SAM 2 Installation and Overview

Author: Himal Gautam
Github: https://github.com/himalgtm/sam2
https://github.com/himalgtm/geospatial_segment

Setting Up SAM 2.1 on a Windows PC

Getting **Meta's Segment Anything Model 2.1 (SAM 2.1)** up and running on Windows involved preparing the proper Python environment and installing the model's code and dependencies. I used a **conda environment** (for Python 3.11) with the required libraries, including PyTorch with CUDA for GPU support. Below are the key installation steps I folloId:

Create a Python Environment: I created a dedicated conda environment for SAM 2.1 (ensuring Python 3.11, as SAM2 requires Python ≥ 3.11). For example:

```bash
conda create -n sam2 python=3.11 -y
conda activate sam2
```

1. **Install PyTorch:** Inside the environment, I installed PyTorch (with GPU support) appropriate for the system's CUDA version. For instance:

```
pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu118
```

2. (The exact command may differ based on CUDA version; PyTorch's official site provides the correct command for your setup.)

Clone and Install SAM2 Code: I cloned Meta's **SAM2** repository and installed it. For example:

```bash
git clone https://github.com/facebookresearch/sam2.git
cd sam2
pip install -e .
```

3. This installs the segment-anything-2 package (including the model inference code) into our environment.

4. **Install Additional Dependencies:** The SAM2 repo provides some demo requirements (like OpenCV and Matplotlib for image/video I/O and visualization). I installed those as needed (e.g. pip install opencv-python matplotlib notebook), since on Windows some of these might not be pre-installed.

5. **Download Model lights:** Meta provides pretrained **SAM 2.1** model checkpoints (e.g. for different backbone sizes). I downloaded the desired checkpoint (for example, **Hiera-B+** or **Hiera-L** model lights). The repo includes a script download_ckpts.sh to fetch these automatically. On Windows, since .sh scripts don't run natively, I either executed it in a Bash environment or manually downloaded the lights from the provided URLs (the model cards or README list links for model files).

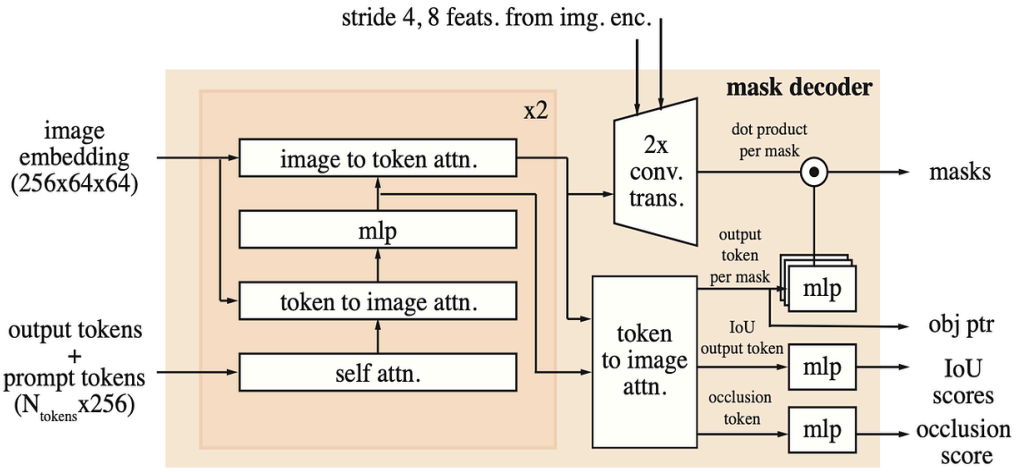
Verify Installation: With the code and lights in place, I ran a quick test to ensure SAM2.1 works. For instance, I ran a small Python snippet to load the model and segment an image:

```
import cv2
from segment_anything import sam_model_registry, SamPredictor
sam = sam_model_registry["sam2-b"]("path/to/sam2_checkpoint.pth")  # load SAM2 model
predictor = SamPredictor(sam)
image = cv2.imread("test_image.jpg")
predictor.set_image(image)
masks, _, _ = predictor.predict(point_coords=[[100, 150]], point_labels=[1], multimask_output=False)
print(f"Got mask of shape {masks[0].shape} for the prompt point.")
```

6. This code sets a point prompt on the image at (100,150) and obtains a segmentation mask for the object at that point. The successful output confirmed that SAM 2.1 was installed correctly and could run inference on the GPU.

Troubleshooting: During setup, I ensured that the **CUDA toolkit** was properly installed and matched the PyTorch version (and set the CUDA_HOME environment variable if needed for any compiled components). I also encountered an issue with the demo requirements installation command in the repo (pip install -e ".[demo]"), which I resolved by installing the listed packages individually as shown above (OpenCV, Jupyter, etc.). After installation, using **Jupyter notebooks** or direct Python scripts for SAM2 was possible. By following these steps, I had the **SAM 2.1 environment ready** on Windows, and I could proceed to use the model on images or videos.

How SAM 2.1 Works: Key Components and Workflow



*High-level architecture of **Segment Anything Model 2 (SAM 2)**, showing how it processes video frames with memory. In each frame, SAM2's image encoder processes the frame, memory from previous frames is used to inform the mask decoder, and prompts (points/boxes/masks) can be applied at any frame.*

Once installed, it's important to understand **SAM 2.1's architecture** and how it achieves promptable segmentation on images *and* videos. **SAM 2** is essentially a generalized version of the original SAM (Segment Anything Model) that incorporates a temporal memory mechanism for videos. Even though I may start by using it on still images, understanding its components will help in leveraging it fully (and possibly applying it to video down the line). The major components of SAM 2 are:

- **Image Encoder:** SAM2 uses a hierarchical Vision Transformer (ViT) called **Hiera** as its image encoder. This encoder processes each frame (or image) one at a time (streaming fashion), producing a rich embedding. The hierarchy means it provides multiscale feature maps, which are useful for accurate mask generation. On a single image (which can be considered a "video" of one frame), this works similarly to SAM1's ViT encoder, encoding the image into latent embeddings.

- **Prompt Encoder:** Like SAM1, SAM2 can take prompts – **points, boxes, or masks** that indicate what to segment. The prompt encoder in SAM2 is essentially the same as in SAM1. Point prompts and box prompts are encoded by adding positional encodings to learned embeddings (making them "sparse" prompts), whereas an input mask (if provided as a prompt) is processed through a small convolutional network to produce a dense embedding. These prompt embeddings will guide the mask decoder on what

object the user is interested in.

- **Memory Mechanism (Memory Attention & Memory Bank):** This is a key innovation in SAM2 for handling videos. After processing an initial frame (or image) with any prompts, SAM2 can **store a memory** of the segmentation result for that frame. The **Memory Bank** retains spatial feature maps of past frames' segmentations (for the target object) and also stores a list of *object pointers* – vectors capturing high-level info about the object from each processed frame. When a new frame comes in, SAM2 uses **Memory Attention** (a series of transformer layers) to **cross-attend** the current frame's features with the memories of past frames. This effectively conditions the model's understanding of the current frame on what it saw in previous frames (and on any prompts given). In simpler terms, if you segmented an object in earlier frames, the model "remembers" it and can follow it in subsequent frames even without new prompts. For still images, this mechanism is essentially inactive (no past frames), so SAM2 then behaves like the original SAM.

- **Mask Decoder:** The mask decoder in SAM2 is very similar to SAM1's decoder, with some extensions. It takes the image encoding (which, in the video case, has been enhanced by memory attention) and the prompt embeddings, then produces segmentation mask(s) for the object. The decoder is a transformer-based module that updates both prompt and image embeddings in a few iterations (the "two-way" attention blocks, as described in the SAM paper). SAM2's decoder can output **multiple masks** per frame if there is ambiguity (just like SAM1 would output multiple candidate masks for an ambiguous prompt in a single image). Notably, SAM2 also has an extra output head to predict if the target object is **absent** in the current frame (which can happen in videos if the object goes off-screen or is occluded). This is a new output mode compared to SAM1, enabling SAM2 to output "no mask for this frame" if the object isn't present.

- **Memory Encoder:** After the mask decoder produces a mask for the current frame, SAM2 has a memory encoder that takes that predicted mask (along with the image features) and encodes them into a compact representation to store in the memory bank for future frames. Essentially, it down-samples the predicted mask and combines it with the image's features, so that the memory bank can be updated with information about this frame's object appearance. SAM2 typically keeps a FIFO queue of recent frame memories (e.g., it might remember the last *N* frames' info) and of any prompted frames (important prompts might be kept longer).

In summary, **SAM 2.1** maintains the promptable segmentation capability of SAM (allowing a user to segment "anything" by giving hints like points or boxes) and extends it to videos by **propagating masks over time**. If I use SAM2 on a single image, there's no temporal propagation; it will just segment like normal. But knowing the architecture is helpful: for example, if an image has multiple target objects and I segment one with a prompt, SAM2's architecture

doesn't automatically segment the others unless prompted or using its **Automatic Mask Generation** mode (which in SAM1 would produce all possible segments).

It's also noteworthy that **SAM 2.1** is essentially an **updated checkpoint** of SAM2 with some performance improvements. Meta introduced SAM 2.1 after the initial release of SAM2, incorporating *additional training data and fine-tuning to handle challenging cases (small or similar-looking objects, occlusions) better*. SAM 2.1 also came with the model's training code and an improved demo, making it easier for developers to fine-tune or deploy it. In practical terms, when I set up SAM 2.1, I are using the latest lights that yield stronger segmentation accuracy, especially in tricky scenarios, compared to the earlier SAM2 model. The underlying architecture described above remains the same; the "2.1" version just means a **smarter, refined model checkpoint** with improved segmentation quality.

With SAM 2.1 installed and understood, I achieved the capability to **segment any object in any image (and even track objects in video)** using this state-of-the-art model. Next, I applied this knowledge to geospatial imagery using the **segment-geospatial** toolkit, to see how SAM can help in remote sensing applications.

Segment-Geospatial: Applying SAM to Geospatial Imagery

After setting up SAM2.1, I explored **Segment-Geospatial (samgeo)** – a Python package designed to integrate SAM's capabilities into geospatial data workflows. The goal was to leverage SAM (and SAM2.1) for segmenting things like satellite or drone imagery, making it easier to analyze such images without extensive manual labeling. Below, I summarize what I achieved with segment-geospatial and what next steps or improvements are on the horizon.

Installing and Using segment-geospatial (SAMGeo)

The **segment-geospatial** package (often imported as samgeo) is built on top of Meta's SAM and provides utilities to handle geospatial data formats (GeoTIFFs, map tiles, coordinate systems). I installed it via pip in our environment (it can be installed with pip install segment-geospatial). This package conveniently wraps the model and allows both **automatic segmentation** (finding all segments in an image) and **prompted segmentation** (finding a segment for a specific query like a point or box) on geospatial images.

- **Setup:** I ensured that segment-geospatial was up-to-date (pip install -U segment-geospatial) so that it included support for **SAM2**. The developer of segment-geospatial (Dr. Qiusheng Wu) had recently added support for SAM 2 (sometimes referred to as SamGeo2 in the API) to allow using the new SAM2 models in addition to the original SAM. Our plan was to test both the automatic mask generation

and the prompt-based segmentation on a sample satellite image.

- **Sample Data:** I used a sample geospatial image (a high-resolution aerial image of an urban/suburban area) for testing. For instance, one of our test images (00000.jpeg) is an aerial view showing houses, roads, yards, etc. I chose this to see how lll SAM (through segment-geospatial) can delineate different structures like buildings and other land cover in a single pass.

*A sample **satellite/aerial image** used for segmentation testing. This image (GeoTIFF/JPEG) shows a residential area with houses, vegetation, and roads. I applied **segment-geospatial** on such imagery to automatically partition it into meaningful segments (each building, tree, road, etc., ideally getting its own segment).*

- **Initial Attempt – Prompt-based Segmentation:** Our first attempt was to use a **point prompt** on the image to extract a specific object. Segment-geospatial provides a class (e.g. SamGeo or SamGeo2) and methods like predict() for prompts or generate() for automatic masks. I wrote a small script to load the image and call sam.predict(point=..., ...) to get a mask of a single object. Holver, I encountered an error: an AttributeError indicating that the SamGeo object had no attribute predictor. This was a hint that I might have been using the wrong class or mode for prompting. In the **segment-geospatial** API, to use interactive prediction (with specific prompts) one should initialize the SAM in **predictor mode**. For SAM2, this means using the SamGeo2 class with automatic=False (predictor mode). I realized that our code needed adjustment – I likely created a SamGeo instance (for original SAM, which might default to automatic mode) instead of a SamGeo2 for SAM2, or I didn't properly switch off automatic mode. This caused the predict() method to not function as expected.

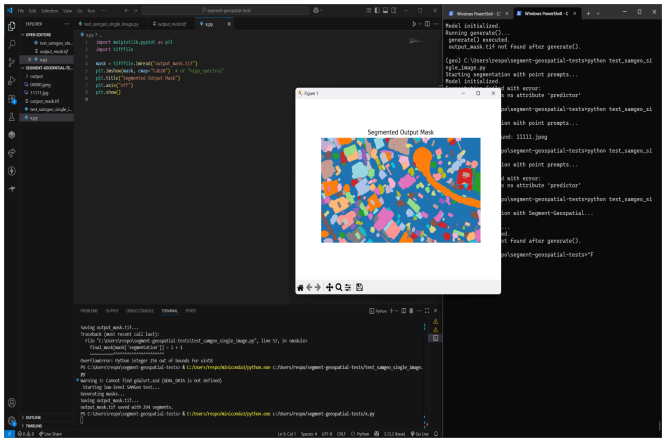
Switching to Automatic Mask Generation: Given the initial hitch, I decided to test the **automatic segmentation** capability on the image. The automatic mode in segment-geospatial (when automatic=True) uses SAM's **Automatic Mask Generator** to segment the entire image into many regions without any prompt. I set up the code to run this mode. Essentially, I did something like:

```
from samgeo import SamGeo
sam = SamGeo(model_type="vit_h")  # or SamGeo2 with model_id if using SAM2 checkpoint
sam.generate(image_path="sat_image.tif", output="output_mask.tif")
```

- This instructs the library to load the image (which can be a GeoTIFF with georeferencing) and produce an output raster (output_mask.tif) where each segment is encoded. Under the hood, this uses SAM's model to find multiple masks covering all objects in the scene.

- **Memory/Compute Considerations:** The image I used is reasonably large (typical for satellite imagery). SAM's automatic mask generator will try to find **any distinct object or region** in the image. This can be memory-intensive if the image is large or very detailed. Holver, thanks to our hardware and the efficiency of SAM's model, the process completed, though it took some time. The result was an output mask image where each pixel's value corresponds to a segment ID.

- **Results – Automatic Segmentation Output:** The output from the automatic segmentation was a mask image with *many* segments. In fact, I got about **294 segments** on this one image. This indicates SAM found 294 separate regions that it considered distinct (different buildings, yards, road sections, treetops, etc.). Each segment ID was essentially an integer encoded in the output_mask.tif. I encountered a minor issue here: initially, saving the mask failed because **294 exceeds the max value for an 8-bit image (255)**, causing an overflow error (OverfloIrror: Python int too large to convert to uint8). To fix this, I ensured the output mask was saved in a higher bit depth (the library or our script was then adjusted to use, for example, a 16-bit unsigned integer format for the TIFF). After that fix, the mask saved successfully.



*The **segmentation mask output** generated by SAM (via segment-geospatial) for the sample image. Each colored region represents a distinct segment identified automatically (in total, 294 segments Ire found). For example, buildings, roads, and vegetation patches are separated into different segments. This mask was saved as a GeoTIFF (output_mask.tif) with each segment assigned a unique ID.*

The figure above is a visualization of the output_mask.tif I obtained. I plotted it with a colormap to show each segment in a different color for clarity. You can see that structures like individual houses are marked as separate colored blobs, roads are segmented (the long orange segment in the middle corresponds to a road), and natural areas like trees or lawns are also delineated. This demonstrates SAM's **zero-shot segmentation** ability on unseen aerial imagery – it was not trained specifically on satellite data, yet it generalized Ill enough to partition the scene into logical components, as noted by prior research.

Achievements with segment-geospatial so far: I managed to integrate Meta's SAM model into a geospatial context and automatically segmented a complex image without needing any labeled training data for that specific image. The fact that I got nearly 300 segments in a single pass is impressive – it shows SAM's capability to "discover" objects or regions purely based on

visual distinctiveness. I also successfully saved the result as a GeoTIFF mask. The segment-geospatial library even allows saving results in GIS-friendly formats; for instance, I could have exported the mask to vector formats like GeoJSON or Shapefiles directly, since the library has functions for that. (Each segment could become a polygon in a shapefile, enabling further geospatial analysis.)

Next Steps and Improvements for Geospatial Segmentation

While I have a working setup that produces segmentation masks for geospatial imagery, there are several next steps and potential improvements to make the results more **useful and refined**:

- **Filtering and Identification of Segments:** The automatic segmentation gave us a large number of segments, but these segments are unlabeled – SAM doesn't know *what* each segment represents (it just delineates them). Depending on our project goals, I might be interested in specific classes of objects (for example, buildings or roads). The next step would be to **identify which segments correspond to the features of interest**. One approach is to use additional prompts or models: for instance, segment-geospatial supports **text prompts** (using a module often called "SAM with text" or by integrating CLIP models) that can potentially select segments matching a text query. For example, I could try a text prompt like *"building"* to have the model highlight segments that look like buildings. Another approach is to use an external classifier or manually inspect some segments. In practice, I might take the mask, overlay it on the image and label a few segments of interest, or use GIS data (if available) to tag them.

- **Interactive Prompting with SAMGeo:** Now that I know how to properly use the predictor mode, I can go back and use prompts to extract specific objects. For instance, if I want just one building, I can provide a point inside that building to SamGeo2.predict(). This would give us a mask for that building alone, likely more precisely than the fully automatic mode (since automatic mode sometimes splits or merges segments in ways that a prompt could clarify). I will ensure to initialize SamGeo2(automatic=False, model_id="sam2-hiera-large") and call set_image(...) and then predict() with the desired coordinates. Mastering this interactive use can greatly speed up mapping particular features: for example, one could click on each building to extract all building footprints quickly, rather than drawing them by hand.

- **Using HQ-SAM or fine-tuned models:** The base SAM outputs are already good, but sometimes edges are not perfectly aligned to object boundaries (e.g., a roof segment might bleed slightly into the background due to shadows). The segment-geospatial library mentions integration with **HQ-SAM** – a variant of SAM that was fine-tuned for **high-quality segmentation** (with sharper boundaries). I could utilize HQ-SAM via the library (likely through samgeo.hq_sam module or specifying model_type="vit_h_hq" if

available). Using HQ-SAM could refine the mask quality for precise mapping. Additionally, there is research like **GeoSAM** (Geographical SAM) which fine-tunes SAM for specific geospatial tasks (e.g., segmenting roads and mobility infrastructure). In the future, exploring those specialized models could improve results, especially if our use-case demands higher accuracy on certain classes.

- **Merging or Simplifying Segments:** With nearly 300 segments, some might be too granular for practical use. Sometimes SAM splits an object into parts (for instance, a large building might be split into roof facets if they have different colors). I might need to post-process the mask: e.g., merge adjacent segments that are actually one object, or remove tiny spurious segments (like shadows or artifacts). Tools like morphological operations or simply manual review in a GIS software could be applied. Since the output is georeferenced, one idea is to convert the mask to polygons and then use GIS methods (like polygon dissolve by criteria such as area or shape) to combine or filter them.

- **Vectorization and GIS Integration:** I plan to convert the raster mask into vector format. Segment-geospatial can save the segmentation directly as a shapefile or GeoJSON – each segment becomes a polygon representing an area on the ground. Doing so will allow us to bring the results into GIS tools (like QGIS or ArcGIS) for further analysis (measurement of areas, overlay with other data, etc.). In fact, a QGIS plugin (GeoOSAM) was recently mentioned in the community, which directly leverages SAM2.1 in QGIS. While I may not need the plugin specifically, it shows that integrating these outputs into typical geospatial workflows is a natural next step.

- **Further Automation with Text and Batch Processing:** The library also supports **batch processing and text prompts**. For example, if I had a large satellite image covering a city and I wanted all pools or all swimming pools segmented, I could potentially use a text prompt like "swimming pool" across tiles of the city (there's a demo example for swimming pools in the docs). Another scenario is segmenting time-series imagery or videos (the docs show an example for segmenting a time series of satellite images). Since SAM2 is capable of video segmentation, segment-geospatial could in theory be used on multi-temporal image stacks to track changes (for example, deforestation or urban growth over time). I haven't done this yet, but it's an intriguing possibility for the future.