-L SAM model

III. vit_b: ViT-B SAM model

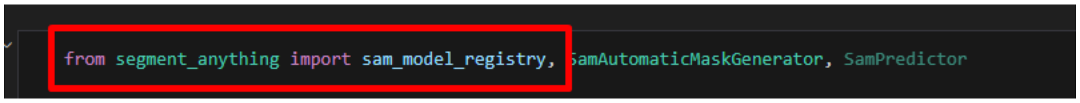
These checkpoints contain the pre-trained model weights and are essential for utilizing SAM's segmentation capabilities.


Click the links below to download the checkpoint for the corresponding model type.

- `default` or `vit_h` : ViT-H SAM model.
- `vit_l` : ViT-L SAM model.
- `vit_b` : ViT-B SAM model.

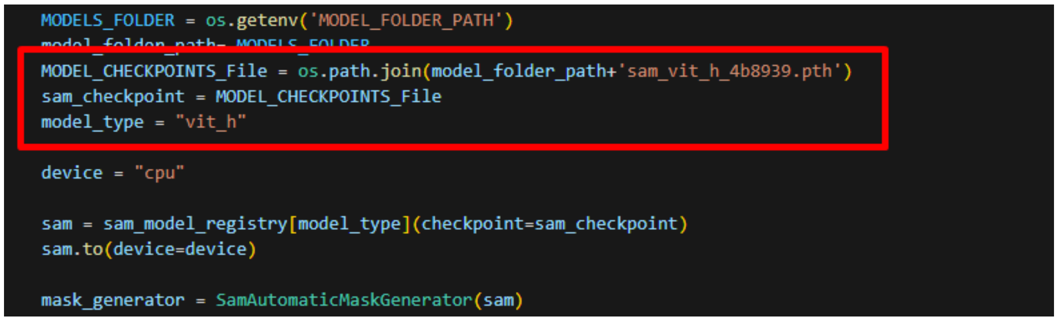
Step 3: Import SAM Functions and Models

In my code editor, imported the necessary functions and classes from the SAM library


```
from segment_anything import sam_model_registry, SamAutomaticMaskGenerator, SamPredictor
```

Step 4: Configure SAM Environment

. For this project, I chose to use the default vit_h model due to its balance of performance and efficiency. After importing the necessary components, I configured the SAM environment by initializing the chosen model variant with its corresponding checkpoint. This step loaded the pre-trained weights into the model, preparing it for segmentation tasks.

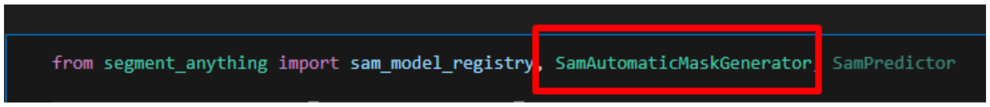

```
MODELS_FOLDER = os.getenv('MODEL_FOLDER_PATH')
model_folder_path= MODELS_FOLDER
MODEL_CHECKPOINTS_File = os.path.join(model_folder_path+'sam_vit_h_4b8939.pth')
sam_checkpoint = MODEL_CHECKPOINTS_File
model_type = "vit_h"

device = "cpu"

sam = sam_model_registry[model_type](checkpoint=sam_checkpoint)
sam.to(device=device)

mask_generator = SamAutomaticMaskGenerator(sam)
```

With these steps completed, I successfully installed and configured the SAM environment, enabling me to utilize SAM's advanced segmentation capabilities for my project.

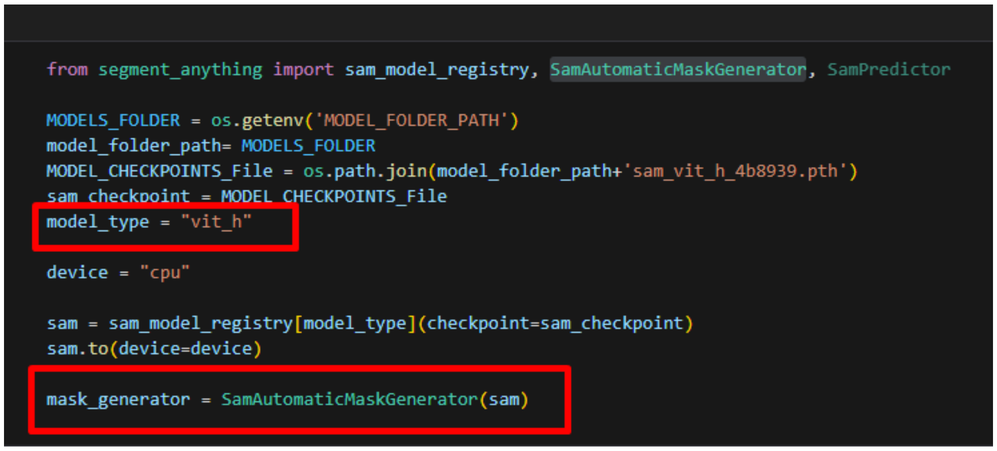
Exercise 6: Using SAM Auto Generate Image Mask

Step 1: Initialize the Mask Generator

I imported the SamAutomaticMaskGenerator class from the SAM library and selected the VIT_H model for mask generation. Using the initialized SAM model (sam), I created an instance of the SamAutomaticMaskGenerator class, named mask_generator.



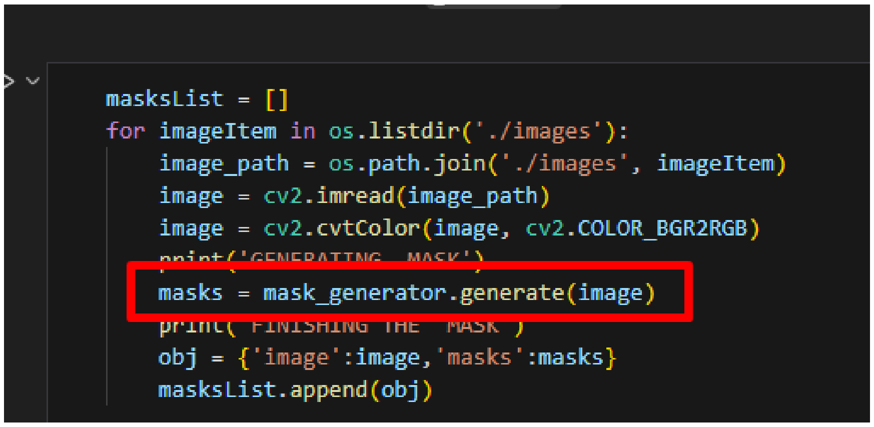
```python
from segment_anything import sam_model_registry, SamAutomaticMaskGenerator, SamPredictor
```



```python
from segment_anything import sam_model_registry, SamAutomaticMaskGenerator, SamPredictor

MODELS_FOLDER = os.getenv('MODEL_FOLDER_PATH')
model_folder_path= MODELS_FOLDER
MODEL_CHECKPOINTS_File = os.path.join(model_folder_path+'sam_vit_h_4b8939.pth')
sam_checkpoint = MODEL_CHECKPOINTS_File
model_type = "vit_h"

device = "cpu"

sam = sam_model_registry[model_type](checkpoint=sam_checkpoint)
sam.to(device=device)

mask_generator = SamAutomaticMaskGenerator(sam)
```

Step 2: Generate Masks

With the mask_generator instance ready, I applied it to my selected images to automatically generate segmentation masks. SAM efficiently produced masks outlining the prominent objects within each image.



```python
masksList = []
for imageItem in os.listdir('./images'):
    image_path = os.path.join('./images', imageItem)
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    print('GENERATING  MASK')
    masks = mask_generator.generate(image)
    print('FINISHING THE  MASK')
    obj = {'image':image,'masks':masks}
    masksList.append(obj)
```

Step 3: Fine-tune Mask Generation Parameters

SAM's SamAutomaticMaskGenerator offers numerous parameters to customize mask generation. I experimented with these parameters to refine the mask quality. To create mask_generator_2, I modified the following:

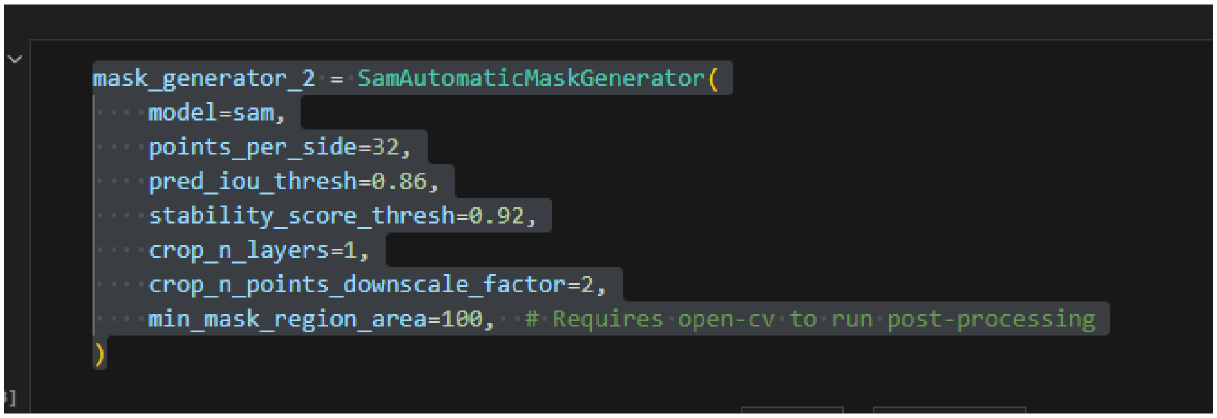
points_per_side: Controls the density of point sampling.

pred_iou_thresh: Sets the threshold for predicted IoU to filter low-quality masks.

stability_score_thresh: Sets the threshold for stability scores to remove unstable masks.

crop_n_layers, crop_n_points_downscale_factor: These parameters affect how image cropping is used to improve performance on smaller objects.

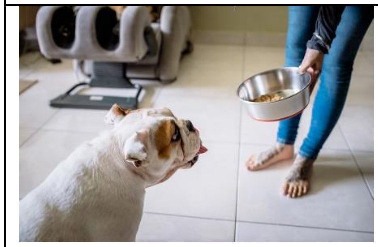
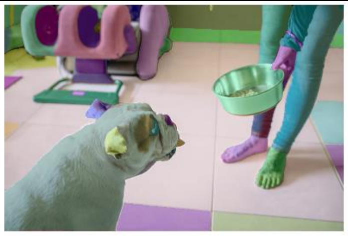
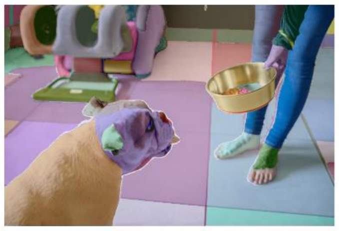
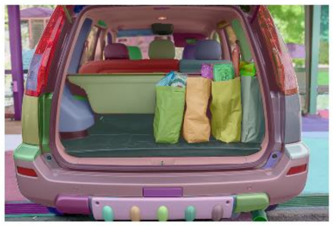
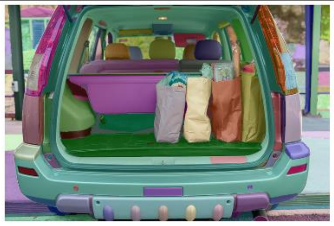
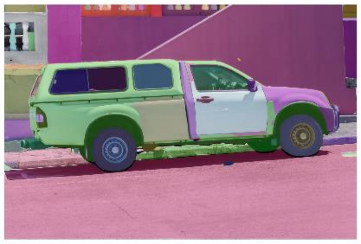
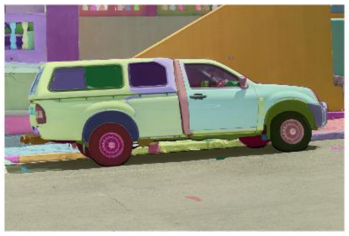
min_mask_region_area: Sets the minimum area for a mask region to be retained (requires OpenCV for post-processing).



```
mask_generator_2 = SamAutomaticMaskGenerator(
    model=sam,
    points_per_side=32,
    pred_iou_thresh=0.86,
    stability_score_thresh=0.92,
    crop_n_layers=1,
    crop_n_points_downscale_factor=2,
    min_mask_region_area=100,  # Requires open-cv to run post-processing
)
```

Step 4: Compare Mask Outputs

Using both mask_generator and mask_generator_2, I generated masks for the same set of images. This allowed me to directly compare the output of both configurations and observe how adjusting the parameters impacted mask quality, granularity, and overall segmentation performance.

ACTUAL IMAGE	MASK 1	MASK 2
		
		
		

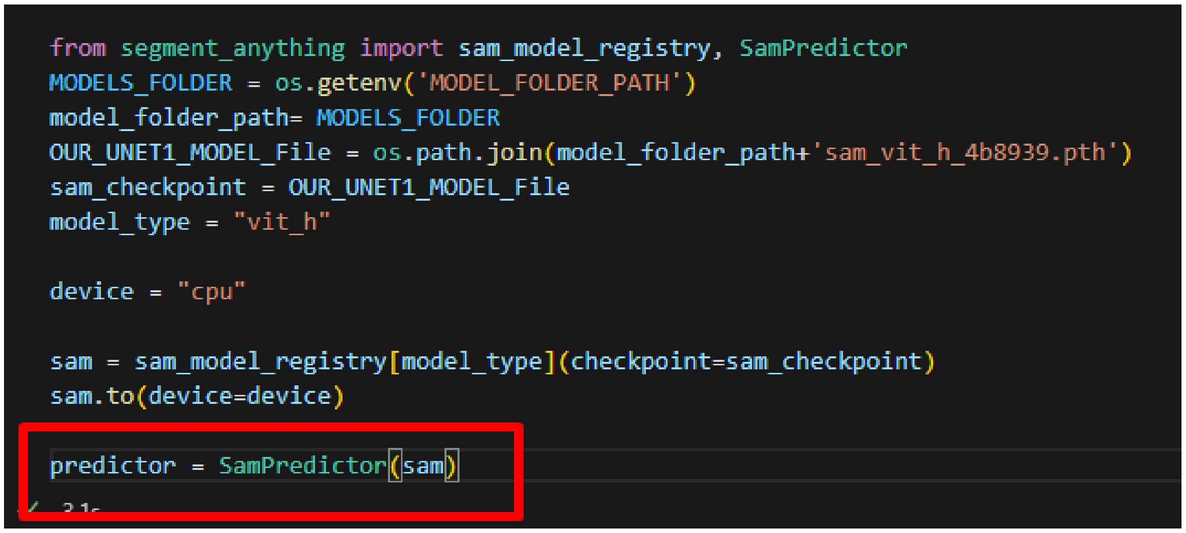
This exploration demonstrated SAM's flexibility in generating automatic masks and the significant influence of parameter tuning on achieving optimal segmentation results.

Exercise 7: Using Prompts to Choose Objects

Step 1: Initialize the SAM Predictor

I started by initializing a SamPredictor object, using my previously loaded SAM model (e.g., vit_h). This predictor object allows for interactive segmentation based on prompts.

```
predictor = SamPredictor(sam)
```

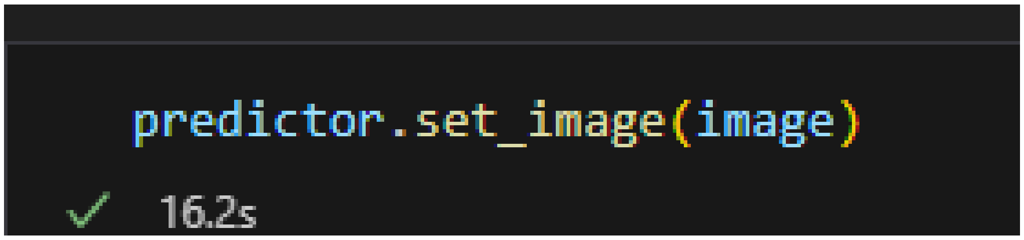


```
from segment_anything import sam_model_registry, SamPredictor
MODELS_FOLDER = os.getenv('MODEL_FOLDER_PATH')
model_folder_path= MODELS_FOLDER
OUR_UNET1_MODEL_File = os.path.join(model_folder_path+'sam_vit_h_4b8939.pth')
sam_checkpoint = OUR_UNET1_MODEL_File
model_type = "vit_h"

device = "cpu"

sam = sam_model_registry[model_type](checkpoint=sam_checkpoint)
sam.to(device=device)

predictor = SamPredictor(sam)
```

Step 2: Set the Image for Segmentation

I then provided the image I wanted to segment to the predictor using the set_image() function:

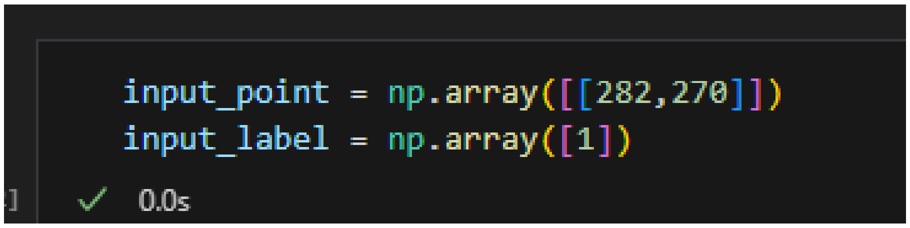
predictor.set_image(image)



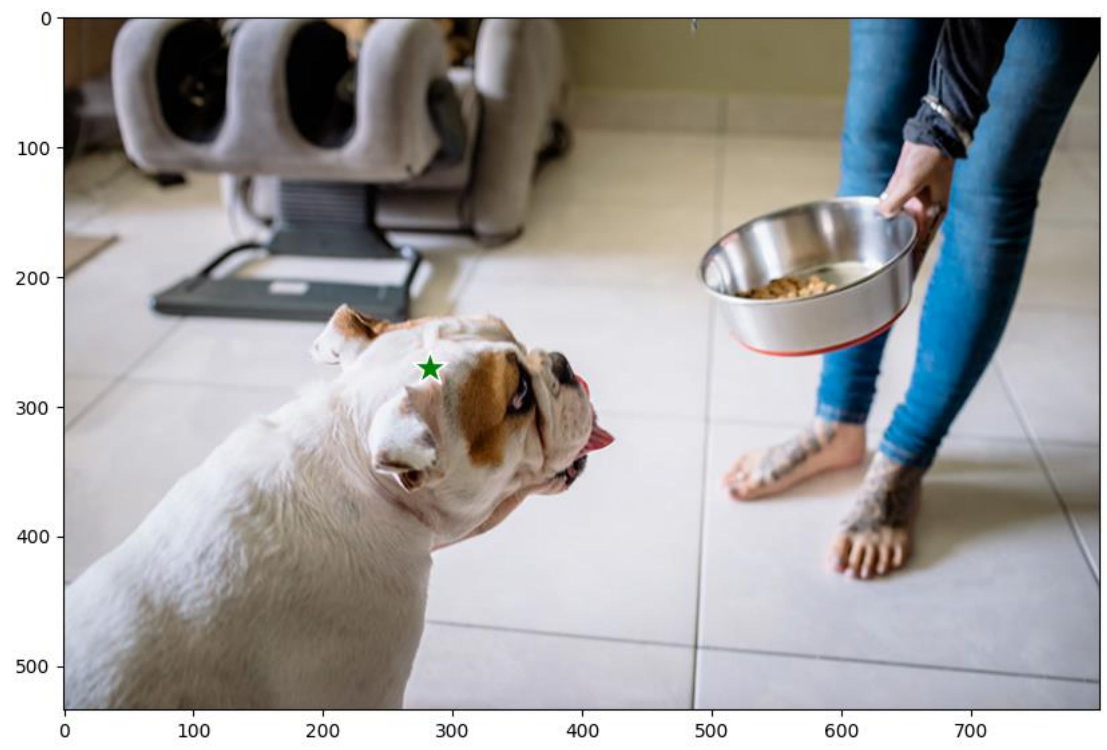
```
predictor.set_image(image)
✓ 16.2s
```

Step 3: Define the Point Prompt

I selected a specific point on the image as a prompt to guide the segmentation. In this case, I chose an image of a dog and placed the prompt point on the dog's head. I defined the point's coordinates (input_point) and a corresponding label (input_label), which typically indicates whether the point should be inside (1) or outside (0) the desired object.



```
input_point = np.array([[282,270]])
input_label = np.array([1])
✓ 0.0s
```



Step 4: Perform the Segmentation

I used the predictor.predict() function to generate segmentation masks. I provided the point coordinates, point label, and set multimask_output=True to obtain multiple potential masks:

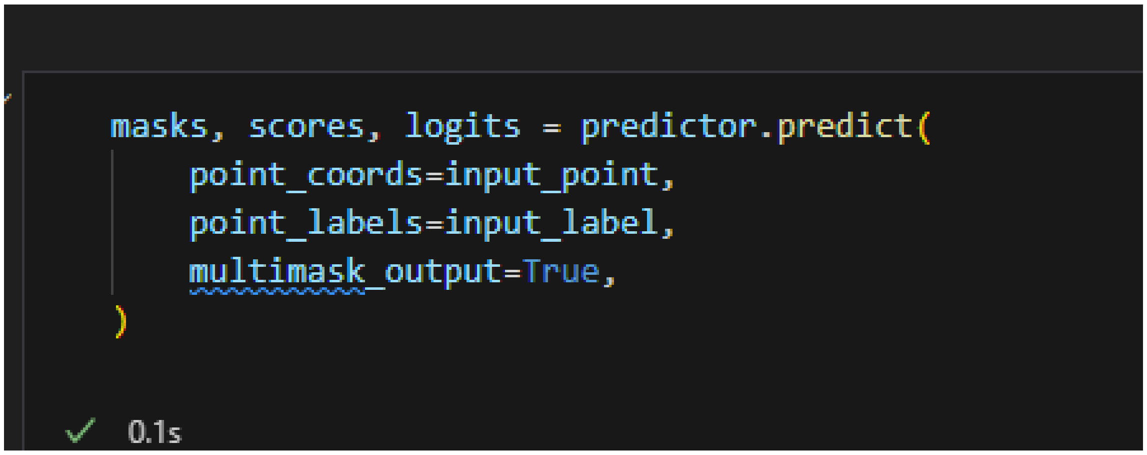
```
masks, scores, logits = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    multimask_output=True,
)
```

This function returned:

masks: An array of predicted segmentation masks.

scores: The confidence scores associated with each mask.

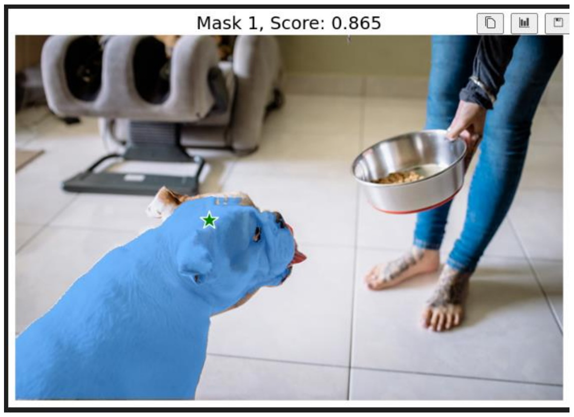
logits: The raw model output before applying the sigmoid function.

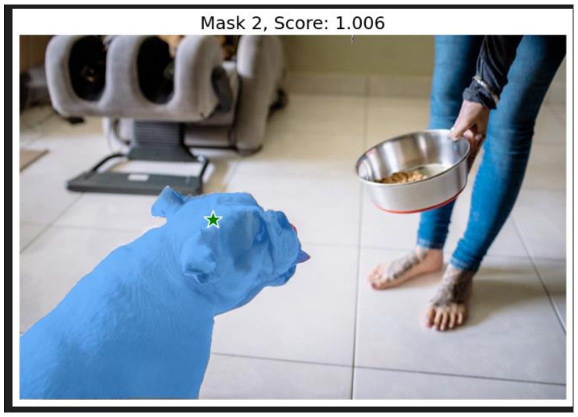


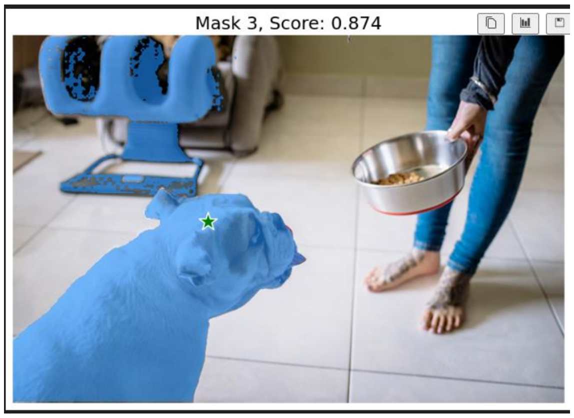
```
masks, scores, logits = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    multimask_output=True,
)
```

```
✓ 0.1s
```

By providing a simple point prompt, I was able to guide SAM to segment the dog in the image effectively. This interactive approach highlights the flexibility and power of SAM in segmenting objects of interest based on user input.



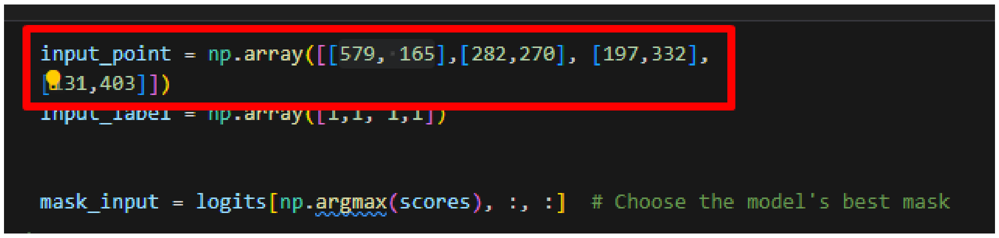




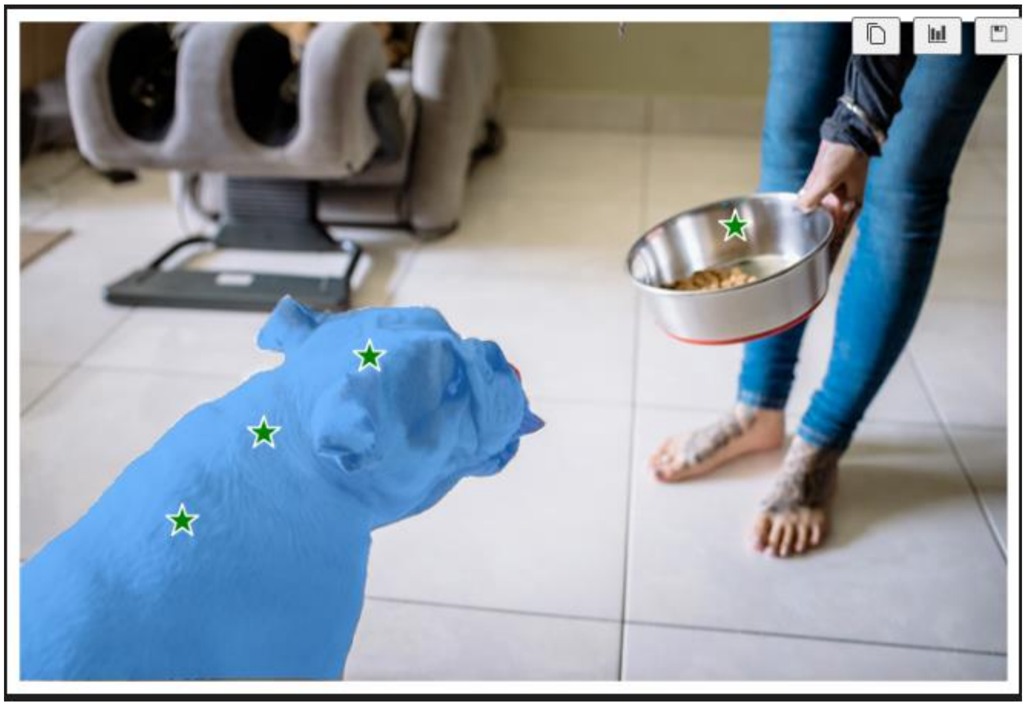
Exercise 8: Specify A Specific Goal Through Additional Points

Step 1: Refining Segmentation with Multiple Prompts

To achieve a more precise segmentation, I decided to provide SAM with additional point prompts. Instead of just one point on the dog's head, I marked four points on the dog's body in the image.

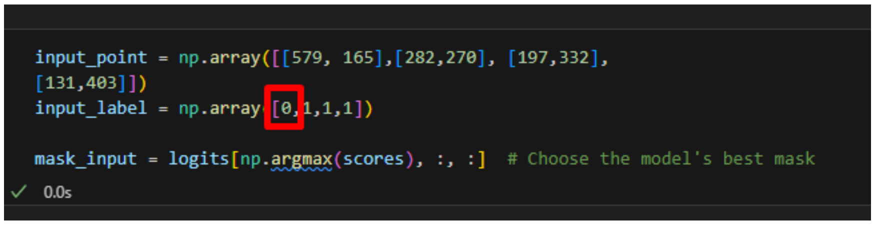


```
input_point = np.array([[579, 165],[282,270], [197,332],
[131,403]])
input_label = np.array([1,1, 1,1])


mask_input = logits[np.argmax(scores), :, :]  # Choose the model's best mask
```



Step 2: Adjusting Point Labels for Accuracy

After running the prediction with four points, I noticed that the fourth point was not accurately placed on the dog. To correct this, I modified the corresponding input_label for that point to 0, indicating that it should lie outside the desired object (the dog).



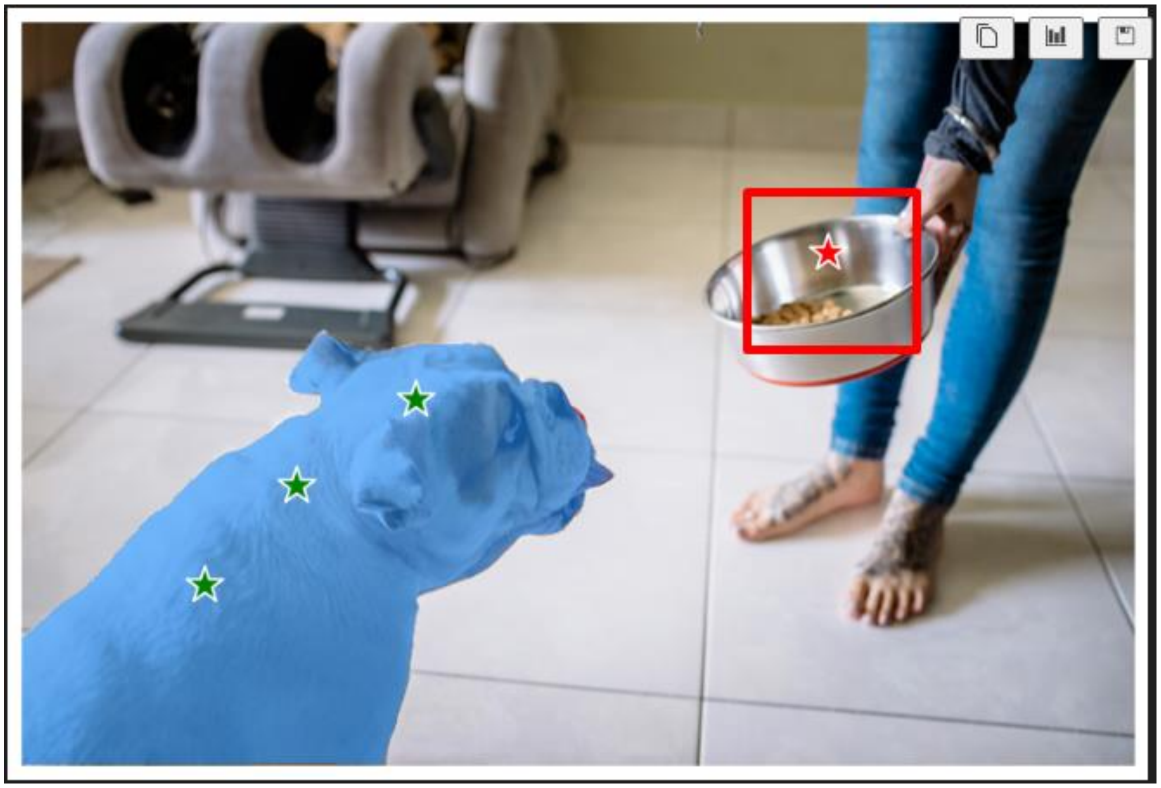
```
input_point = np.array([[579, 165],[282,270], [197,332],
[131,403]])
input_label = np.array([0,1,1,1])

mask_input = logits[np.argmax(scores), :, :]  # Choose the model's best mask
```
✓ 0.0s

Step 3: Predicting with Refined Prompts

Using the updated point coordinates and labels, I re-ran the predictor.predict() function. This time, the segmentation results were significantly more accurate. The mask precisely outlined the dog, and the fourth point, which was incorrectly placed, was now visually marked in red, indicating it was considered "outside" the object.

Outcome:

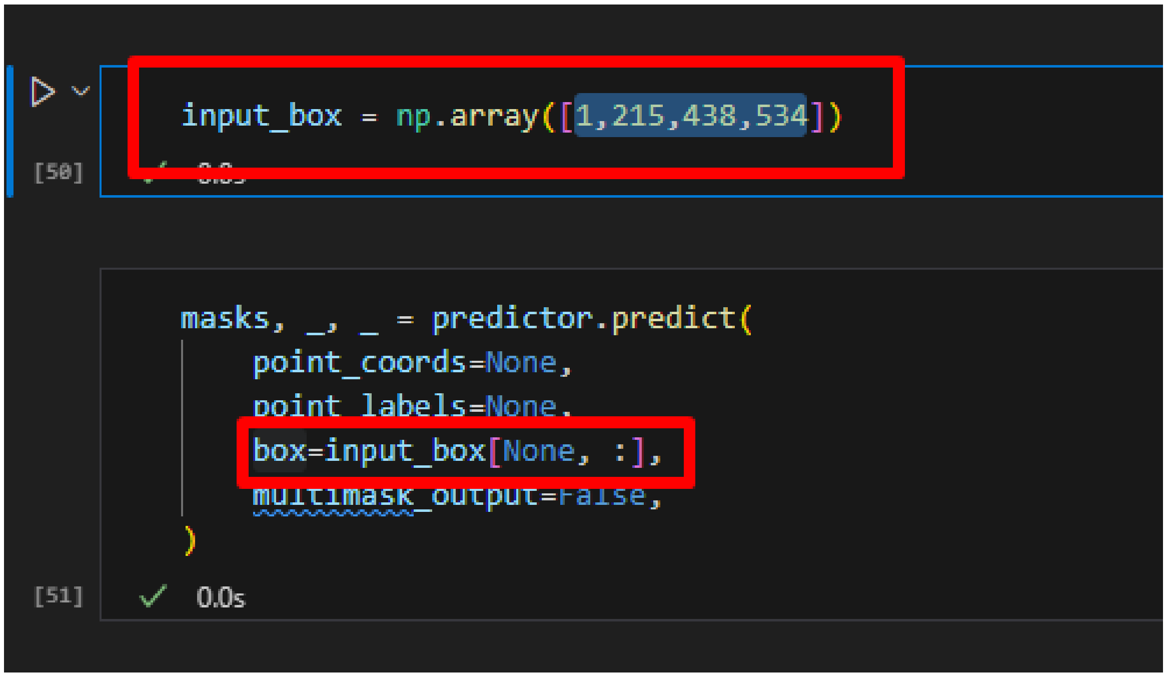
By providing additional prompts and carefully adjusting the point labels, I was able to guide SAM to produce a highly specific and accurate segmentation of the dog in the image. This demonstrates the power of using multiple prompts and label adjustments to achieve precise segmentation results with SAM.



Exercise 9: Specify an Object Through A Box

Step 1: Define the Bounding Box

Instead of using points, I decided to specify the object of interest (the dog) using a bounding box. I created an array, input_box, containing the coordinates of the bounding box in the format: [xmin, ymin, xmax, ymax].



```
input_box = np.array([1,215,438,534])
```

[50] ✓ 0.0s

```
masks, _, _ = predictor.predict(
    point_coords=None,
    point_labels=None,
    box=input_box[None, :],
    multimask_output=False,
)
```

[51] ✓ 0.0s

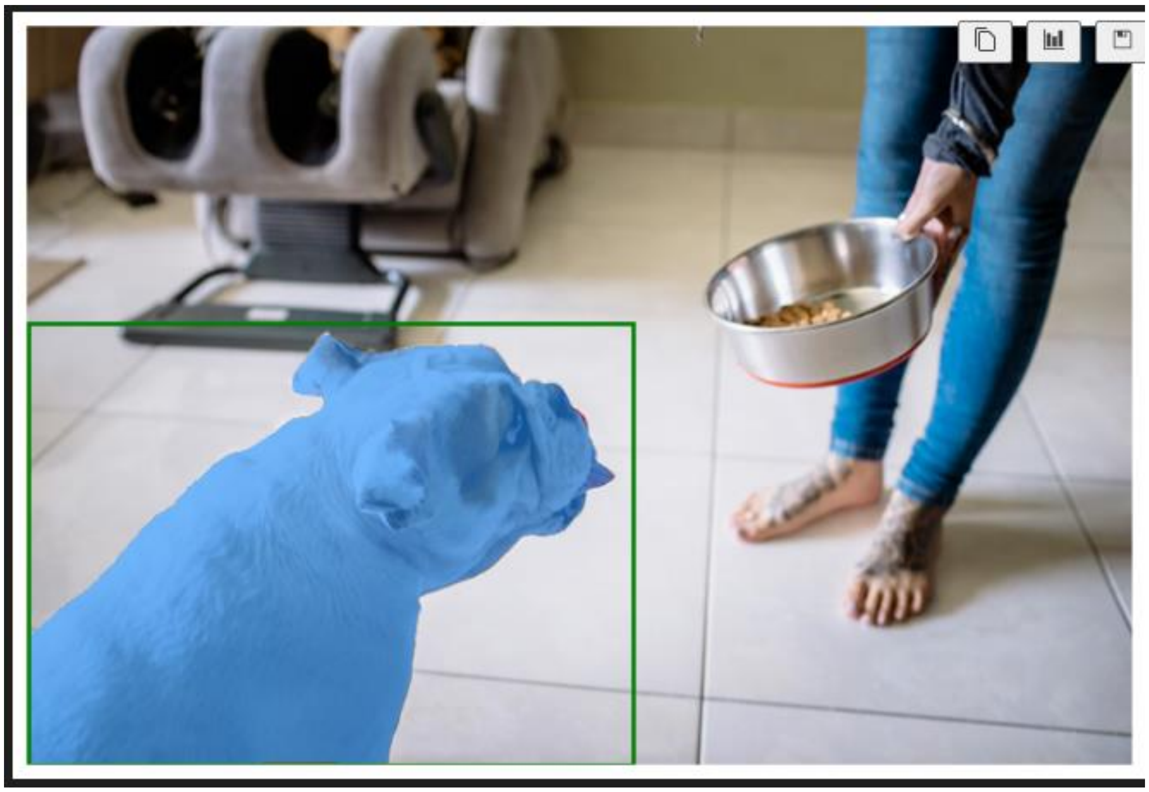
Step 2: Predict with the Bounding Box

The predictor.predict() function accepts a box parameter. I passed the input_box to this parameter to guide the segmentation:

```
masks, _, _ = predictor.predict(
    box=input_box,
    multimask_output=False,
)
```

I set multimask_output=False in this case because I only wanted the most likely segmentation mask.

Outcome: SAM successfully segmented the dog within the specified bounding box, demonstrating its ability to use box prompts for object segmentation.



Exercise 10: Combining Points and Boxes

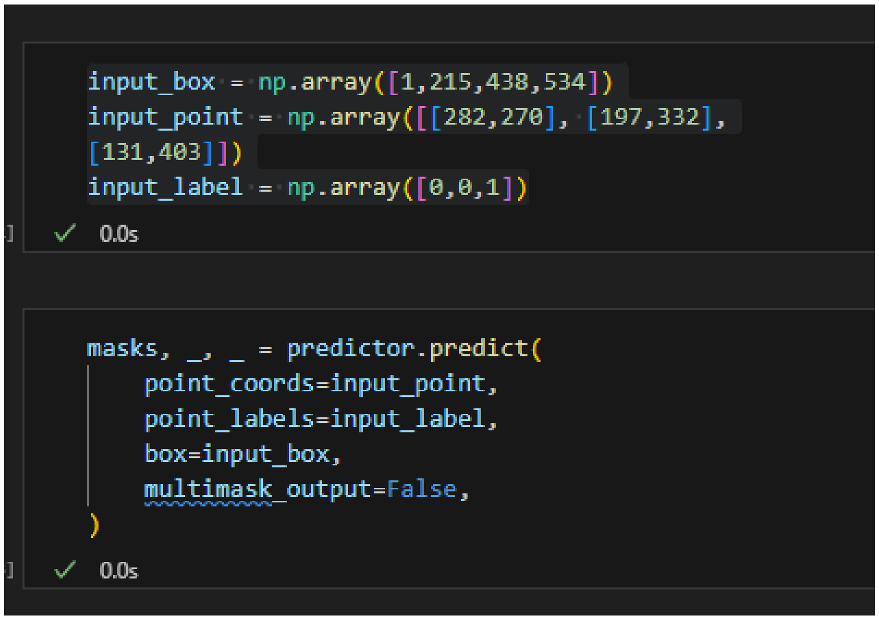
Step 1: Define Points and Box

I wanted to explore combining point and box prompts for even finer control. I defined both point coordinates (input_point) with labels (input_label) and a bounding box (input_box).

Step 2: Predict with Combined Prompts

I passed both the points and box to the predictor.predict() function:

```
masks, _, _ = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    box=input_box,
    multimask_output=False,
)
```

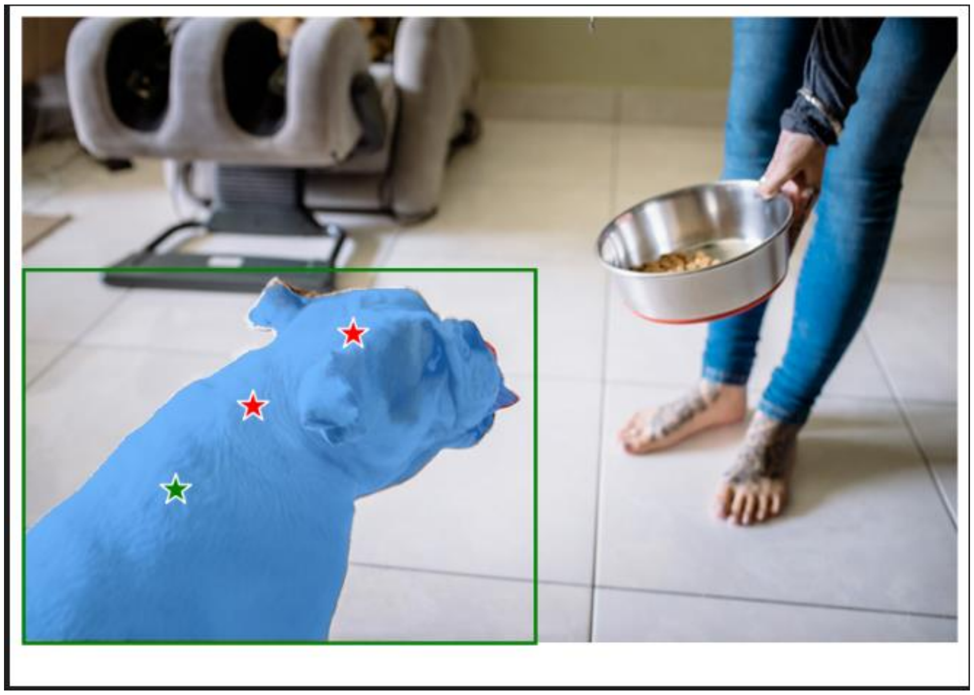


```
input_box = np.array([1,215,438,534])
input_point = np.array([[282,270], [197,332],
[131,403]])
input_label = np.array([0,0,1])
```
✓ 0.0s

```
masks, _, _ = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    box=input_box,
    multimask_output=False,
)
```
✓ 0.0s

Outcome:

SAM effectively used the combined prompts to generate a highly accurate segmentation mask. This flexibility in prompt types showcases SAM's powerful ability to accommodate various forms of user input for precise object segmentation.

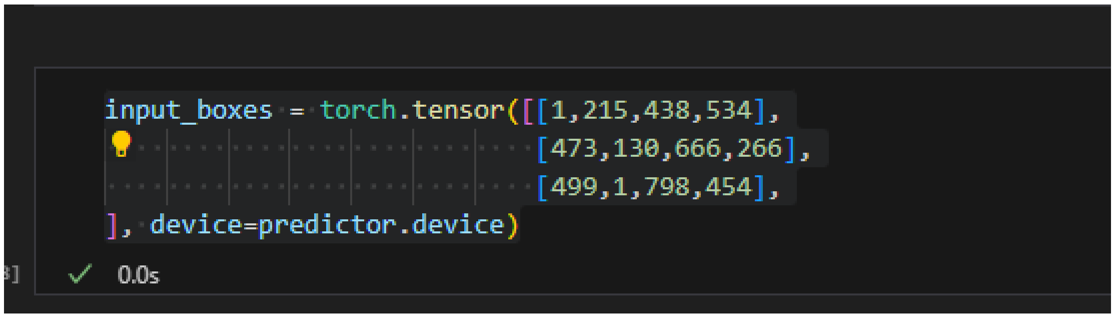


Exercise 11: Batch Prompts Input

Step 1: Define Multiple Bounding Boxes

To segment multiple objects simultaneously, I defined three bounding boxes corresponding to the dog, bowl, and human in the image. I stored these box coordinates in a PyTorch tensor named input_boxes:

```
input_boxes = torch.tensor([[1,215,438,534],
                    [473,130,666,266],
                    [499,1,798,454],
], device=predictor.device)
```

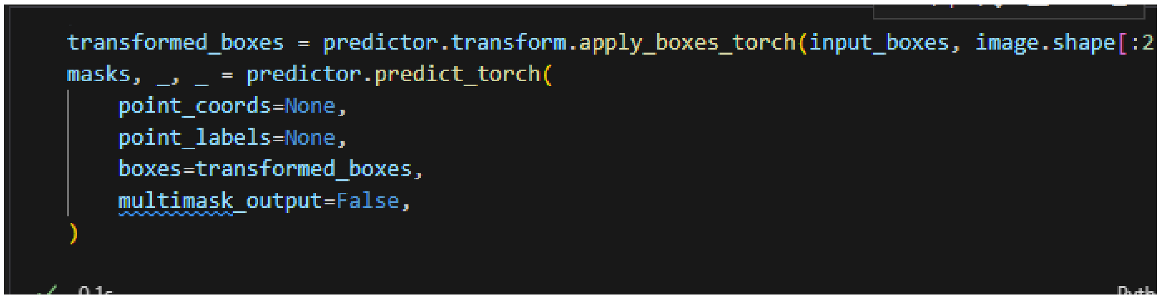


```
input_boxes = torch.tensor([[1,215,438,534],
                            [473,130,666,266],
                            [499,1,798,454],
], device=predictor.device)
✓ 0.0s
```

Step 2: Transform Boxes to Input Frame

The SamPredictor expects bounding box coordinates in the original image's coordinate frame. I used the predictor.transform.apply_boxes_torch() function to transform the input_boxes to the correct frame:

```
transformed_boxes = predictor.transform.apply_boxes_torch(input_boxes, image.shape[:2])
```



```
transformed_boxes = predictor.transform.apply_boxes_torch(input_boxes, image.shape[:2
masks, _, _ = predictor.predict_torch(
    point_coords=None,
    point_labels=None,
    boxes=transformed_boxes,
    multimask_output=False,
)
```

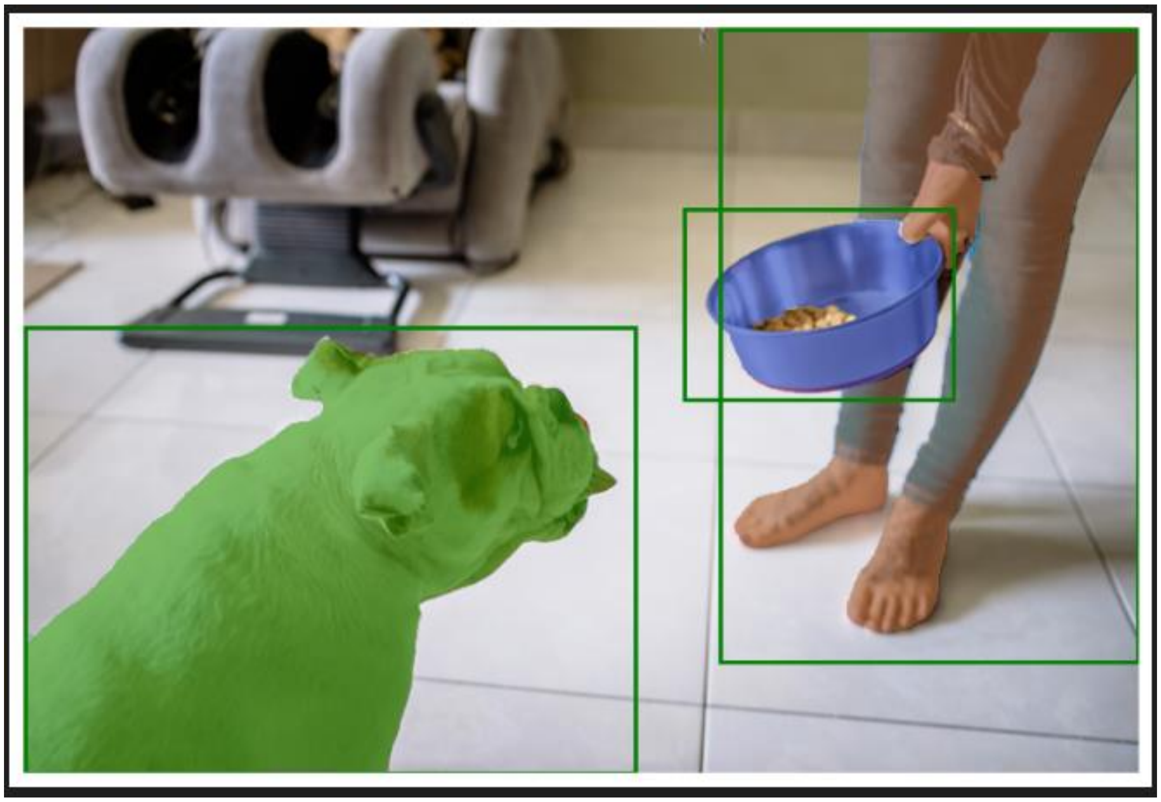
Step 3: Predict with Multiple Boxes

Instead of predictor.predict(), I used the predictor.predict_torch() method, which accepts input in tensor format. I provided the transformed boxes as input:

```
masks, _, _ = predictor.predict_torch(
    point_coords=None,
    point_labels=None,
    boxes=transformed_boxes,
    multimask_output=False,
)
```

Outcome:

SAM successfully generated segmentation masks for all three objects (dog, bowl, and human) based on the provided bounding boxes. This capability to handle multiple prompts in batch format demonstrates SAM's efficiency and its suitability for processing multiple objects simultaneously.



Exercise 12: End To End Batch Processing

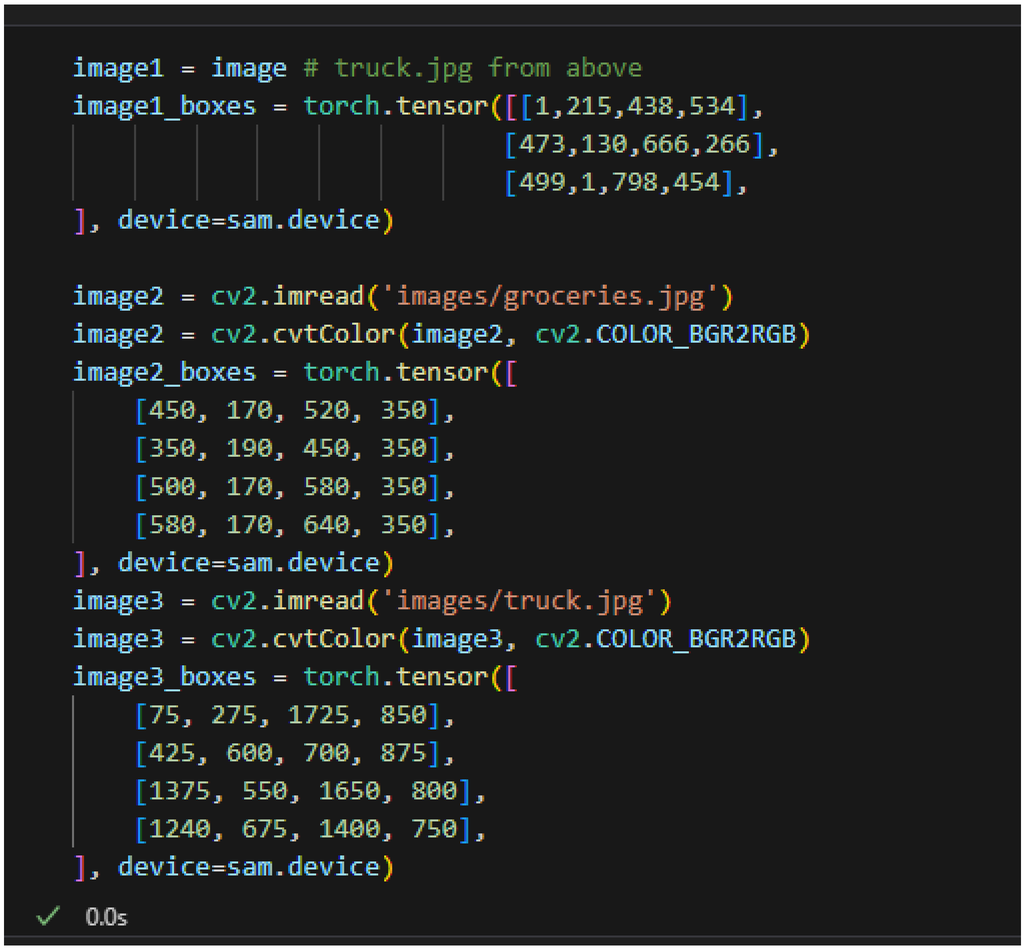
Step 1: Preprocess Images and Prompts

To leverage SAM's end-to-end batch processing capabilities, I preprocessed the images and prompts. This involved:

Reading the images and converting them to PyTorch tensors in CHW (Channel, Height, Width) format.

Defining the bounding boxes for each object of interest in each image and converting them to PyTorch tensors.

Transforming the bounding boxes to the correct input frame using the predictor.transform.apply_boxes_torch() function.



```python
image1 = image # truck.jpg from above
image1_boxes = torch.tensor([[1,215,438,534],
                             [473,130,666,266],
                             [499,1,798,454],
], device=sam.device)

image2 = cv2.imread('images/groceries.jpg')
image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2RGB)
image2_boxes = torch.tensor([
    [450, 170, 520, 350],
    [350, 190, 450, 350],
    [500, 170, 580, 350],
    [580, 170, 640, 350],
], device=sam.device)
image3 = cv2.imread('images/truck.jpg')
image3 = cv2.cvtColor(image3, cv2.COLOR_BGR2RGB)
image3_boxes = torch.tensor([
    [75, 275, 1725, 850],
    [425, 600, 700, 875],
    [1375, 550, 1650, 800],
    [1240, 675, 1400, 750],
], device=sam.device)
```

✓ 0.0s

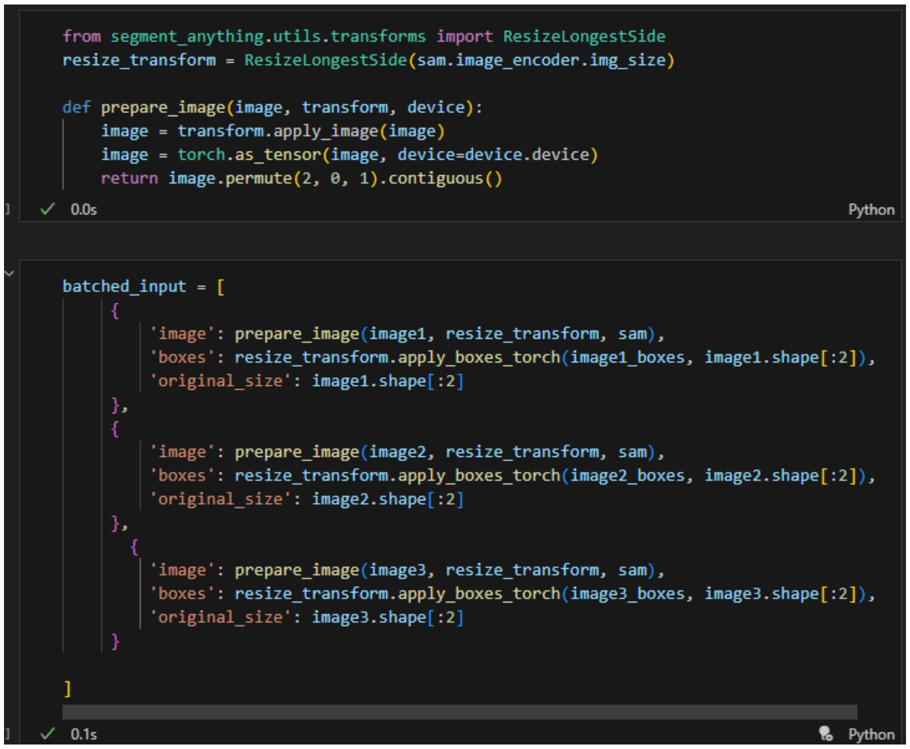
Step 2: Package Inputs

I packaged all the preprocessed data into a list, where each element represented a single image. Each element was a dictionary containing the following keys:

image: The image tensor.

original_size: The original size of the image (Height, Width).

boxes: The transformed bounding boxes for the image.



```
from segment_anything.utils.transforms import ResizeLongestSide
resize_transform = ResizeLongestSide(sam.image_encoder.img_size)

def prepare_image(image, transform, device):
    image = transform.apply_image(image)
    image = torch.as_tensor(image, device=device.device)
    return image.permute(2, 0, 1).contiguous()
```

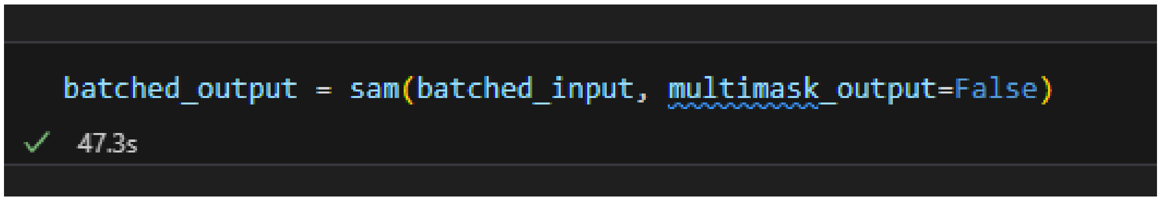
```
✓ 0.0s                                                    Python
```

```
batched_input = [
    {
        'image': prepare_image(image1, resize_transform, sam),
        'boxes': resize_transform.apply_boxes_torch(image1_boxes, image1.shape[:2]),
        'original_size': image1.shape[:2]
    },
    {
        'image': prepare_image(image2, resize_transform, sam),
        'boxes': resize_transform.apply_boxes_torch(image2_boxes, image2.shape[:2]),
        'original_size': image2.shape[:2]
    },
      {
        'image': prepare_image(image3, resize_transform, sam),
        'boxes': resize_transform.apply_boxes_torch(image3_boxes, image3.shape[:2]),
        'original_size': image3.shape[:2]
    }

]
```

```
✓ 0.1s                                                    Python
```

Step 3: Perform End-to-End Prediction

I directly used the sam model for prediction, bypassing the SamPredictor in this end-to-end mode. I provided the list of preprocessed input data to the model.



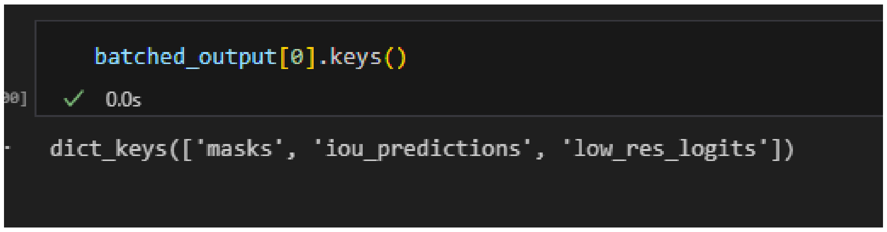
```
batched_output = sam(batched_input, multimask_output=False)
```

```
✓ 47.3s
```

Step 4: Process Output

The model returned a list of results, one for each input image. Each result was a dictionary containing:

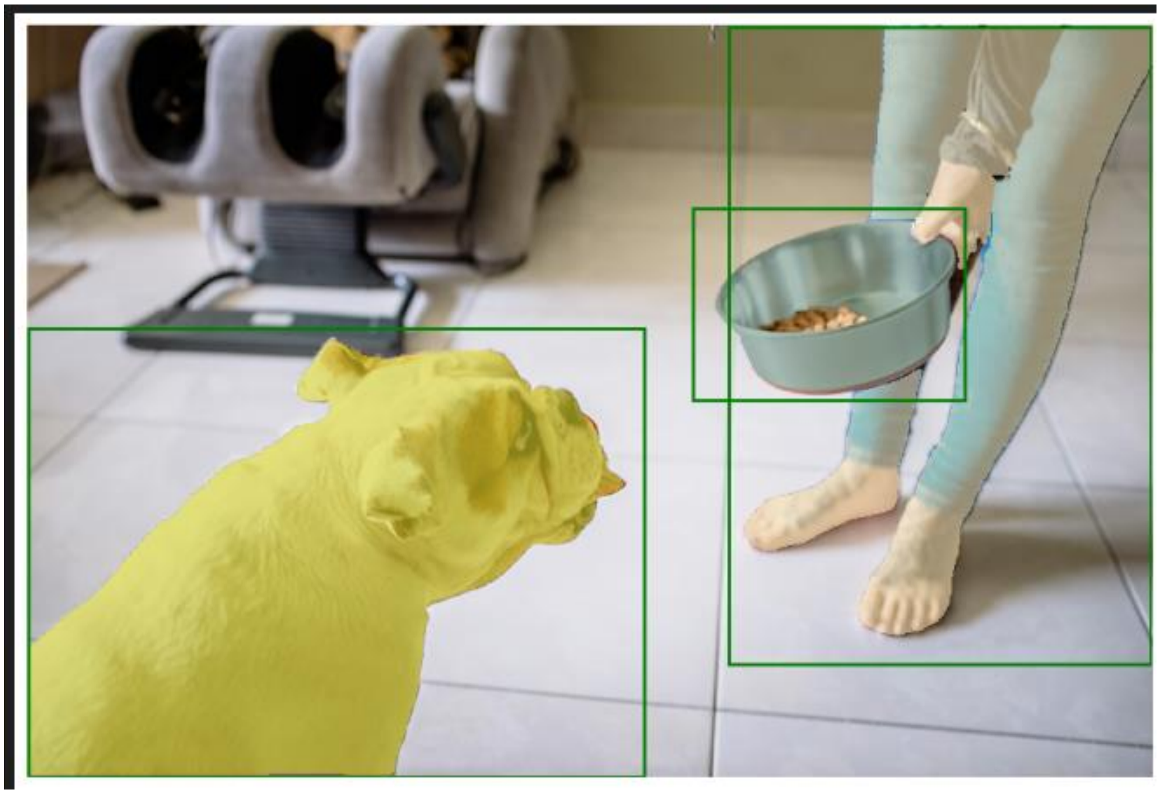
Masks , iou_predictions, low_res_logits

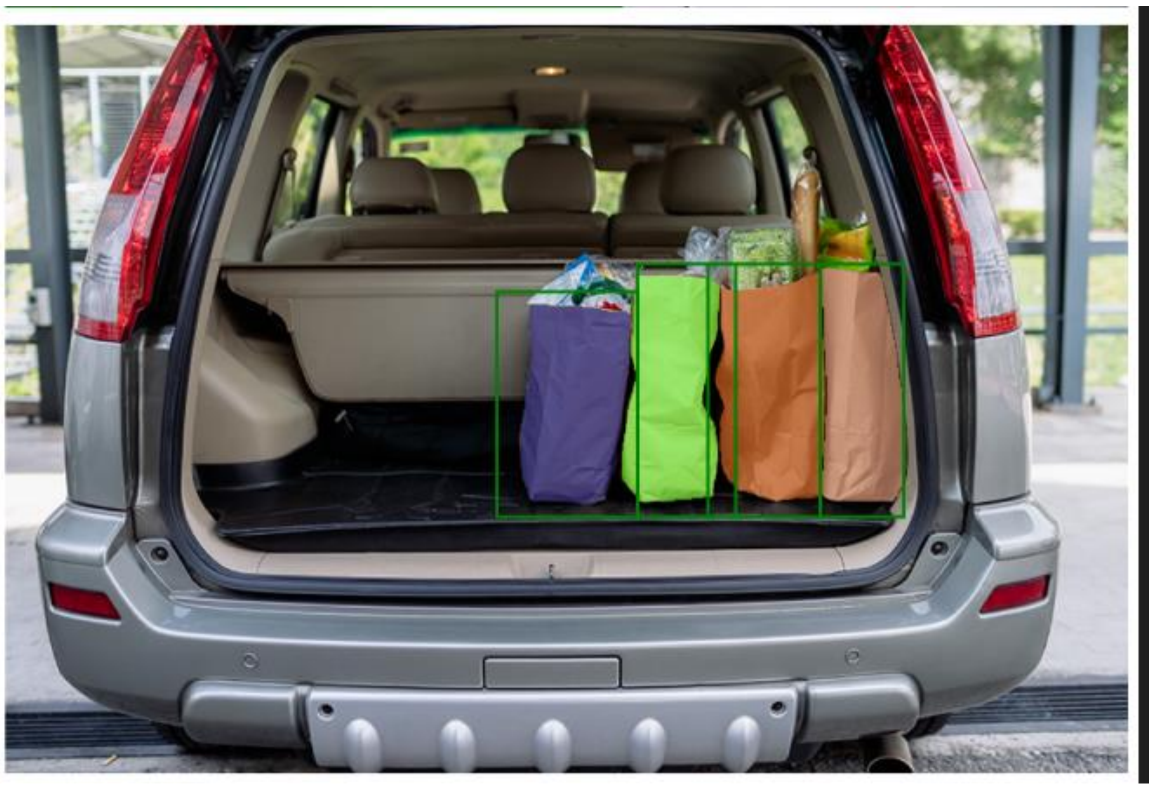


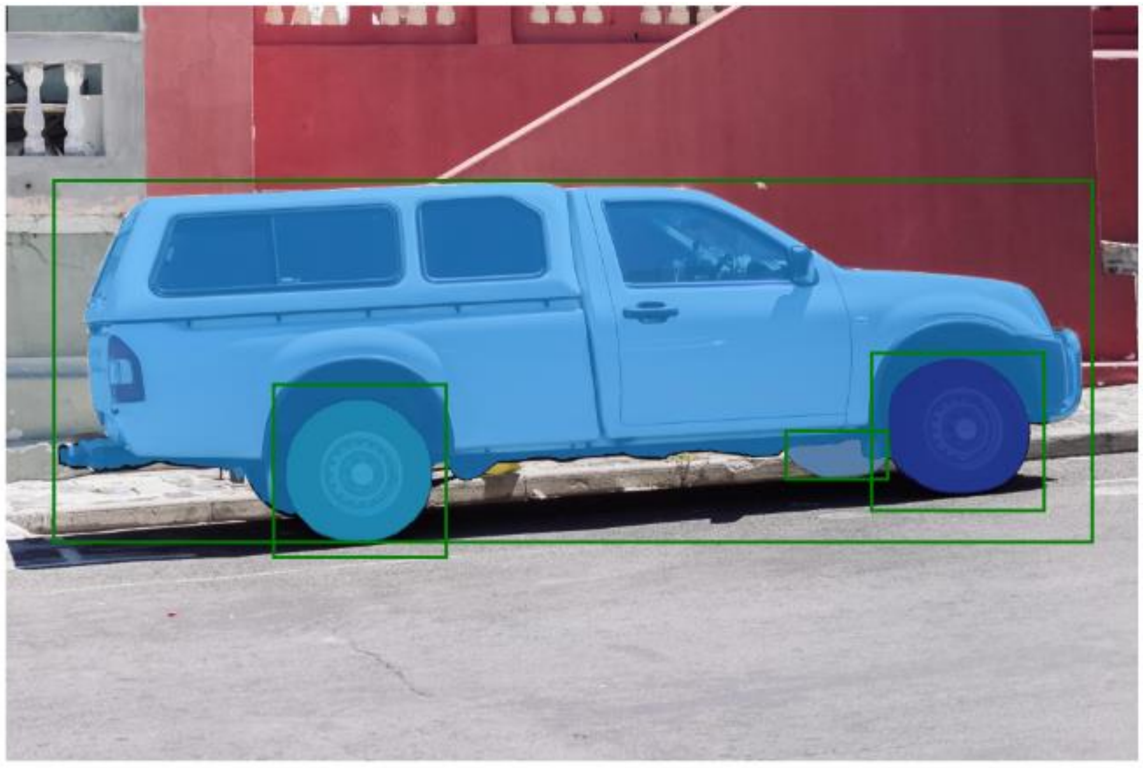
```
batched_output[0].keys()
✓ 0.0s

dict_keys(['masks', 'iou_predictions', 'low_res_logits'])
```

Outcome:

By performing end-to-end batch processing, I was able to efficiently segment multiple objects in multiple images simultaneously. This approach leveraged SAM's capability for parallel processing and optimized the segmentation workflow for scenarios where all prompts are known in advance.







Report score: ___________

Instructor's signature: ___________