Efficient Segment Anything on the Edge

by

Nicole Stiles

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by: Nicole Stiles
Department of Electrical Engineering and Computer Science
May 9, 2024

Certified by: Song Han
Associate Professor, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Efficient Segment Anything on the Edge

by

Nicole Stiles

Submitted to the Department of Electrical Engineering and Computer Science

on May 9, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The Segment-Anything Model (SAM) is a vision foundation model facilitating promptable and zero-shot image segmentation. SAM-based models have a wide range of applications including autonomous driving, medical image segmentation, VR, and data annotation. However, SAM models are highly computationally intensive and lack a flexible prompting mechanism. On an NVIDIA A100 GPU, SAM runs at 11 frames/second, missing the mark for real-time performance and preventing the usage of SAM on edge devices. To tackle both the latency constraint and the prompt flexibility constraint, we introduce GazeSAM, a new real-time gaze-prompted image segmentation model. GazeSAM uses face and gaze detection to determine the direction of a user's gaze, object detection to find candidate objects of interest, depth estimation to perform background detection, and image segmentation to generate masks. The final output is a mask segmenting the object at the focus of the user's gaze. By performing algorithmic optimizations, employing inference engines, and applying FP16 and INT8 quantization, we achieve a 24x speedup relative to the baseline FP32 PyTorch implementation. GazeSAM runs at a speed of over 30 FPS, enabling real-time performance on an RTX 4070 GPU.

Thesis supervisor: Song Han
Title: Associate Professor

Acknowledgments

I'd like to thank Prof. Han for welcoming me to the HAN Lab, opening up my eyes to the incredibly interesting field of efficient machine learning, and guiding me through this thesis. Your mentorship has been instrumental in this journey - thank you for always taking the time to give me advice and provide encouragement.

A huge thank you to my mentor Han Cai - your suggestions, feedback, and guidance have been invaluable. Thank you for always being so generous with your time, for answering all my questions with such thought and care, and for supporting me every step of the way.

Thank you to my colleagues in the HAN Lab - I'm grateful to have had the chance to meet and learn so much from all of you this past year. Thank you for the warm welcome and making me feel like a part of the lab, and for setting an example of what it means to be both an excellent person and researcher.

To my friends - your steadfast support and encouragement over the past few years have meant the world to me. I'm looking forward to many more years together.

Finally, thank you mom, dad, lao lao, and Sophia for being there for me every step of the way. The best parts of me are learned from you - thank you for a lifetime of love, advice, and unconditional support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

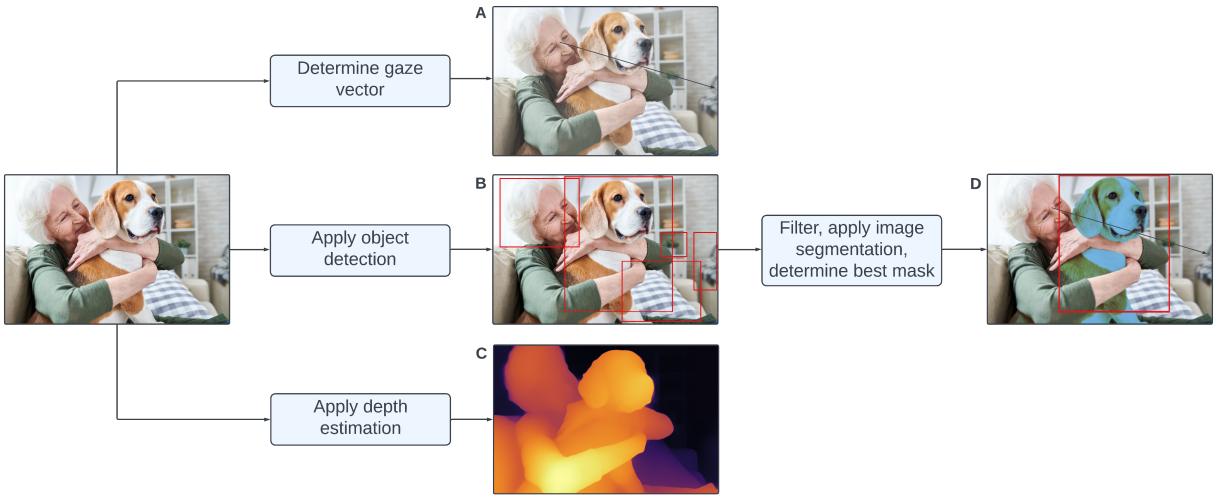

Figure 1.1: **GazeSAM Pipeline Overview.** We find the intersection between the gaze vector, mask generated by depth estimation, and object bounding boxes. This subset of bounding boxes is used as inputs to the image segmentation model. Postprocessing and filtering are applied to determine the final segmentation output.

Promptable image segmentation models have applications ranging from data annotation to medical image processing. Segment-Anything (SAM) [1] is a foundational work in this area, enabling zero-shot promptable image segmentation while also achieving top-tier performance on a wide range of vision tasks. However, SAM's state-of-the-art performance comes with a cost: slower inference speed, running at 12 frames/second on an A100 GPU. As such, SAM cannot be leveraged for real-time video segmentation tasks as real-time inference must be carried out with a speed of at least 24 frames/second.

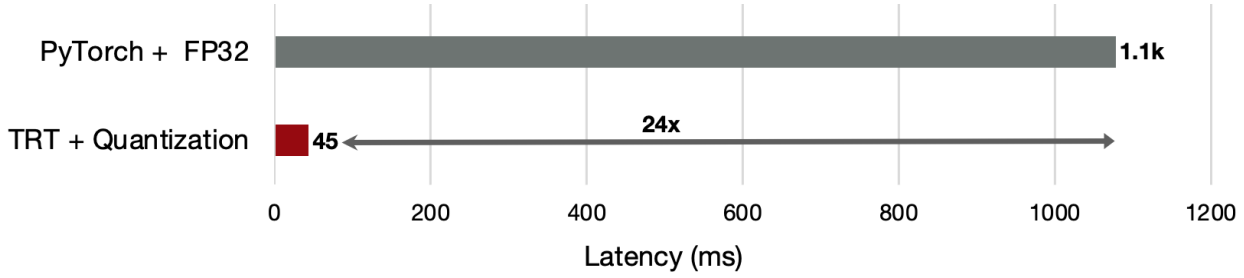

Figure 1.2: **GazeSAM Speedup.** GazeSAM's model component speedup using TensorRT-based quantization relative to the FP32 PyTorch baseline implementation. Latency is measured on an NVIDIA Jetson AGX Orin.

SAM and its follow-up works [2] [3] [4] enable point and bounding box prompts, which perform very well on single-frame image segmentation tasks. However, tasks in need of an efficient and easy-to-use prompting mechanism where the model must run in real-time may not best be supported by the point and box prompt types. A prompt type like a user's gaze, however, is much better suited for video-based frame segmentation tasks under real-time latency constraints. There are multiple advantages of this approach: gaze-based prompting requires very little human effort while also being incredibly flexible and efficient. Using gaze, applications of SAM-based models extend into enabling better assistive technologies, human-computer interaction, and virtual reality.

Prior works [5] explore gaze as a prompt mode for medical image segmentation, allowing physicians to quickly segment the image displayed on their screen during image diagnosis. To move beyond segmenting medical images, we employ an object detection model to determine candidate objects as well as a depth estimation model to add background awareness. This work broadens the ability of a gaze-prompted image segmentation model to segment any object between the user and the camera in real-time.

Four general tasks are necessary to enable gaze-prompted image segmentation: define the line of sight, detect potential objects of interest, remove unlikely bounding box and mask candidates, and segment the image. Figure 1.1 illustrates the model components that loosely correspond to the tasks listed above. Section **A** of the figure defines the line of sight and generates a gaze vector using the ProxylessGaze [6] model for face detection. The facial bounding box is then used by L2CS-Net [7] to perform gaze estimation. To detect potential objects of interest, we use the YOLO-NAS [8] object detection model to

generate bounding boxes around all potential candidate objects in the image, shown in **B**. The Depth-Anything [9] model in **C** is used to perform background detection and helps filter out irrelevant bounding boxes and masks. The EfficientViT [2] image segmentation model processes bounding boxes and outputs segmentation masks corresponding to each bounding box, shown in **D**. Figure 1.2 illustrates the 24x speedup we experience after performing latency optimizations. GazeSAM's model components use FP32, FP16, or INT8 precision depending on each component's ability to be quantized while maintaining accuracy.

The main contributions of my thesis are:

1. The design and implementation of a gaze-prompted image segmentation system (GazeSAM), described in Chapter 3. GazeSAM is capable of running in real-time on an RTX 4070.

2. An overview of the methods used to improve GazeSAM latency, from applying algorithmic optimizations to leveraging industry tools like TensorRT. Runtime acceleration experiments are performed across the PyTorch, ONNX, and TensorRT runtimes. In addition, TensorRT quantization experiments measure the speedup and accuracy preservation abilities of GazeSAM across FP32, FP16, and INT8 precisions using the entropy and minmax calibration algorithms. Setup and results are detailed in Chapter 4.

3. The implementation of components that can be used to quickly and easily interact with SAM-based models in a web interface. The interface supports point, box, point and box, and gaze prompts along with full image segmentation. Component design as well as GazeSAM segmentation results are discussed in Chapter 4.

Chapter 2

Related Work

2.1 Segment Anything

Segment-Anything (SAM) [1] is a widely used vision transformer model that produces excellent results on image segmentation tasks. More specifically, its novelty lies in enabling promptable image segmentation. SAM is comprised of an image encoder, prompt encoder, and mask decoder, shown in Figure 2.1. The image encoder uses an MAE [10] pre-trained Vision Transformer (ViT) [11] to produce 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 (but only needs to run once per image).

As expected, SAM's prompting capabilities are enabled by its prompt encoder. SAM's prompt encoder takes in two types of inputs: sparse and dense. Spare input types include text, 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. Sparse prompts like boxes or points are represented using a positional encoding, then summed with a learned embedding. For points, this embedding indicates whether the point is in the foreground or background (foreground indicates that the object at the point should be segmented while background indicates the object at the point should be ignored). For boxes, we use a pair of embeddings to represent the top-left corner and the bottom-right corner, both of which are summed with learned embeddings specific to that corner.

The mask decoder is very lightweight in comparison to the encoder. It uses cross-attention

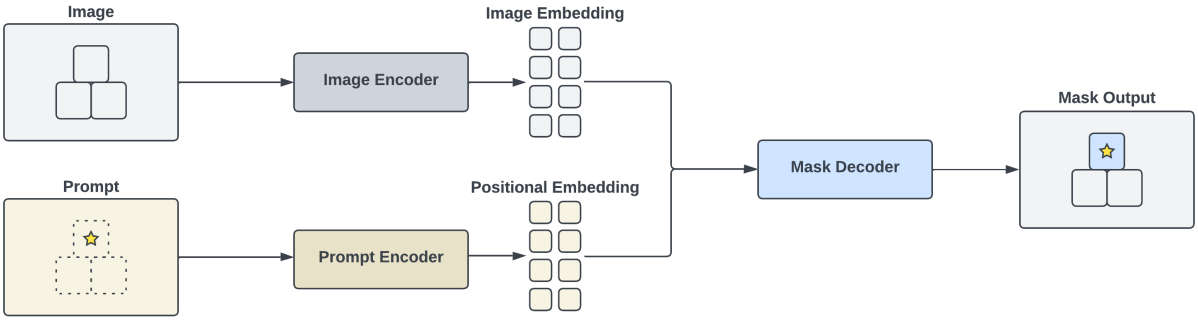

Figure 2.1: **SAM pipeline.** SAM is composed of an image encoder, prompt encoder, and mask decoder. The embeddings produced by the image and prompt encoder are fed into the decoder network to produce the final mask predictions.

to update the image embedding and prompt tokens. The updated tokens are used along with the upscaled image embedding to produce mask candidates. Since each 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 [1].

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 [12]. Multiple works [3] [4] [13] have examined the potential to make the image encoder more efficient.

EfficientViT [2] is one such example, achieving over an 80x increase in encoder speed and reaching a throughput of over 700 images/s (NVIDIA A100 GPU, batchsize 16). To improve latency, the inefficient softmax attention operation, quadratic in time and memory, is replaced with a ReLU-based linear attention mechanism, resulting in linear time and memory complexity. To preserve the local information extraction capabilities of the encoder, ReLU linear attention is supplemented by small kernel convolutions of varying sizes with nearby tokens to create multi-scale tokens. Depthwise convolution operations are also added to FFN layers. Together, these additions improve the local feature extraction abilities of the

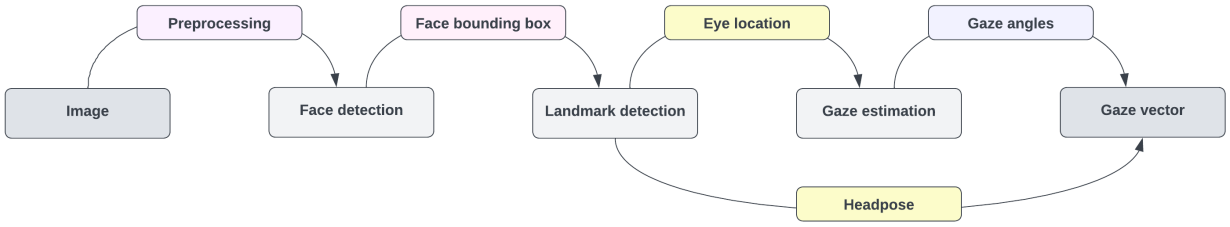

Figure 2.2: **Gaze estimation model pipeline.** The image is passed through a face detection, landmark detection, and gaze estimation model to produce the headpose, eye location, pitch angle, and yaw angle that define the gaze vector.

model and recover performance to match that of the original SAM model. When performing image segmentation, the EfficientViT encoder is used, along with the original SAM prompt encoder and mask decoder. The original SAM components continue to be employed as they are lightweight in comparison to the encoder.

2.3 Gaze Prediction

Gaze prediction has applications ranging from gauging consumer interest when viewing products 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 [14] shown in Figure 2.2. 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 [6] [7]. While efficient gaze detection algorithms exist, the exact object a person is looking at is hard to determine from the gaze vector alone. This motivates the need for an object detection model or depth estimation model to better determine the object of interest along the gaze.

2.4 Object Detection

Object detection models can be used to locate, bound, and label objects in an image or video with applications from surveillance to autonomous driving. Popular architectures powering object detection typically CNNs (R-CNN [15] and YOLO [16]) and transformers (ViTDet [10]).

YOLO-NAS is built off of previous versions of YOLO, specifically YOLOv6 and YOLOv8. In GazeSAM, we specifically use the YOLO-NAS-M 51M parameter model. YOLO-NAS adopts YOLOv8's transformer architecture but incorporates a new post-training quantization (PTQ) friendly block based off of the VGG architecture. YOLO-NAS [8] is pre-trained on the COCO, Objects365, and Roboflow 100 datasets. One advantage of YOLO-NAS is that the model is designed to be easily quantizable, which we further explore in Section 4.2.

2.5 Depth Estimation

Monocular depth estimation (MDE) has applications ranging from autonomous driving to virtual reality. These depth estimation models are used to determine how far away from the camera an object lies. Newer popular depth estimation models [17] [9] rely on a transformer-based architecture. GazeSAM uses Depth-Anything-Base [9], a new 97.5M parameter zero-shot depth estimation model that utilizes a DINOv2-based encoder [18] and DPT-based decoder [19] to achieve real-time performance.

Chapter 3

GazeSAM

3.1 Implementation

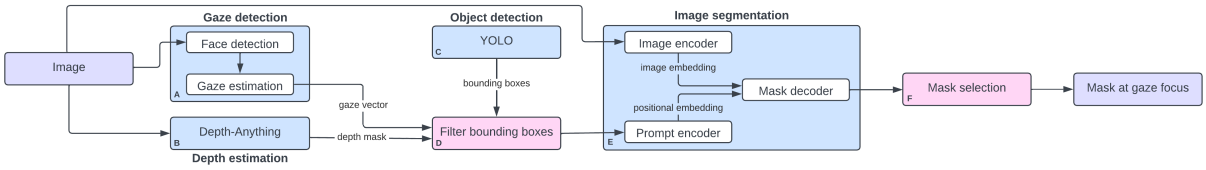

Figure 3.1: **GazeSAM system diagram.** In A, the gaze detection model takes in an image and produces a gaze vector. In B, we run a depth estimation model to produce a depth mask. In C, we run the object detection model to generate bounding boxes around all objects in the image. In D, we filter the bounding boxes based off the gaze vector and depth mask. In E, we process the raw image and filtered bounding boxes to the image segmentation model, outputting candidate masks. In F, we perform additional filtering to determine the mask at the gaze's focus point.

Referencing Figure 3.1, the first step of the pipeline is to pass the image through the gaze detection pipeline shown in **A** (detailed gaze estimation model diagram shown in Fig 2.2). The face detection model processes the image to produce a bounding box around the face. We use the center of the box to approximate the center of the gaze (midpoint between the eyes). The face crop is passed into the gaze estimation engine to produce the gaze angles (specifically pitch and yaw). Given the "start" of the gaze along with the pitch/yaw angles, we can define the gaze vector.

The Depth-Anything model in **B** takes the input frame and outputs a map indicating

each pixel's distance away from the camera. To determine which objects are exclusively in front of us, we use the depth of the eye location as depth zero. With this, we can create a binary depth mask where pixels corresponding to locations in front of the eyes are set to true. Because the depth model does make small errors when comparing objects of similar depth and we'd like to avoid false negatives, we also take pixels whose normalized depth values are within a margin of the eye's depth values. The addition of the margin allows pixels that have depths not strictly in front of the eye position (but may in reality lie in front of the person) to still be considered, helping to offset potential model inaccuracy.

The YOLO model shown in **C** also takes in the input frame and outputs bounding boxes around all detected objects in the image.

With the depth mask, gaze vector, and bounding boxes in hand, we can now filter the set of object bounding boxes in **D**. We keep bounding boxes that satisfy the two following conditions:

1. The gaze vector must intersect the bounding box. While any given bounding box on the gaze vector is not guaranteed to be the object of attention, we are guaranteed that a bounding box not on the gaze vector is not the object of attention.

2. The bounding box must identify an object that lies in front of the person depth-wise. We use the depth at the eyes of the person as "depth zero". This makes it possible to remove larger background objects and images that cannot possibly be in the user's realm of vision.

In **E**, we input the select set of bounding boxes in front of the user and in the line of sight to the EfficientViT model to produce an image embedding. The box prompts are passed to the model's prompt encoder, where the top-left and bottom-right corners of each bounding box are summed with learned embeddings corresponding to the two corners. The resulting positional embedding and image embedding are inputted to the lightweight two-layer mask decoder. Inside the decoder, the cross-attention is used to update the image embedding with the prompt embedding to provide it with positional context, then dot-producted with the processed tokens to finally produce the mask outputs. Different filtering

methods - for example, IoU threshold filtering, stability score filtering, and NMS (non-maximal suppression) filtering - are applied before three masks are produced with varying levels of granularity. We then extract the mask with the highest IoU of the three masks. Descriptions of these filtering techniques are described in more detail in Section 4.5.

To determine the final mask from all candidate masks produced by EfficientViT, we calculate the percentage overlap between the mask segmentation and its corresponding bounding box in **F**. The mask-box pair where the mask occupies the largest percentage of the bounding box becomes our final prediction. To render the mask over the image, we convert the image from RGB to RGBA and shade the area corresponding to the mask. The best match bounding box along with the gaze vector is drawn.

We can then save our annotated frame using cv2's `VideoWriter` class. Example results from running GazeSAM over different frames of video are shown in Section 4.4.

3.2 Quality Enhancements

Default Segmentations: Swift movements will sometimes result in blurry frames. This results in the object detection model being unable to detect objects. Therefore, no mask segmentations are produced. The result of this in the output video can appear mildly jarring. Therefore, when presented with these sorts of gaps, we use the previous frame's bounding box and mask to annotate the current blurry frame. Of course, it's possible that the user may not currently be looking at something. In this case, after the second frame in which GazeSAM doesn't naturally produce a segmentation, our output stops rendering the previous mask over the frame.

Removing Self-Segmentations: The object detection model frequently produces a bounding box containing the face and upper body of the user. One potential way to deal with this is to discard any box that contains the gaze vector head. However, this may incorrectly remove boxes bounding an object that simply lies very close to the eyes. Instead, we filter out the bounding boxes that contain the entirety of the face bounding box.

Gaze Smoothing: Because we run the gaze estimation model more than thirty times a second, minute eye movements will result in noticeable differences in the gaze vector's angle.

This is problematic when we are trying to segment a very small object. As such, we employ a 1€ filter to smooth and stabilize the vector.

Bounding Box, Segmentation, and IoU Filtering: A careful observer may notice that filtering the bounding boxes using the depth mask and gaze vector in **F** is distinct from filtering the actual segmentations contained *within* the bounding box. More specifically, as the pixels defined by the segmentation are a subset of the pixels defined by the bounding box, it's possible that the depth estimation check passes because the pixels *outside* the segmentation but *inside* the bounding box are depthwise "valid". This is especially true for masks with larger bounding boxes. Therefore, when processing to determine the final mask prediction, we check that the actual segmentation itself contains a portion of pixels that have positive depth. Finally, because our gaze vector still is rendered even when we aren't looking at something, sometimes the gaze vector will catch a small area that seems to be an object but isn't (for example, a section of clothing). To decrease the frequency of these sorts of events, we experimentally determine an IoU threshold and remove masks whose IoU values fall below the threshold.

3.3 Discussion

We explored other designs for GazeSAM before arriving at this final version. Initially, we didn't include the depth estimation model in the system and solely filtered the bounding boxes with the gaze vector. While this works with clean backgrounds (and results in a sizeable latency reduction), the downside of this approach is objects in the background may be segmented. As we determine the best mask by finding the mask/bounding box pair that has the greatest overlap, if the object detection model identifies a large background object (e.g. chair), the larger object is more likely to be outputted as our final segmentation instead of a smaller object that we're looking at. Depth estimation mitigates this problem by helping us to only consider objects in front of us (at the cost of additional latency).

We also experimented with prompting the image segmentation model with a set of uniformly scattered points along the gaze vector instead of prompting with bounding boxes. However, a box inherently contains more information about an object than a point does

even if they aim to segment the same object. Therefore, a segmentation prompted by a box is usually sharper than a segmentation prompted by a point. In some cases where the object to segment isn't very clear, we would need more than one point to segment the object captured by one box, resulting in higher EfficientViT mask decoder latency. As such, we experimentally determined at least 32 points per frame (uniformly distributed along the gaze vector) are needed to achieve reasonable segmentation results. As each prompt corresponds to one run of the decoder, 32+ runs of the decoder very quickly will lead to the decoder becoming GazeSAM's bottleneck and preventing real-time performance. In comparison, the range of box prompts passed into the encoder very rarely exceeds six (with a median of around two). Beyond the time it takes to complete a run of the decoder, when more masks are outputted, more masks must be processed. As each run produces multiple masks, there is a much higher computational overhead associated with the post-processing stage. More latency analysis will be conducted in Chapter 4. From both a latency and quality of segmentation perspective, it makes more sense to pass in a filtered set of bounding boxes instead of a series of points lying on the gaze vector.

Chapter 4

Latency Optimizations and Results

4.1 Inference Engines

One of the most common ways to achieve speedup without changing the underlying model architecture or making algorithmic adjustments is to use a runtime optimized for inference. Two common options are ONNXRuntime [20] and TensorRT.

One use of ONNXRuntime is creating an accelerated runtime environment for models in ONNX format. ONNX (Open Neural Network Exchange) is a common denominator file format that can help translate models implemented using different ML frameworks like PyTorch, TensorFlow, and Caffe2 into one model format. ONNXRuntime supports a wide range of execution providers including CPU, CUDA, TensorRT, and TVM. A strength of ONNXRuntime is that it's easy to get working - all that needs to be done is to convert the model to ONNX format and specify its execution provider (CUDA or CPU, for example). The conversion to ONNX format involves defining the expected shapes of the model and defining the actions that occur during a forward pass of the model. To improve performance while running ONNXRuntime with a GPU, `IOBinding` can be employed. `IOBinding` improves data transfer time by pre-allocating input/output memory on the target devices/ preventing needless data copy between devices. We employ `IOBinding` when benchmarking performance in Table 4.1.

TensorRT is a library designed specifically to accelerate inference on NVIDIA GPUs. To take advantage of this library, we can convert an ONNX model into a TensorRT engine.

This compilation process includes fusing layers, removing unused operations, fixing model size, and determining which kernel to use for each operation given the specific hardware and kernel options available. TensorRT also supports post-training quantization (PTQ), which we will discuss in Section 4.2.

Model	PyTorch	ONNX	TensorRT
Face detection	0.026	0.012	0.00108
Gaze estimation	0.329	0.020	0.00473
Image encoder	0.059	0.052	0.01254
Mask decoder	0.056	0.029	0.0074
Object detection	0.375	0.052	0.01107
Depth estimation	0.233	0.118	0.03730

Table 4.1: **Model latencies using PyTorch, ONNX, and TensorRT runtimes**. Measured on Jetson AGX Orin, FP32.

Table 4.1 displays each model's average inference time across all three runtimes. Times are calculated by finding the median over 250 frames of video run on a 64GB Jetson AGX Orin excluding pre/postprocessing but including data transfer time. PyTorch CUDA `synchronize` statements fence each timing measurement. The face detection model is a component of `ProxylessGaze`. The gaze estimation model is a component of `L2CS-Net`. The image encoder and mask decoder are components of EfficientViT's `l0` model. We use the `YOLO-NAS-M` object detection model and the `Depth-Anything-vitb14` depth estimation model. Cases where a model isn't run for that specific frame (for example, the EfficientViT model isn't run if the no bounding boxes post-filtering) are not factored into the average. The mask decoder is run on a median of two box prompts, meaning that there are usually two runs of the decoder. The reason we can't combine all boxes into one batch is because all prompts part of a single batch go toward segmenting the same object. We aim to segment one object per bounding box with the EfficientViT-SAM model. Therefore, we need multiple input batches.

Looking at ONNX results relative to the PyTorch baseline, it's clear that ONNX provides a performance boost. We implement ONNX mode using IOBinding - this preallocates memory on the GPU and speeds up the data transfer times. The usage of IOBinding relative to running without IOBinding typically improves results by a of couple milliseconds per frame.

We notice that the image encoder offers the lowest speedup (1.1x), while the gaze estimation model has the largest speedup - 16.5x. The mask decoder, depth estimation model, face detection model, and object detection model have speedups of 1.9x, 2.0x, 2.2x, and 7.2x, respectively. We suspect the range in speedup values across models can be attributed to differences in layer implementation and support between the two runtimes.

TensorRT timing results are recorded as reported by `trtexec` with FP32 weight/activation precision. TensorRT typically results in a 3-5x speedup relative to ONNX results (while remaining in FP32 precision) The reason additional speedup is achieved when running with TensorRT is that the kernel operations are specialized for the specific GPU whereas ONNX models are not specifically tuned to NVIDIA GPUs.

The first three rows of Figure 4.1 illustrate the speedup achieved when comparing the sum of model latencies gained by changing runtimes. TensorRT's 14.6x speedup vs. ONNX's 3.8x speedup motivates using TensorRT for real-time applications. The next logical step is to explore TensorRT-enabled quantization.

4.2 Quantization

Quantization is a technique where the number of bits used to store a model's weights and activations are reduced, decreasing the model's memory footprint and latency while preserving accuracy. In an ideal world, halving the precision halves the latency. As such, we hope to move from FP32 (32-bit floating point) precision for weights/ activations to FP16 or INT8.

We will focus on the post-training quantization (PTQ) workflow. TensorRT employs symmetric quantization - we either divide (quantization) or multiply (dequantization, round, clamp) by the scaling factor to go from one state to the other. To execute the quantization process without training, we must find each activation's typical distribution of values and use this to estimate the quantization scale factor. TensorRT uses this process, called calibration, to create a cache mapping each layer to its scale factor. When TensorRT goes to compile the engine, if it determines the lower precision implementation is faster and the calibration cache contains a scale factor, it sets the layer to the quantized precision.

We explore both FP16 and INT8 quantization in hopes of achieving larger speedups with

minimal performance degradation. We create a calibration cache for each model that goes that undergoes INT8 quantization (FP16 quantization doesn't require a calibration cache). TensorRT guidelines state around 500 images are sufficient for ImageNet calibration and around 5k images are sufficient for Google BERT. As such, we err on the side of caution and use 5k calibration images. Our calibration dataset is simply 5k frames of video we would expect to be representative of typical input to the GazeSAM engine - different headposes, gaze vectors, and objects are present within the dataset. We pass in all 5k images in one batch to the calibrator. This is because scale factors are based on the activations we observe per batch, and smaller batches make it more likely to encounter an outlier that skews the scale factor. The larger and therefore more representative the batch is of the data, the better the quantization result.

Finally, we explore whether quantization results differ when the quantization cache creation algorithm differs. To study this, we consider both the `IINT8EntropyCalibrator2` and `IINT8MinMaxCalibrator`. `IINT8EntropyCalibrator2` is the default calibrator and recommended for CNN-based networks, while `IINT8MinMaxCalibrator` is more suited for NLP tasks like BERT.

The EfficientViT decoder is compiled with minimum, optimal, and max batch sizes of 1, 2, and 10. While the decoder during inference typically takes in a varying number of box prompts, when generating the calibration cache, we pass in a single box prompt as we're unable to process different-size inputs within the same batch.

Model	FP32	FP16	INT8
Face detection	1.08	0.95	0.91
Gaze estimation	4.73	2.64	1.53
Image encoder	12.54	5.65*	4.54†
Mask decoder	7.43	3.87	4.58
Object detection	11.07	6.05	3.89
Depth estimation	37.30	22.38	56.04

Table 4.2: **TensorRT model latencies (ms).** Measured on a Jetson AGX Orin across FP32, FP16, and INT8 precision.

Results across FP32 precision, FP16 precision, and INT8 quantization are shown in Table 4.2. FP16 results come from using the FP16 flag, while INT8 results are averaged across

using the entropy and minmax calibration algorithms. The † indicates a result where there is a major drop in the quality of the model outputs, while a * indicates that visual differences between the FP32 model and quantized model are noticeable but don't severely change the output. In the INT8 column, because results are averaged across both calibration algorithms, the value is only marked if both calibration algorithms produce results that don't match the FP32 version (as switching to the other calibration algorithm fixes the accuracy degradation). A couple of general observations - quantizing from FP32 to FP16 results in a speedup while quantizing from FP16 to INT8 results in lowered latency for two-thirds of the models. While there isn't a latency difference between the models generated using the two INT8 calibration algorithms, as previously mentioned, there is occasionally an accuracy difference.

Looking at the face detection engine, we notice that there we don't achieve much speedup between the different precisions, perhaps due to the model's smaller size (314k parameters). One thing to note is the entropy calibration algorithm results in no face segmentation while using the minmax entropy calibration algorithm results in accuracy preservation. This is interesting because the entropy algorithm is advertised to typically be more compatible with CNN-based networks and the minmax algorithm more compatible with NLP/BERT-like tasks and models, but the face detection model uses a conv-based architecture and performs much better with the minmax algorithm.

The gaze estimation engine experiences a 1.8x speedup from FP32 to FP16, then a 1.7x speedup from FP16 to INT8. All quantization attempts are successful, but it is interesting to note that the entropy-based INT8 quantization has slightly better accuracy than the minmax-quantized model. I hypothesize this is because the CNN-based architecture of the gaze estimation model is, as advertised, better suited for the entropy-based calibration method.

For the EfficientViT image encoder, we see that quantizing the model from FP32 to FP16 results in over a 2x speedup with a slight decrease in performance. However, further quantization from FP16 to INT8 results in very poor mask segmentation quality. We suspect the transformer architecture within the model poses difficulty for the TensorRT compiler to try and quantize correctly. In terms of the decoder, we achieve a 1.8x speedup moving from FP32 to FP16, but latency increases when we perform INT8 quantization. We hypothesize this may be due to the TensorRT compiler finding some layers unsuitable for quantization

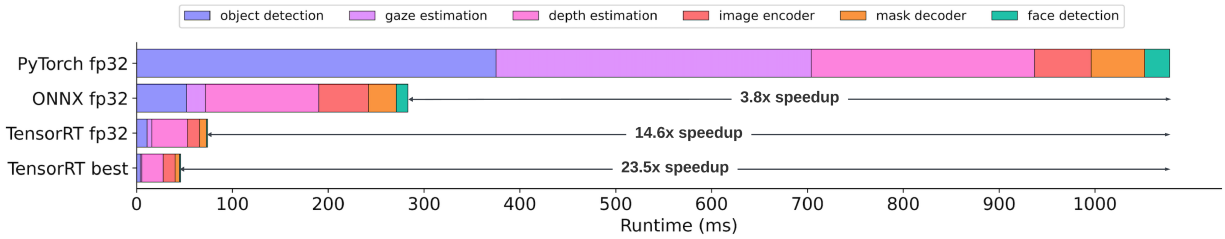

Figure 4.1: **Model speedups across runtimes and quantization.** "Best" in row four refers to using the fastest model components that still preserve accuracy.

and thus choosing an FP32 equivalent, leading to an overall latency slowdown.

The YOLO object detection model experiences a 1.8x speedup moving from FP32 to FP16, and moving from FP16 to INT8 results in another 1.6x latency reduction. We suspect the calibration cache is the reason for accuracy improvement - the access to the activation range when presented with similar inputs allows for the scale factor to better reflect the original weight and activation values. This aligns with YOLO-NAS's advertising of the model's quantizability.

Depth-Anything's quantization from FP32 to FP16 results in over a 1.7x speed-up while preserving accuracy, but trying to quantize the model to INT8 is unsuccessful. With both calibration algorithms, latency shoots up to above FP32 levels. Interestingly, while the entropy calibration algorithm's outputs match that of a model generated with FP32 precision, the minmax calibration algorithm results in very poor depth estimation capabilities. We hypothesize the increase in latency going from FP32 to INT8 is the TensorRT calibrator finding that the quantized versions of the layers result in too large of an accuracy decrease, so alternate kernel operations are chosen. Perhaps due to the nature of INT8 quantization, by trying to optimize certain layers to INT8 and moving the majority back to FP32, overall latency relative to the FP16 version increases.

Figure 4.1 summarizes the result of runtime and quantization optimizations. The numbers within the "TensorRT best" category come from taking the fastest model that doesn't experience accuracy degradation. Thus, the results are a mix of FP32 (EfficientViT image encoder), FP16 (depth estimation model, EfficientViT image decoder), and INT8 (all other models). TensorRT quantization results in a 1.6x speedup compared to using FP32 engines.

4.3 Algorithmic Optimizations

Beyond the time it takes to run the models that comprise GazeSAM, the majority of our time is spent as follows: (1) preprocessing/postprocessing, (2) filtering the original set of bounding boxes using the gaze vector and depth estimation mask, (3) determining the final bounding box/ segmentation and annotating the image, and (4) writing the annotated frame back to the `VideoWriter`.

Preprocessing and Postprocessing: Two examples of latency optimizations made to the pre/postprocessing stage of the pipeline include removing steps from preprocessing and improving data transfer time. The default preprocessing that comes with the Depth-Anything model includes normalizing each channel of the image using a constant mean and standard deviation, then scaling the values to lie between 0 and 1. However, we found removing normalization in the preprocessing stage still results in accurate depth estimation masks. Removing this step and then scaling post-image-downsize shaves around 9ms off the pipeline latency, which is significant considering that running in real-time results in an average of 40ms/frame. In terms of decreasing data transfer time, because a multimask output is of shape (720, 1280), the longer the mask can remain on the GPU and the more filtering operations that can be done there, the better. Keeping the masks on the GPU during processing and only moving to the CPU for mask rendering also results in a couple-millisecond decrease in latency.

Bounding Box Filtering: In order to determine whether a bounding box falls on the line of the gaze, we first must determine the endpoint of the gaze vector; that is, where it intersects the edge of the image. The latency of this task is under 0.2ms. To determine the intersection of the gaze vector with a bounding box, there are two approaches: find the intersection of a gaze vector mask with a bounding box mask, or use geometry to determine if and where the gaze vector intersects with a box edge. The former strategy is much easier to implement (a simple `AND` operation between two masks) but is more computationally intensive due to the size of the masks and the overhead of aggregating all the points along the gaze line. Instead, we check for intersections by determining whether the gaze vector intersects any bounding box edges. Switching to this method saves around 30ms/frame.

Mask Segmentation Filtering: As mentioned in Section 3.1, we determine the best mask and bounding box based on which mask occupies the largest percentage of its corresponding bounding box. This part of the algorithm is inherently slow - we must sum all values of a binary mask, and as masks can get large, latency can quickly accumulate. The reason why this part of the algorithm still remains so efficient is that we typically have to deal with only a couple of bounding boxes/ masks. The reason for this is due to the design decision of filtering the bounding boxes with the gaze vector and depth mask before generating image segmentations. As a result, best mask determination and rendering usually takes less than 4ms.

Frame Writing: If we choose to save the GazeSAM output, we will have to write the frame back to the VideoWriter. This operation is surprisingly costly for frame sizes of around (720, 1080) - writing takes over 20ms per frame on the AGX Orin. However, we find that if the frame written back is the default size for the VideoWriter - (640, 480) - writing the image takes less than 1 ms. While running in webcam mode, we apply this optimization, but when run with input video, we process the frames without resizing down to (640, 480).

Removing Landmark Detection: Another latency optimization made was using the center of the face bounding box as the head of the gaze vector, thereby eliminating the need to run a landmark detection algorithm to generate the eye position along with other unused facial landmarks.

General Strategies: Generally speaking, to keep latency down, we take steps to ensure that operations are vectorized and expensive operations are pushed downstream after we've typically downsized images. When confronted with two Numpy or Pytorch functions that can both be used for a specific task, we integrate the one that performs slightly faster.

4.4 GazeSAM Results

We create a `cv2`-based demo where each frame from the webcam is read, processed, and outputted in real-time. On an RTX 4070 GPU, the average speed is around 30 FPS. We reach around 25 FPS when an object is being segmented and 80 FPS when no object is segmented. Figure 4.2 shows a few examples of segmentations mid-video of the GazeSAM

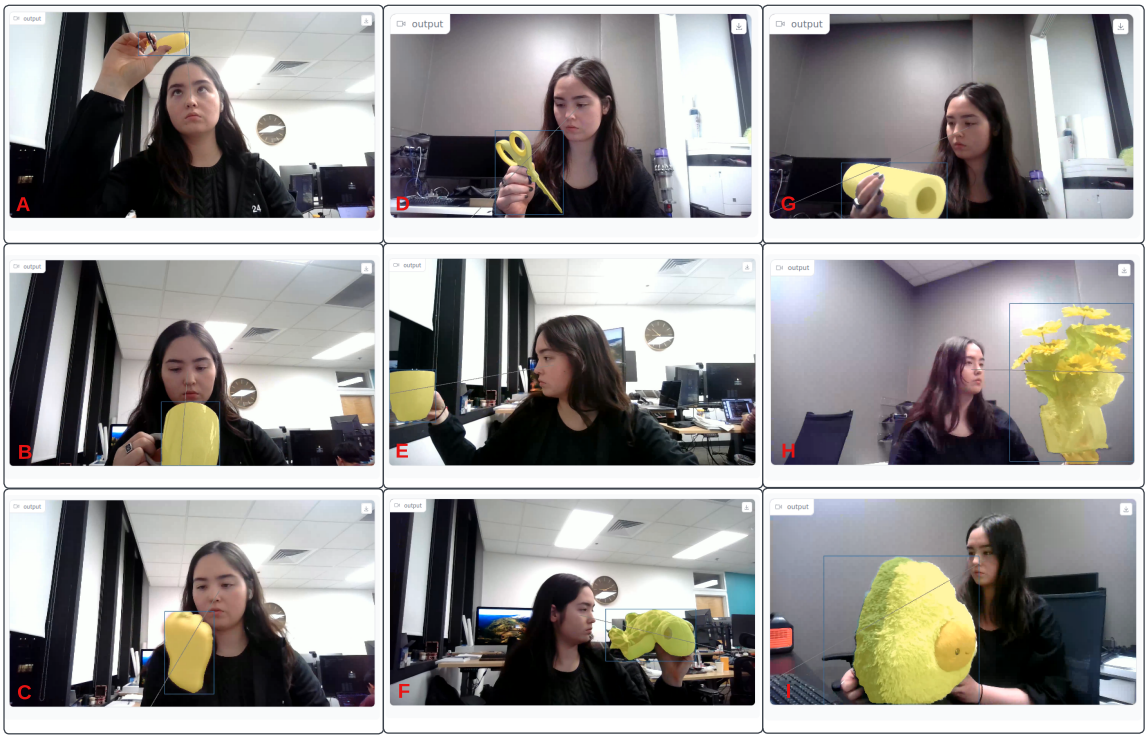

Figure 4.2: **GazeSAM.** Gaze-prompted segmentation results captured mid-video are shown in yellow.

demo. The intent here is to demonstrate the adaptability of the GazeSAM model to perform correct segmentations across a variety of headposes, objects, and background complexity. **A** demonstrates segmenting a small object at an upward angle. **B** and **E** demonstrate a cup being segmented from different angles - both the downward and sideways angles show less of the eye/ front section of the face, making it harder for the model to make a good prediction of the gaze angle. However, the ProxylessNAS face detection and L2CS-Net gaze estimation models do a good job of covering the edge cases of gaze prediction. **C**, **G**, and **I** segment a hand, paper towel roll, and a stuffed animal, respectively, demonstrating the range of objects that can be detected by the model. **D** and **H** demonstrate the crispness of the segmentations even when the object shapes are irregular. **F** demonstrates the depth-awareness of GazeSAM - the background behind the turtle along the line of the gaze contains other objects like the monitor and desk, but only the turtle is segmented.

4.5 Web Interface

We also create a Gradio [21] web interface to enable interaction with the EfficientViT-SAM models. The models available are `l0`, `l1`, `l2`, `xl0`, and `xl1`, where `l0` is the fastest and `xl1` is the most accurate. We enable three runtime environments - PyTorch, ONNX, and TensorRT - to accommodate a wide range of operating platforms and latency expectations. The PyTorch implementation provides for a widely usable runtime across different device types, while ONNX is used to accelerate inference on CPUs. TensorRT is our fastest runtime and is used to accelerate inference on NVIDIA GPUs. We support four prompt formats: point-prompted, box-prompted, box- and point-prompted, along with full image segmentation. The interface supports built-in examples, user uploads, and webcam functionality as modes for image input. Results are illustrated in Fig 4.3.

Row 1 of Fig 4.3 shows the demo before adding image input. The top of the image displays the four tabs corresponding to the four segmentation modes. There are three ways to add an input image to the demo - clicking on a pre-populated example from the image example bar (bottom of Row 1), uploading an image (click within the input box), or using the webcam (click the circular icon slightly under the input box).

Row 2 shows the resulting segmentation from the point segmentation and box segmentation modes. The image on the left shows the result of point-segmentation mode. As we can tell, a single point is enough to crisply segment the image. The right image shows box-segmentation mode. Each box prompt corresponds to a different object segmented - a key difference between point and box mode. Under the hood, we call a single run of the decoder with all of the points in point mode, but we do one run of the decoder per box in box mode. All points go toward segmenting one image, whereas each box goes toward segmenting a different object.

Row 3 shows the result of single-box multiple-point (SBMP) mode where our segmentation inputs are constrained to at most one box and an uncapped number of points. All prompt inputs are processed in a single run of the decoder and go toward segmenting the same image. Note the difference between the image on the left and the right - the left image has parts of the pebbles segmented (undesirable), while the right image has the addition of

three points (in pink) acting as negative point prompts over the previously segmented pebble region, resulting in a clean segment of just the bear.

Row 4 shows the results of full image segmentation. Four parameters alter segmentation granularity: number of guiding points, IoU threshold, stability score threshold, and box NMS threshold. The number of guiding points controls the number of input prompts to the mask decoder model. In full image segmentation mode, these points are evenly scattered over the image. Therefore, increasing the number of points generally increases the mask granularity. The IoU threshold parameter filters out masks whose IoU (intersection over union) is smaller than a user-determined threshold. The stability score threshold measures how much results change with a small shift in the input, and results less than the threshold are removed. Finally, overlapping masks are removed with batched NMS (non-maximal suppression) filtering.

Looking back at Figure 4.3, the leftmost image in Row 4 shows segmentation results using the default segmentation parameters. The middle image increases the IOU threshold from 0.80 to 0.84. Note the resulting change in granularity (the building on the right edge of the frame is now segmented by a single mask) as well as the number of masks (the road/ sidewalk is no longer segmented). Finally, the rightmost image is an example of webcam input functionality using the default segmentation parameters.

4.5.1 Custom Components

Building this demo requires components that support image prompting. We create four components - `PointPromptableImage`, `BoxPromptableImage`, `SBMPPromptableImage`, and `ClickableArrowDropdown` - to enable this functionality.

The `ClickableArrowDropdown` component simply enables a UI change: the default Gradio dropdown has an unclickable dropdown expansion arrow, while the intuitive user behavior is to click the arrow to display the contents. This custom component fixes that.

These three custom components are built off of PhyscalX's `GradioImagePrompter` custom component. There are a couple of reasons we do not use the `GradioImagePrompter` component:

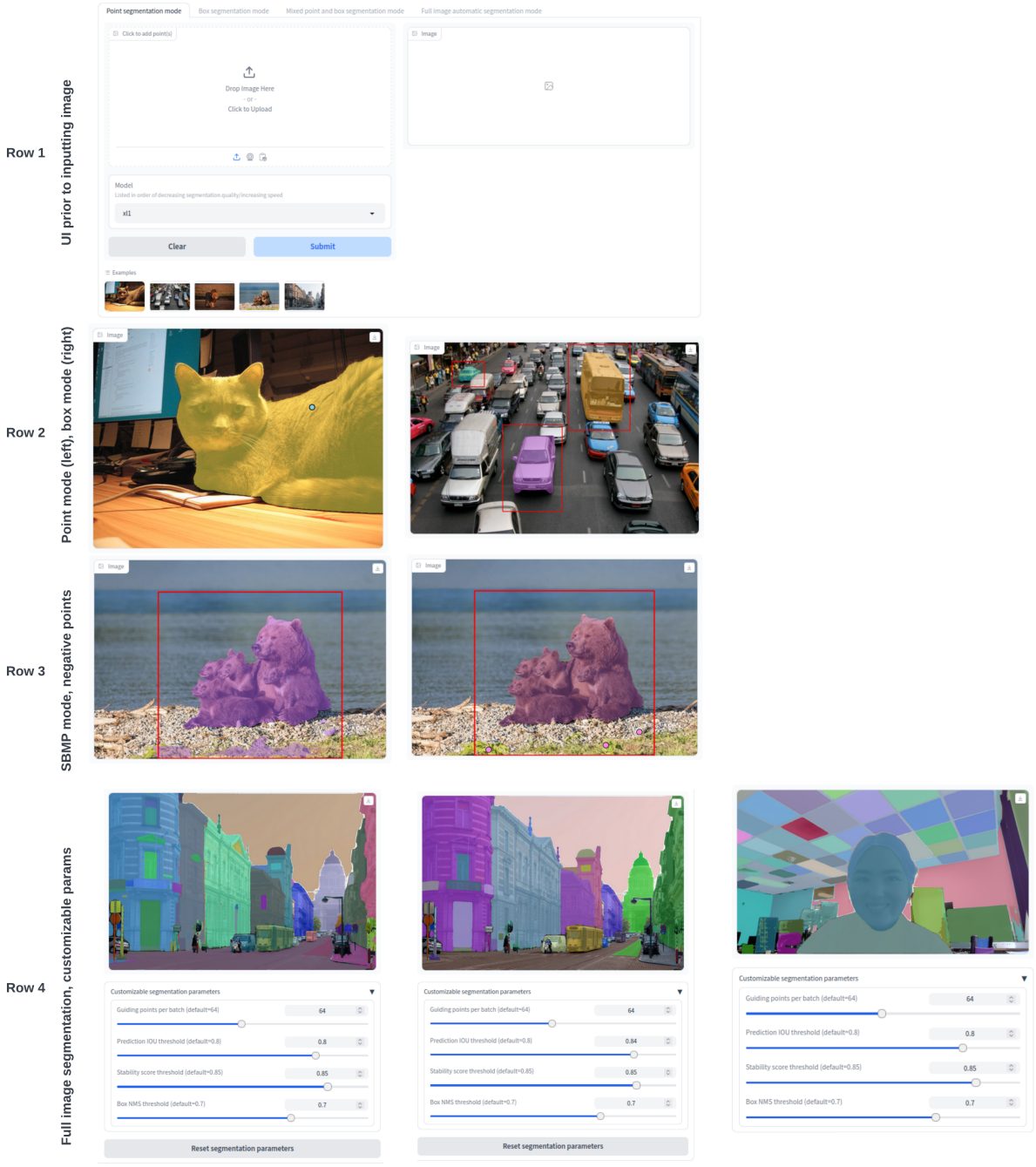

Figure 4.3: **Web Interface.** Row 1 shows the demo interface before image upload. Rows 2-4 display results produced by the different prompt types.

1. We do not want to conflate different prompt types; i.e. use a component that allows both point and box prompting when we're trying to demo just a point-promptable (and not box-promptable) image.

2. We want to support webcam functionality.

3. We'd like to make additional UI enhancements.

As such, our `PointPromptableImage` custom component allows for any number of point prompts but doesn't respond to box prompt actions. The `BoxPromptableImage` component works precisely in the opposite direction. `SMBPPromptableImage` allows for at most one box and any number of point prompts. All three of these components used Gradio's own `Image` component as a template. Each custom component is made up of a frontend and backend. The frontend is powered by Svelte, a front-end framework. The backend uses the Node.js framework. Changes need to be made to both the frontend and backend to control the component's appearance, handle user interaction, and parse image uploads. We also include buttons to "undo" the previous prompt, "clear" all prompts prior, and "delete" both the image and prompt.

Chapter 5

Conclusion and Future Work

GazeSAM introduces gaze as a novel prompt type for image segmentation tasks. This work explores the components and design decisions behind GazeSAM, enabling real-time performance through algorithmic, runtime, and quantization-based optimizations. Furthermore, we open-source the custom components that can be broadly used across all promptable SAM-based models. Our custom components support point prompts, box prompts, gaze prompts, mixed prompt types, and full image segmentation. Future work can explore ways to improve segmentation accuracy and increase speed. Integrating object tracking may help to improve object detection accuracy during swift movements. We can explore implementing GazeSAM in C as well as performing quantization-aware training to further reduce latency.

Appendix A

Resources

- `https://github.com/mit-han-lab/efficientvit/tree/master/demo/gazesam`
  - GazeSAM implementation and demo.

- `https://github.com/mit-han-lab/efficientvit/tree/master/demo/sam`
  - EfficientViT web interface.

- `https://github.com/ncstiles/gradio-point-promptable-image`
  - Custom component to enable image prompting with positive and negative points.
  - Available on PyPI: https://pypi.org/project/gradio-point-promptable-image

- `https://github.com/ncstiles/gradio-box-promptable-image`
  - Custom component to enable image prompting with bounding boxes.
  - Available on PyPI: https://pypi.org/project/gradio-box-promptable-image

- `https://github.com/ncstiles/gradio-sbmp-promptable-image`
  - Custom component to enable image prompting with at most one bounding box and any number of points.
  - Available on PyPI: https://pypi.org/project/gradio-sbmp-promptable-image

- `https://github.com/ncstiles/gradio-clickable-arrow-dropdown`
  - Custom component to make dropdown arrow of the component clickable
  - Available on PyPI: https://pypi.org/project/gradio-clickable-arrow-dropdown

References

[1] A. Kirillov, E. Mintun, N. Ravi, *et al.*, *Segment anything*, 2023. arXiv: 2304.02643 [cs.CV].

[2] H. Cai, J. Li, M. Hu, C. Gan, and S. Han, *Efficientvit: Multi-scale linear attention for high-resolution dense prediction*, 2023. arXiv: 2205.14756 [cs.CV].

[3] Y. Xiong, B. Varadarajan, L. Wu, *et al.*, *Efficientsam: Leveraged masked image pre-training for efficient segment anything*, 2023. arXiv: 2312.00863 [cs.CV].

[4] C. Zhang, D. Han, Y. Qiao, J. U. Kim, S.-H. Bae, S. Lee, and C. S. Hong, *Faster segment anything: Towards lightweight sam for mobile applications*, 2023. arXiv: 2306.14289 [cs.CV].

[5] B. Wang, A. Aboah, Z. Zhang, and U. Bagci, *Gazesam: What you see is what you segment*, 2023. arXiv: 2304.13844 [cs.CV].

[6] H. Cai, L. Zhu, and S. Han, *Proxylessnas: Direct neural architecture search on target task and hardware*, 2019. arXiv: 1812.00332 [cs.LG].

[7] A. A. Abdelrahman, T. Hempel, A. Khalifa, and A. Al-Hamadi, *L2cs-net: Fine-grained gaze estimation in unconstrained environments*, 2022. arXiv: 2203.03339 [cs.CV].

[8] Deci, *Yolo-nas by deci achieves state-of-the-art performance on object detection using neural architecture search*, 2024. URL: https://deci.ai/blog/yolo-nas-object-detection-foundation-model/.

[9] L. Yang, B. Kang, Z. Huang, X. Xu, J. Feng, and H. Zhao, *Depth anything: Unleashing the power of large-scale unlabeled data*, 2024. arXiv: 2401.10891 [cs.CV].

[10] Y. Li, H. Mao, R. Girshick, and K. He, *Exploring plain vision transformer backbones for object detection*, 2022. arXiv: 2203.16527 [cs.CV].

[11] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, *An image is worth 16x16 words: Transformers for image recognition at scale*, 2021. arXiv: 2010.11929 [cs.CV].

[12] M. Kurniawan and H. Hara, *A beginner's guide to frame rates in movies*. URL: https://www.adobe.com/creativecloud/video/discover/frame-rate.html.

[13] X. Zhao, W. Ding, Y. An, Y. Du, T. Yu, M. Li, M. Tang, and J. Wang, *Fast segment anything*, 2023. arXiv: 2306.12156 [cs.CV].

[14] H. Balim, S. Park, X. Wang, X. Zhang, and O. Hilliges, *Efe: End-to-end frame-to-gaze estimation*, 2023. arXiv: 2305.05526 [cs.CV].

[15] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, 2014. arXiv: 1311.2524 [cs.CV].

[16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You only look once: Unified, real-time object detection*, 2016. arXiv: 1506.02640 [cs.CV].

[17] R. Birkl, D. Wofk, and M. Müller, *Midas v3.1 – a model zoo for robust monocular relative depth estimation*, 2023. arXiv: 2307.14460 [cs.CV].

[18] M. Oquab, T. Darcet, T. Moutakanni, *et al.*, *Dinov2: Learning robust visual features without supervision*, 2024. arXiv: 2304.07193 [cs.CV].

[19] R. Ranftl, A. Bochkovskiy, and V. Koltun, *Vision transformers for dense prediction*, 2021. arXiv: 2103.13413 [cs.CV].

[20] O. R. developers, *Onnx runtime*, https://onnxruntime.ai/, Version: x.y.z, 2021.

[21] A. Abid, A. Abdalla, A. Abid, D. Khan, A. Alfozan, and J. Zou, *Gradio: Hassle-free sharing and testing of ml models in the wild*, 2019. arXiv: 1906.02569 [cs.LG].