# GazeSAM: Real-Time Gaze Prompted Image Segmentation

**Antonio Luera**
MIT
jluera@mit.edu

**Fabrizzio Orderique**
MIT
porderiq@mit.edu

**Nicole Stiles**
MIT
nstiles@mit.edu

## Abstract

The Segment-Anything Model (SAM) [5] is a vision foundation model facilitating promptable and zero-shot image segmentation. However, SAM models are highly computationally intensive and lack a flexible prompting mechanism. To tackle this problem, we introduce GazeSAM, a new gaze-prompted image segmentation model. We deliver a real-time demo of GazeSAM on NVIDIA's 3090 GPUs.

## 1 Introduction

SAM achieves top-tier performance on a wide range of vision tasks [5]. However, SAM misses the mark for real-time segmentation, running at 12 frames/second on a GPU while the generally accepted minimum throughput for smooth video is considered to be 24 frames/second [6].

In addition, SAM primarily supports point and bounding box-based prompting, both of which are not flexible enough for use in applications like AR/VR. To solve this problem, this work develops gaze as a new input format for image segmentation models, shown in Figure 1. Gaze-based prompting makes it easier for users to interact with SAM-based models in use cases ranging from assistive technology to human-computer interaction and virtual reality. In addition, to tackle the efficiency issues of SAM-based models, this work explores both the algorithmic and systems-based optimizations used to enable running GazeSAM in real time.

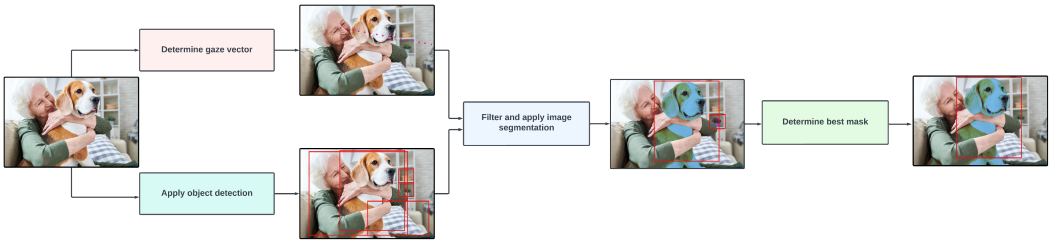


Figure 1: **Overview of the GazeSAM pipeline.** We find the intersection between the gaze vector and object bounding boxes. This subset of bounding boxes is used to selectively segment the image. Postprocessing and filtering are then applied to determine the final segmentation output.

## 2 Background

### 2.1 Segment Anything

SAM is comprised of an image encoder, prompt encoder, and mask decoder as shown in Figure 2. The image encoder produces an image embedding. As most of the image's context is encoded in the image embedding, the encoder is the heaviest of the three components. SAM's prompt encoder

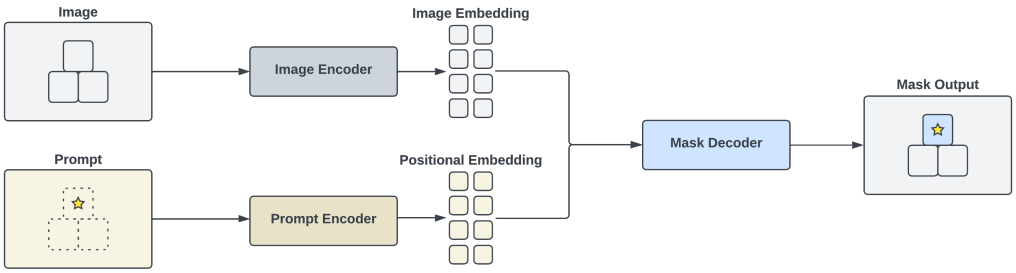


Figure 2: **SAM pipeline.** SAM is composed of an image encoder, prompt encoder, and mask decoder. The embeddings produced by the image and prompt encoder feed into the decoder network to produce the final mask predictions.

takes in two types of inputs: sparse and dense. Spare input types include text (not publicly available), points, and bounding boxes. The dense input type allows for prompting with masks, helpful if SAM is run for multiple iterations on the same image [5].

Given the image embedding and positional embedding, the decoder model produces a set of best-fit masks which segment the image within the constraints of the prompt boundaries. Since an object is often composed of multiple components (for example, a prompt located on a person's ear might produce a mask of just the ear, the whole face, and the person's face and hair), SAM produces a set of three best-fit masks at different granularities [5].

### 2.2 EfficientViT

SAM's image encoder is the model's primary latency bottleneck. On an NVIDIA A100 GPU, the image encoder (SAM-ViT-H) alone has a throughput of 12 images/s, while 24 frames/s is generally recognized as the minimum for smooth video [6]. EfficientViT, a follow-up work to SAM, achieves over an 80x increase in encoder speed, reaching a throughput of over 1000 images/s. It does this by replacing the inefficient softmax attention mechanism with a more hardware-friendly ReLU-based linear attention mechanism, reducing the module's time complexity from quadratic to linear [3].

Works like EfficientViT enable real-time execution of image encoders on cloud-based GPUs, which can shift the bottleneck from the image encoder to the image decoder. This occurs if the model must segment multiple areas within an image, requiring the decoder to run once per input.

### 2.3 Gaze Prediction

Gaze prediction has applications from determining consumer sentiment to detecting sleepy drivers. Given a front or side view of a person's face, gaze estimation models will produce a best-guess vector anchored between the eyes and along the direction of one's gaze.

Most approaches split this task into three parts [2]. First, an image is processed by a facial recognition model that produces a bounding box around the face. Given the bounding box, a second model performs landmark detection, identifying features like the eyes, nose, mouth, and chin. Finally, using landmark information, a third model determines the gaze's pitch, yaw, and roll angles. We can use the gaze's angles to render a gaze vector over the image or video input [4]. The complexity lies in the fact that while efficient gaze detection algorithms exist, the exact object a person is looking at is hard to determine from the gaze vector alone.

## 3 Methods

### 3.1 Pipeline Implementation

GazeSAM utilizes ProxylessGaze's [4] gaze detection model, EfficientViT's L0 image encoder model [3], and YOLO-NAS's object detection model[1]. The first step in the pipeline is to pass the image through the gaze detection pipeline shown in A. The headpose prediction and gaze angle prediction

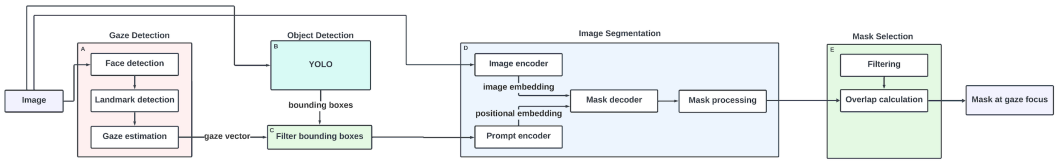


Figure 3: **GazeSAM system diagram.** In A, the gaze detection model takes in an image and produces a gaze vector. In B, we run the object detection model to generate bounding boxes around all objects in the image. In C, we filter the bounding boxes, keeping only those that intersect with the gaze vector. In D, we input the raw image and filtered bounding boxes to the image segmentation model, outputting candidate masks. In E, we perform additional filtering to produce the final prediction for the mask at the gaze's focus point.

are combined to produce the gaze vector. In B, we use the object detection model to find bounding boxes around all objects in the image. In C, we filter the bounding boxes to select only those that intersect with the gaze vector. This set of filtered bounding boxes are inputted into the prompt encoder in D. Both the prompt encoder and image encoder produce embeddings which are then fed into the mask decoder. Further processing - IoU threshold and stability score filtering, for example - are applied before outputting a set of candidate masks. In E, we take this set of candidate masks and filter away all masks and bounding boxes that relate to the person gazing in the image. We then determine which mask occupies the largest percentage of any bounding box along the gaze vector. This mask is then outputted as our final prediction.

### 3.2 Latency Optimizations

With the pipeline implemented, we turn our focus to latency optimizations. There are three categories of optimizations that we pursue: (1) implementing algorithmic refinements to decrease the latency of mask generation and selection code, (2) leveraging industry-standard tools designed for efficient inference like TensorRT, and (3) integrating research-based techniques like quantization to achieve additional performance gains.

#### 3.2.1 Mask Generation and Selection

As decoder speed was not a bottleneck in prior SAM-based works, the SAM/ EfficientViT code we built off of for mask generation has room for further optimization. Mask generation optimizations are especially impactful because the decoder is run once for every prompt encoder input.

One optimization we implemented was running the image segmentation model only on the bounding boxes along the gaze vector. SAM previously ran the decoder on points uniformly scatted across the entire image. We take out a filter operation that checks for and removes masks touching the edge of the image as we still would like to evaluate bounding boxes along the gaze vector that near the edge of an image. In addition, because we preemptively limit the number of segmentations we generate, we eliminate the need to perform RLE conversions.

#### 3.2.2 TensorRT for Inference

TensorRT is a tool created by NVIDIA to optimize model performance on NVIDIA GPUs during inference with the potential to reduce latency by 40x relative to the model's speed on a CPU [7]. This is done by fusing layers and removing redundant operations from the model during compile time, as well as applying hardware-specific optimizations.

EfficientViT's image encoder and mask decoder models originally ran in Pytorch. ProxylessGaze's face detection, landmark detection, and gaze estimation models ran using ONNX Runtime, a hardware-agnostic inference time optimizer. YOLO runs out of the box using Pytorch. Converting all six of these models to fp32 TensorRT engines can result in considerable performance enhancements which we further explore in Section 4.2.

### 3.2.3 Quantization

Quantization is a technique where the number of bits used to express a model's weights and activations are reduced, decreasing the model's memory footprint and latency while preserving accuracy. As quantization can result in over a 2x latency reduction, we experiment with both fp16-based and int8-based post-training quantization (PTQ).

We apply fp16 quantization to the gaze segmentation model, YOLO model, and segmentation model. We apply int8 quantization to just the YOLO and image segmentation models as all the gaze segmentation models are not a bottleneck to pipeline performance.

fp16 quantization of the gaze segmentation model and YOLO model were straightforward - we utilized TensorRT's built-in fp16 support. However, applying fp16 quantization to the image segmentation model's encoder and decoder adversely affects segmentation accuracy. We hypothesize that this drop in performance is due to crucial layers losing precision, and therefore they lose their ability to reliably store information. We attempt using fp16+fp32 mixed precision to recover accuracy. Specifically, we keep the layernorm calculations in fp32 as layernorm calculations are known to be harder to quantize. TensorRT warnings from fp16 runs empirically support this claim, stating that values in the norm layers are detected to be subnormal. While implementing mixed precision does not improve encoder accuracy, it does recover decoder accuracy.

In order to apply int8 quantization with TensorRT, we create a calibration cache using 5,000 images from the COCO dataset. This calibration cache is used by TensorRT's engine builder to get a precise estimate of the appropriate scale factors to use for quantization. These scale factors are calibrated by running the original model on inputs similar to those seen during inference time.

## 4 Evaluation

### 4.1 Pipeline Implementation

GazeSAM is successful in segmenting objects along the gaze vector at 25 FPS, achieving real-time performance. We using bounding-boxed based prompting, instantiate the image segmentation and YOLO models as fp32 TensorRT engines, and instantiate the gaze models as fp16 TensorRT engines. Because minute eye movements have the potential to change the mask prediction, to increase the stability of the mask prediction, we run the image segmentation model every other frame. Figure 4 provides an example of GazeSAM's output.

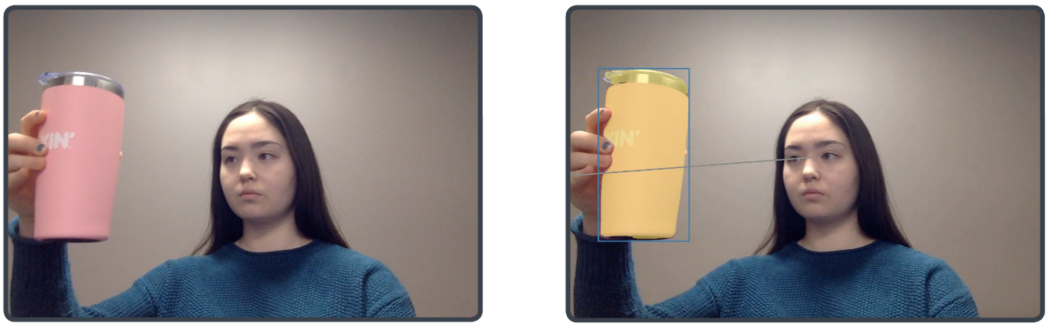

Figure 4: **GazeSAM prediction.** Input frame (left), GazeSAM's mask prediction (right).

### 4.2 Latency Optimizations

We measure the six engines' latencies, shown in Table 1. We measure latency with (1) the baseline PyTorch implementation, (2) compiled as a fp32 TensorRT engine, (3) compiled as a quantized fp16 TensorRT engine, and (4) compiled as a quantized int8 TensorRT engine. † indicates an engine that we recorded performance for but doesn't have sufficient accuracy to be used in real tasks. * indicates an engine implemented using mixed precision - in this case, fp16 + fp32.

All three gaze engines experience similar changes in latency when moving between model formats. All experience a >10x latency reduction when compiling to an fp32 engine from the baseline PyTorch

Table 1: Model latencies (ms) across baseline and different precision TensorRT engines, run on an RTX 3090.

| Model | PyTorch | fp32 engine | fp16 engine | int8 engine |
|---|---|---|---|---|
| Face detection | 10.63 | 0.87 | 0.77 | — |
| Landmark detection | 4.99 | 0.30 | 0.29 | — |
| Gaze estimation | 27.11 | 1.37 | 1.33 | — |
| Image encoder | 11.90 | 5.92 | 2.71† | 2.30† |
| Mask decoder | 15.26 | 18.74 | 15.33* | 18.24 |
| Object detection | 623.44 | 3.16 | 1.61 | 1.14 |

implementation. Compiling down from fp32 to fp16 does not result in a large change in latency. We do not benchmark the gaze engines' int8 performance as these engines are not the bottleneck of the system.

The image encoder and mask decoder models are the most difficult to optimize, which we hypothesize is due to the transformer units present in both models. Going from Pytorch model to TensorRT engine cuts latency in half for the encoder, but further attempts to quantize the model result in large drops in accuracy. We are unsure as to why the PyTorch model performs similarly to the engine versions of the model. We suspect this accuracy decrease relates to TensorRT's layernorm sensitivity as discussed in Section 3.2.3. When quantizing the fp32 decoder engine, we must apply mixed precision to preserve accuracy, manually converting layernorms to fp32. Interestingly, the mixed precision implementation of the mask decoder results in lower latency than the int8 implementation.

Using YOLO-NAS's object detection model out of the box via its predict function results in higher than expected latency. However, converting to an fp32 TensorRT engine shifts latency back in line with published YOLO-NAS metrics. We chose YOLO-NAS for its ease of quantization, and our results support this claim - latency of the fp16 model is approximately half of the fp32 model.

## 5 Conclusion

This work enables gaze as a new type of prompt input for image segmentation tasks, unlocking a wider range of image segmentation applications from assistive technology to virtual reality. In addition to implementing gaze-based prompting, we apply algorithmic optimizations, TensorRT engine creation, and quantization to run GazeSAM in real time. Future work will explore additional latency optimizations to enable GazeSAM to run in real time on edge devices.

## References


[1] Shay Aharon, Louis-Dupont, Ofri Masad, Kate Yurkova, Lotem Fridman, Lkdci, Eugene Khvedchenya, Ran Rubin, Natan Bagrov, Borys Tymchenko, Tomer Keren, Alexander Zhilko, and Eran-Deci. Super-gradients, 2021.

[2] Haldun Balim, Seonwook Park, Xi Wang, Xucong Zhang, and Otmar Hilliges. Efe: End-to-end frame-to-gaze estimation, 2023.

[3] Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. Efficientvit: Multi-scale linear attention for high-resolution dense prediction, 2023.

[4] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware, 2019.

[5] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything, 2023.

[6] Margaret Kurniawan and Hiroshi Hara. A beginner's guide to frame rates in movies.

[7] Piotr Wojciechowski, Purnendu Mukherjee, and Siddharth Sharma. How to speed up deep learning inference using tensorrt, 2018.