Research Tracker

Objective 1: SAM2 Segmentation Validation on Desktop GPU

Status: In Progress

Brief Description

Compare multiple different weight files and ultralytics implementations of SAM2 based on accuracy of predicted mask vs inference times. We need to choose the model that provides the best tradeoff between accuracy and inference speed in order to replace the current pipeline which is a Yolo based Object Detection (Bounding Box) pipeline. As an extension of the objective, the goal is to evaluate the reliability of object tracking for our use case across the various implementations of SAM2.

YOLO vs SAM2

Both of these are evaluated based on our use-case, which is to track the object (soccer ball) in order to serve as an object state estimator for the control policy to teach Hector soccer. The initial implementation was as follows:

1. Model: Yolo Bounding box
2. Dataset: 260-270 images of Soccer Ball (augmentation + variations) from Roboflow for fine-tuning
3. Frequency Achieved: 50 - 60 fps

The SAM2 model from Meta is an alternative suggested primarily because it produces segmentation masks which track the centroid of an object across frames better primarily due to variations of the mask with respect to actual boundary being lesser than bounding box fluctuations. The implementation is as follows:

1. Model: SAM2 (Native weights + MobileSAM & FastSAM from Ultralytics)
2. Dataset: 77 images of Soccer Ball from Roboflow for evaluation
3. Frequency Achieved: TBD

Comparison:

1. The YOLO bounding box seemed to fluctuate considerably more than the segmentation mask, creating high variations in the computed centroid of the object. This causes problems downstream in the calculations of the control policy.
2. Due to use of a fisheye lens camera chosen to increase the field-of-view, the object in the frame tends to get warped in pixel space due to the nature of the lens. YOLO isn't able to draw boxes around the object reliably in such situations and hence the centroid's

distance tends to infinity causing problems in the control policy. With SAM2, the hypothesis is that with the prompted segmentation pipeline (VOS), the input point we provide from the first frame will track the object much better due to the superior architecture and training of the model. This will ensure that the object's distance doesn't tend to infinity as often.

Progress Overview

I created a simple pipeline using hints from the demo notebooks provided by Meta to run the Automatic Mask Generator API which is meant to segment everything in the input frame, and the Image Predictor API which is meant to perform prompted segmentation using either a point in pixel space or a bounding box.

The code was setup using PyTorch and utilising the CUDA 12.6 toolkit for the RTX 2080 Ti GPU on the Desktop. Images were loaded from the zip file downloaded from Roboflow using COCO API within the Pycocotools library and fed to each of the following variants of SAM2:

1. Sam_2.1_heira_base_plus.pt
2. Sam_2.1_heira_large.pt
3. Sam_2.1_heira_small.pt
4. Sam_2.1_heira_tiny.pt
5. Ultralytics - FastSAM, MobileSAM

For each variant and for each image, the inference time (using time.time) and predicted mask accuracy was recorded and a mean value was considered aggregating the entire dataset's values and plotted below.

Plot obtained on the **prompted** image segmentation task on SAM2:

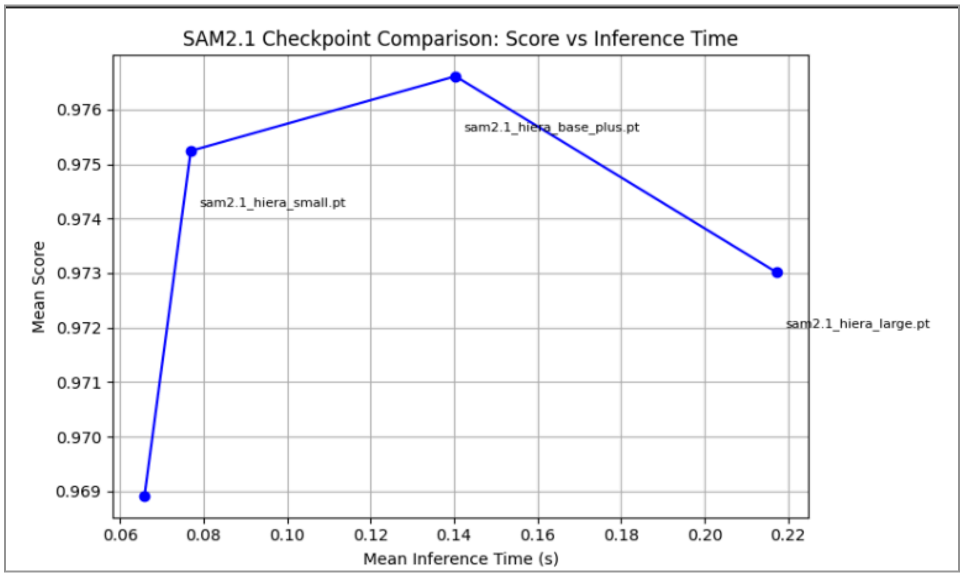


Fig: Comparison of different model checkpoint files inference time vs predicted_iou

Plot obtained on the **non-prompted** image segmentation task on SAM2:

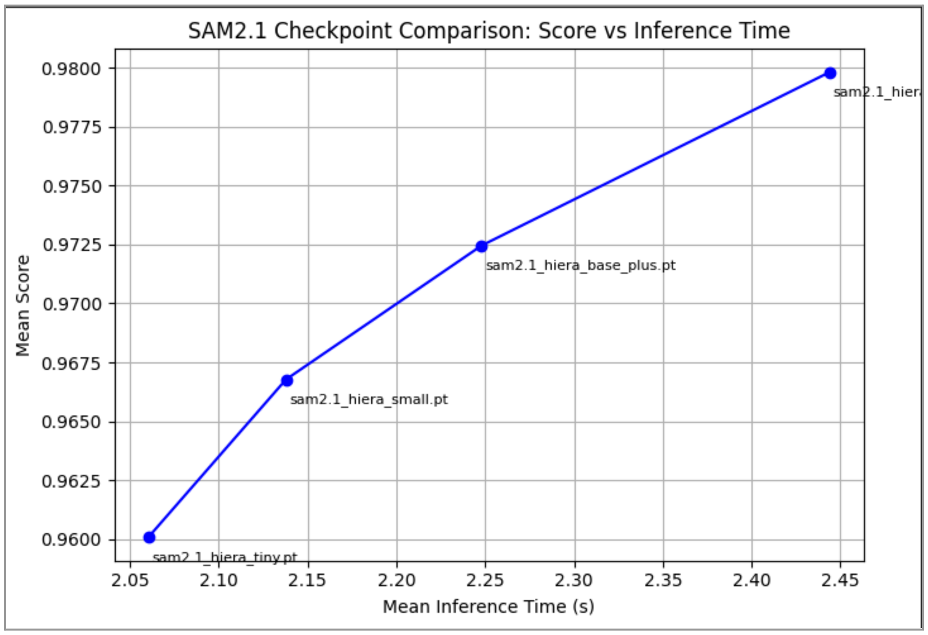


Fig: Comparison of different model checkpoint files inference time vs predicted_iou

Observations

1. It was observed that the non-prompted segmentation was on average 2-4 times slower than the equivalent weight for prompted segmentation. This is due to reduction in search space when you provide the region of interest as the prompt.
2. Between the smallest weight (tiny) and the largest (large), there isn't a significant difference in terms of predicted mask accuracy, but inference speed is almost 4 times faster for the tiny weight.
3. Hence, from the native implementation, we can expect a frame rate of about 12-16 fps.
4. When MobileSAM and FastSAM were implemented, although the predicted masks weren't as accurate in terms of IoU scores, the inference speeds went up significantly (~0.03 seconds per frame) indicating a frame rate of about 30-40 fps.

Next Steps

1. The next plan of action is to deploy the model on the Jetson Orin Nano to evaluate inference speeds and frame rates we can achieve on the edge device which is what the robot will use for computation.
2. In parallel, we need to create a dataset of videos with different trajectories of the soccer ball in order to evaluate the ball tracking capabilities of the different variants.

Objective 2: SAM2 Segmentation on Validation Jetson Orin Nano

Status: Done

Brief Description

Setup Jetson Orin Nano with relevant ROS, CV and GPU Acceleration libraries. Perform SAM2 inference of Jetson Orin Nano and evaluate frequency that can be achieved before and after optimisation. We need to do this both for static image segmentation as earlier and a custom video dataset to evaluate object tracking of the variants.

Current Setup on Jetson:
Jetpack: 6.0
CUDA: 12.2 (in-built)
cuDNN - 8.9 (in-built)
TensorRT - 8.6 (in-built)
ROS 2 - Humble
Torch - 2.4.0a0+3bcc3cddb5.nv24.07

Progress Overview

I set up the device with ROS2 Humble Hawksbill and studied the different ways of implementing a PyTorch model on a Jetson.

Jetson Inference vs TensorRT vs PyTorch API

Metric	PyTorch API	Jetson Inference	TensorRT (FP16)
Inference Time	19.50 ms/im (YOLO11n)	~8-15 ms/im	4.85 ms/im (YOLO11n)
Ease of Setup	Native (pip install)	Docker/Pre-built	Manual conversion
Hardware Control	Full GPIO/Camera	Limited by container	Full
Flexibility	Custom models	Pre-defined CV tasks	Any ONNX-compatible
FP16/INT8 Support	No	Limited	Yes (2-4x speedup)

Jetson setup with GPU acceleration through CUDA and PyTorch. It required the installation of additional SDK Components from the NVIDIA SDK Manager into Jetpack. We are currently using Jetpack 6.0 as PyTorch isn't supported on the latest version of Jetpack (6.2) (namely, cusparselt isn't supported with CUDA 12.6).

I installed NanoSAM and implemented our validation pipeline using a dataset from ROBOFLOW consisting of soccer balls, using COCO and PyTorch. We achieved a preliminary 10 Hz frequency of average on the dataset of about 70 images. In its current state, it is implemented with TensorRT models which are optimisations on the PyTorch model files.

Metrics returned: masks, scores, stability scores, areas, low_res_logits

I am now experimenting with the following:

1. NanoSAM with optimisation within TensorRT:
    a. useCudaGraphs - optimization by reducing compute overhead by treating some execution blocks as a graph and hence not needing to reload certain things. Speedup: marginal (0.3-0.4 Hz). Can't confirm whether it is actually using this feature when enabled, purely empirical.
    b. Alternate Image Encoders - Adding a much faster model that is architecturally very different from ResNet-18 will require re-training, so not explored. I tried using VIT-Tiny as an image encoder, but that dropped fps down to 5-6.
    c. Optimised Validation pipeline - Made a few optimisation tweaks within the optimisation pipeline. Only the centroid of the bounding box used as prompt instead of all 4 corners, increased frequency by about 4-5 Hz. The mask accuracy and quality didn't depreciate at least visibly. We can still track the bounding box, but using only the center for the segmentation cue.
    d. Quantization - Currently, the mask decoder is FP32 and image encoder is a mixture of FP32 and FP16. Quantization beyond this to INT8 doesn't really work or give added benefit.
        i. Quantization requires calibration datasets or ONNX Dynamic quantization which isn't supported by TensorRT, so we can't convert the model.
        ii. Jetson is already optimised to run FLOPS, INT ops have considerably less throughput.
    e. Quantization with FP16 - Some weights overflow and hence get truncated. There are quite a few warnings given by the Layernorm layers in MobileSAM Mask Decoder. Doesn't convert giving a "Adding shape ops very late in compilation failed" error. Works fine for the image encoder.
    f. **GELU Approximation and Return Single Mask** - All are optimisation flags possible while converting a .pt file to an ONNX file. The problem is that they utilize a 'IIOneHotLayer' layer somewhere in the pipeline that TensorRT doesn't support using the OneHot layer as part of the shape computation graph (seems to be a plug-in incompatibility). The fix would be to make the model just have static shapes of tensors (calculated at runtime instead of compile time). I can explore this option, but it involves going into the intricacies of ONNX file and model architecture, something that will take time. I tried using onnxsim to simplify the ONNX computation graph to make the model static and remove OneHot encoded layers but then I got dimension conflicts between what input the model wants at inference and what we're providing in our inference pipeline. We need to attempt constant-folding or static shapes. (Can do more investigation into this) Update: Tried simplifying the model, converting it to TensorRT engine and running the same pipeline on it. We gain about 1-1.5 Hz, but masks have much lower accuracy and quality, so aborting it.

What I haven’t tried with respect to optimisation:

1. Reduce Input Size - When i tried on Desktop GPU, it gave an appreciable increase in frequency (about 1.5x) when I reduced resolution from 1024 x 1024 to 512 x 512.
2. Profile pipeline and data flow - Figure out bottlenecks in the pipeline and processes that are taking the longest time.
3. **Applying all optimisations to ONNX files and running without TensorRT** - GELU Approximation, Return Single Mask and other optimisation techniques that ONNX allows but TensorRT doesn’t and running it. Last time I tried ONNX only was without a GPU and it gave about 1 Hz frequency. Update: You cannot run ONNX directly on the Jetson GPU because ONNX Runtime build does not support GPU acceleration. I can try building ONNXRuntime from source with CUDA and TensorRT support. It requires 4GB of swap space.

Installing ROS2 Humble on Jetpack 6.0 gave problems but worked fine on 6.2, because JetPack 6.2 ships with:

1. A more complete and updated Ubuntu 22.04 base.
2. Pre-configured APT sources pointing to ports.ubuntu.com by default.
3. A better-maintained root filesystem aligned with ARM64 expectations.
4. Fixes to avoid duplicate or broken entries in /etc/apt/sources.list.

I had to just redirect APT sources in sources.list using this command: sudo sed -i 's|http://us.archive.ubuntu.com/ubuntu|http://ports.ubuntu.com/ubuntu-ports|g' /etc/apt/sources.list. This just renames the relevant links from the x86 architecture to arm64 architecture’s APT repository.

Implementation of Tracker Node on ROS2

Status: In Progress

Brief Description

Porting the code to ROS2 code base and packaging it into a modular API for researchers to use as a plug and play tool. We will be using a YOLO object detector at time T=0 to generate a bounding box for the SAM2 mask segmentation that tracks the object in all the frames further until the object leaves the frame altogether. This node outputs position coordinates by using the camera extrinsic matrix, which serves as the object state estimator that will be the input to the Control Policy. So end-to-end, the node takes in a frame as input, performs segmentation across frames and then outputs position coordinates in the world/robot frame.

Progress Overview

1. I have implemented the YOLO initialization function that returns key points from the first frame.
2. I have implemented the SAM tracker node that takes the keypoints, calculates centroid (quicker) and provides that as the point prompt to the mask decoder (VIT_Tiny from MobileSAM). There is a function that runs an initial frozen frame mask segmentation using the keypoints from YOLO and then, the tracker tracks this object as it moves across frames as part of the callback function.

Experimentations Planned for Validation:

1. Static Test:
    a. Description: Camera mounted on robot, robot stationary and control policy deactivated, ball stationary.
    b. Feature Tested: Variation in Centroid positions in 3D coordinate space when there’s no relative movement -> Should test SAM2’s capability of generating masks purely.
2. Static Test:
    a. Description: Camera mounted on robot, robot stationary and control policy activated, ball stationary
    b. Feature Tested: Variation in Centroid positions in 3D coordinate space when there’s no relative movement but there is vibration induced by the control policy -> Should test whether SAM’s centroid offsets are proportional to the vibration amplitude so we can remove anomalies and compensate for the centroid offset
3. Static Test:
    a. Description: Camera mounted on robot, robot stationary and control policy deactivated, ball moving slightly
    b. Feature Tested: Tracking accuracy of SAM across frames with minimal relative motion and no vibration (best case scenario for SAM)
4. Static Test:
    a. Description: Camera mounted on robot, robot stationary and control policy activated, ball moving slightly
    b. Feature Tested: Tracking accuracy of SAM across frames with minimal relative motion and vibration (general case scenario for SAM)

Observations
No Vibration
Qualitative: When the camera is still, the variation is minimal and tracking accuracy is really good.
Quantitative:

Vibration
Qualitative: When no relative motion, one in about ten frames loses a mask and is much more when there is relative motion. Additionally, it tends to mask a totally different object when an object comes near the object of interest transferring the segmentation task to the new object, which is a big problem.

Renewed approach:
We treat the SAM Node as a client and we have an OWL-ViT running as a server all the time. Whenever there’s a frame where the mask goes None, we call upon the OWL-ViT server to

return us keypoints with the prompt ‘Vibrating Football’ as the frame fed by the Sam Node is a shaky image.
This works almost all the time with a few edge cases:

1. When the ball leaves the frame entirely and hence on calling OWL-ViT, it returns no detections and hence the mask is irretrievable
    a. Possible Fix: Wait for new frames until the ball comes back and call OWL-ViT again.
2. Masks other objects - happens sometimes where SAM starts to mask other objects by mistake, but OWL is called upon and rectifies that error within a few frames.
3. Every call to OWL-ViT takes about 2-2.5 seconds, which corresponds to inactivity. When vibrations and relative motion are high, there is a relatively high frequency of frames dropped causing inactivity for extended periods of time.

Frequency Improvement

Status: In progress

Brief Description

Currently the entire pipeline runs at about 7-8Hz. This involves receiving both frames, running NanoSAM inference on it with a point as prompt and converting it to 3D world coordinates. The majority of the time (95%) is taken up by the NanoSAM tracker's update function and hence that is the bottleneck. Additionally, each call to the OWL-ViT server takes about 4 seconds to regenerate the mask. This improves only by about 0.5 seconds by every double dimensionality reduction (reduced to half). So, the priority will be to both accelerate the update function in NanoSAM as well as reduce the number of calls to the OWL-ViT server.

Progress Overview

I am currently looking at two problem points:

1. [TRT] [W] Using an engine plan file across different models of devices is not recommended and is likely to affect performance or even cause errors.
2. UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Please consider converting the list to a single numpy.ndarray with numpy.array() before converting to a tensor. (Triggered internally at /opt/pytorch/pytorch/torch/csrc/utils/tensor_new.cpp:278.)
   image_point_coords = torch.tensor([points]).float().cuda()

Both the TensorRT engine files have been compiled for the Orin Nano but I still get this warning for some reason. The second point has to do with the internal code written in the predict method of the Predictor class. While frequency improvements should be minimal due to these two improvements, it will still be a valuable change in the code.

Next Steps:

1. Research into ONNX and TensorRT pipelines to check for any improvements in layer fusion or model optimisation.
   a. Check if a specific ONNX version exists for Jetsons.
2. Contextual prompting for OWL-ViT to improve inference time. Use Nano-OWL for Jetson.
   a. Works: 15-25 times speedup achieved. Using the COCO validation pipeline, OWL-ViT took about 3-3.5 seconds per image, whereas NanoOWL took about 0.25-0.35 seconds per image, without losing accuracy.
3. Attempting this pipeline with Jetpack 6.2 which should allow us to go to the MAX power mode.
   a. Found an appropriate Torch and Torchvision version for Jetpack 6.2 - https://ninjalabo.ai/blogs/jetson_pytorch.html

   b. Need to make a list of important files, folders and processes to replicate.
      i. Control Policy Packages - unitree_legged_real, brax_ctrl_py, joy
      ii. NanoOWL, NanoSAM validation scripts
      iii. ROS2 Package - object_tracking_api

4. Heuristic-based preemptive calling of OWL server when we seem to be losing the Mask (**test it out by running experiments and tracking confidence scores just before it loses the mask**). In similar ways, optimising the SAM2 pipeline via heuristics.
   a. No correlation between iou_scores and loss in mask, cannot sense any patterns. IoU does drop on the frame just before the loss, but it drops on other frames where it doesn't lose the mask also. So, we cannot really account for that.
5. Use mask post-processing to improve frequency.

Update: Having tried everything, the max frequency I am able to achieve is 7.5 Hz to 8 Hz. I will profile this pipeline and check where we're losing time the most and try to optimize that.

Update: On MAXN Super Mode on Jetson 6.2, the pipeline runs at between 13.5 Hz - 14.5 Hz. I've removed all possible warnings. Further optimisations are possible within the code for which I am profiling to find the bottlenecks.

Profiling TensorRT Model and Code

Status: In Progress

Brief Description

In its best state, the pipeline runs at between 12.5 Hz and 14.5 Hz on MAXN Super Mode on Jetpack 6.2. Profiling the code using NVIDIA Nsights Compute will help understand the bottlenecks in the pipeline and give us a priority order to optimise and a platform to choose an appropriate approach for multi-threading the pipeline to further improve accuracy.

Progress Overview

I started by fixing the Warnings from above that were being flagged by TensorRT during runtime. The first is an over-conservative warning thrown by TensorRT and the second was fixed with changes in Predictor class by replacing image.tensor() with image.from_numpy()
I started with building the TensorRT engine with different flags for both compile time optimisation and inference time optimization. Some of the flags I tried:

1. useCudaGraph
2. useSpinWait
3. memPoolSize=workspace:2048
4. Changed minShape, optShape, maxShape to reflect our experiment scenario

After this, I compared the profiles of all the various combination engine files. The image encoder runs almost half as fast as the mask decoder, so that seems to be the bottleneck. I haven't yeti explored multiple inference streams which is a flag in the compilation that allows you to start x multiple streams running the engine.

However, the engine alone is capable of up to 50 fps, given the time taken by each layer in each engine and summing up sequentially, as each frame should take only about 20ms.

After this, I profiled the code in ROS2 to study the bottleneck functions using NVIDIA Nsights. We annotate each function of interest with NVTX tags and then run the profile using nsys profile with appropriate flags to record events like CUDA, NVMedia, NVTX etc. I then uploaded it onto the Nsights GUI to study it and found that for 15W power mode, and got below summary:

Rank	**Avg per call**	**Notes**	**Why it dominates**
1 Tracker Update	127 ms (40 % of run-time)	red NVTX block	Wraps everything that happens after you've set the frame into the tracker.
2 SAM Tracker Callback	140 ms	green wrapper	Essentially ≈ Tracker-Update + a bit of camera / book-keeping.

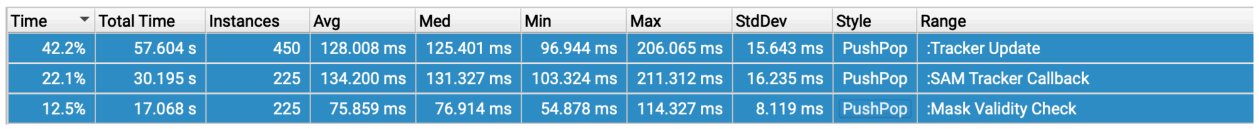
3 Mask Validity Check	73 ms	orange sub-range inside Tracker-Update	Pure Python/Torch ops on the *up-scaled* mask; runs on CPU.
4 Set Image	49 ms	purple block right before Tracker-Update	Image-preprocess + encoder TensorRT kernel + H2D/D2H copies.
5 Token Update / Reject	35 ms	red-orange	Blending tokens, fit_token, and IOU logic on CPU.
(Decoder) Run Mask Decoder	7 ms	GPU kernel	Actual low-res mask decoding isn't a bottleneck.
Camera Stream	2.6 ms	RealSense I/O	Negligible.

Immediate order of attack:

1. Rebuild encoder for FP16 and remove cv2.resize → expect 49 ms → 15 ms.
2. Move Token-Update math to GPU or a background thread → shave another ~20 ms.
3. Pipeline the work
    a. Thread A: grab camera, run encoder → decoder → publish.
    b. Thread B: perform Token-Update, IOU, possible re-init & HTTP.
    c. Exchange over a thread-safe queue so Thread A remains a stable 70–80 ms.
4. GPU-ise Mask-Validity & Token-Update
    a. Moving those two to CUDA often cuts them from 80 ms + 56 ms → < 10 ms combined, shrinking both fast and slow frames.
5. Cache & reuse keypoints
    a. Only calling OWL-ViT once every N frames or when IOU < 0.1 for three consecutive frames—prevents random one-off stalls.

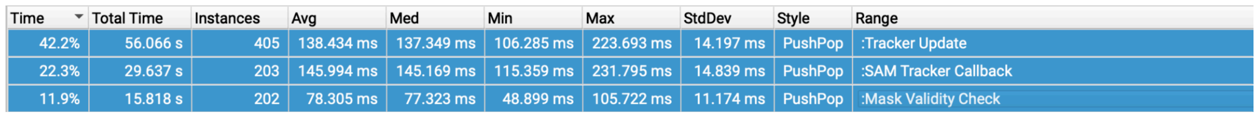
Current scenarios I am experimenting with:

1. Static Robot, static ball



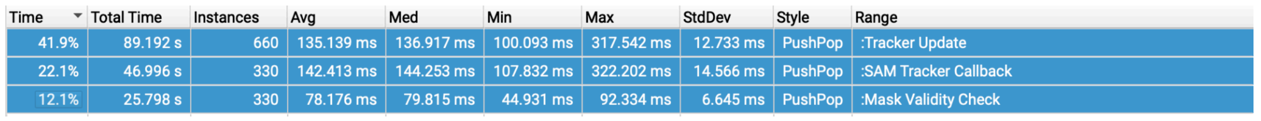
Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Style	Range
42.2%	57.604 s	450	128.008 ms	125.401 ms	96.944 ms	206.065 ms	15.643 ms	PushPop	:Tracker Update
22.1%	30.195 s	225	134.200 ms	131.327 ms	103.324 ms	211.312 ms	16.235 ms	PushPop	:SAM Tracker Callback
12.5%	17.068 s	225	75.859 ms	76.914 ms	54.878 ms	114.327 ms	8.119 ms	PushPop	:Mask Validity Check

2. Policy activated robot, static ball



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Style	Range
42.2%	56.066 s	405	138.434 ms	137.349 ms	106.285 ms	223.693 ms	14.197 ms	PushPop	:Tracker Update
22.3%	29.637 s	203	145.994 ms	145.169 ms	115.359 ms	231.795 ms	14.839 ms	PushPop	:SAM Tracker Callback
11.9%	15.818 s	202	78.305 ms	77.323 ms	48.899 ms	105.722 ms	11.174 ms	PushPop	:Mask Validity Check

3. Relative motion between ball and robot



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Style	Range
41.9%	89.192 s	660	135.139 ms	136.917 ms	100.093 ms	317.542 ms	12.733 ms	PushPop	:Tracker Update
22.1%	46.996 s	330	142.413 ms	144.253 ms	107.832 ms	322.202 ms	14.566 ms	PushPop	:SAM Tracker Callback
12.1%	25.798 s	330	78.176 ms	79.815 ms	44.931 ms	92.334 ms	6.645 ms	PushPop	:Mask Validity Check

Files currently changed for optimisation:

1. export_image_encoder.py - To incorporate 460 by 640 image input for image encoder, instead of 1024 by 1024 (Needs retraining of Distillation engine, re-exportation of ONNX model and then conversion to TensorRT)
2. predictor.py - To change up set_image and preprocess_image to move computation of mean, std and canvas into init.
3. Timm_image_encoder.py - To incorporate 460 by 640 image input for image encoder, instead of 1024 by 1024
4. train.py - To train new distilled model

Distilling Image Encoder

The image encoder was hardcoded to accept 1024 by 1024 dimension images. Our feed produces only 480 by 640 dimension images and it was an overkill to be scaling the image up by a factor of 3.4x. So, I tried scaling it into rectangle coordinates but that proved to be harder than I initially thought. I have hence scaled it to be an image encoder to accept 512 by 512 dimension images and the hope is that as this was the bottleneck of the pipeline (twice as slow as Mask Decoder), we should get immediate speedup.

Process followed:

1. Registered new model in TIMMImageEncoder.py. This should have a output feature map of 32 by 32 or 64 by 64 (both have different consequences)
2. Created a teacher model engine with same dimensions as before (ONNX -> Engine)
3. Train a student model by running train.py and referencing the newly registered model and using the teacher model as reference.
4. Convert the student ONNX model to TensorRT engine and can plug and play to use (hopefully)

Theory behind the training:

1. Image Encoders have a fixed input size as the output size is a function of this due to the many layers of CNNs whose input size is dependent on the feature map size of the previous layers. So, a new input size breaks semantic alignment that the layers would've learnt as spatial features.This is called distribution shift when test data is differently sized from training data.
2. The teacher-student pipeline is to get the student model to mimic the teacher by minimising a loss function between the output feature maps of the two models. The teacher is frozen and the weights of the student are learnable.
3. The student learns the behaviour of the teacher but at the lower depth, meaning it can only be as good as the teacher model. It usually ends up slightly worse off but works faster at the target resolution.
4. An alternative approach to fine-tuning where you'd train from scratch on your task, this is called Imitation learning where the teacher model's outputs are treated as ground truth and an unsupervised learning approach to teach the student.

5. By training a student:
    a. You teach it how to approximate the teacher's high-quality features, even when working at 512×512
    b. The student learns what to prioritize in low-res inputs to match what the teacher sees in high-res
6. In practice, the student isn't being trained to solve the entire task from scratch — it's just learning a compressed representation that approximates what a better model would do. This is especially useful when:
    a. You don't have task-specific labels (unsupervised distillation)
    b. You want to preserve domain knowledge (e.g. pretrained ViT on SAM2) without retraining everything
7.

Why Not Use a Teacher Directly?	**Why Train a Student?**
Teacher was trained on 1024×1024	Student learns to handle 512×512 optimally
Poor generalization to smaller sizes	Efficient and accurate at target resolution
Larger, slower model	Smaller, real-time-friendly encoder
Unadapted features at low-res	Adapted to preserve semantics at low-res

Files changed for inference with 512 image encoder:

1. predictor.py (image encoder size)
2. Export_sam_mask_decoder_onnx.py - embed size and embed dim
3. Build_sam.py - changed build_sam_vit_t function to incorporate change in dimensions
4. Tiny_vit_sam.py - changed x.view() in forward_features() to incorporate 32 by 32 feature map
5. Re-exported mask decoder with above changes using export_sam_mask_decoder_onnx.py
6. To incorporate changes for OneHot:
    a. Prompt_encoder.py - modified the **init** function
    b. onnx.py - modified the _embed_points function to eliminate OneHot logic
7. **Undid the changes for now, experiment failed because the ONNX export function they've provided makes use of OneHot encoding and we can't do much to change that, tried and there are never ending steps. Will try again down the line.**

Files changes to incorporate new image encoder and mask decoder in current pipeline in NanoSAM repo:

1. predictor.py - Changed image_encoder_size and orig_image_encoder_size to 512, changed if condition in run_mask_decoder(), revamped upscale_mask(),
2. tracker.py - Changed down_to_64 and up_to_256 functions to 32 and 128 respectively, changed fit_token and apply_token to be grid agnostic

Process of creating engine of student model (image encoder):

1. Export any checkpoint of student model to ONNX using instructions from nanosam's TRAIN section.
2. Run trtexec normally and it should create an engine file

Process of creating engine of Mask Decoder:

1.

Mask Post-Processing

Status: In progress (reading)

Brief Description

To minimize the mask error, we are going to apply some heuristic approaches that take into account contextual information to eliminate or improve mask output by NanoSAM. This will hopefully reduce the error that the control policy needs to tolerate in terms of 3D coordinates given by the object state estimator.

Progress Overview

So, I want to divide the mask post-processing into 2 steps:

1. Anomaly Elimination - Eliminate masks that are not in accordance with previous masks, Kalman Filters (maybe). This should remove the big peaks in position variation.
2. Mask improvement - To improve every mask output by enforcing shape and boundary constraints using Convex Hull, Shape constraints etc. This should improve the little variations we get in each iteration

These two are independent things that should lower the overall variance in ball positions.

Track volume of ball to be constant along with rate of change of area.

Kalman filter - prediction and correction.
Track velocity as well as IMU outputs from the tracking camera.

Pre-requisities of Kalman Filter
State space
Dynamics equations
Prediction and Correction Steps

Packaging the Code

Status: In Progress

Brief Description

We need to package this code into a repository for anyone with no background knowledge in vision-based state estimation to be able to use as a plug and play tool. We will need to make it generic enough to be task and class agnostic, but also customisable enough to make changes easily.

Objective

To build a state estimator that is task and class agnostic, generalizable, customisable and can scale easily.

Questions to Answer

1. How can we design general-purpose yet tailorable representations of visual input that make policies robust to environmental variations but still transferable across scenarios?
2. What logic do we use to select the keypoints? How do we support multiple objects?
3. How are you going to create configs (yaml) for different objects, cameras etc with options to specify properties and features to easily integrate into existing code?
4. How to get it to handle multiple objects at once? We will continue to return just the keypoint(s) of the objects being tracked.

Validating in Simulation

Status: In progress

Brief Description

The end goal is to test the quality of the tracker as an object state estimator for the control policy. So, now I'm going to train a simple policy in Isaac Lab/Sim and replace the randomised goal points used with the NanoSAM tracker to evaluate whether the pipeline works and is robust.

Objective

To validate the robustness and quality of keypoints provided by NanoSAM tracker to control policy.

Progress Overview

Next Steps:

1. Read and survey the Foundation Pose paper - present the tricks used
2. Setup Isaac lab in a virtual env. (Latest version)
    a. Env_isaaclab - already present in it. Name yours differently
    b. Train a velocity control policy for Go1. Use RSLRL (joystick control policy)

Current Progress:

1. Learned about simulating robots (articulation), rigid bodies, deformable bodies in IsaacLab using the Python standalone scripts method.
2. Learned about 2 ways of creating environments - ManagerBased and DirectRL. They offer varying levels of granular control and abstraction for simplicity. So, based on your need, you can use either.