AIL 862

Lecture 21

Code – Sam

```python
class Sam(nn.Module):
    mask_threshold: float = 0.0
    image_format: str = "RGB"

    def __init__(
        self,
        image_encoder: ImageEncoderViT,
        prompt_encoder: PromptEncoder,
        mask_decoder: MaskDecoder,
        pixel_mean: List[float] = [123.675, 116.28, 103.53],
        pixel_std: List[float] = [58.395, 57.12, 57.375],
    ) -> None:
        """
        SAM predicts object masks from an image and input prompts.

        Arguments:
          image_encoder (ImageEncoderViT): The backbone used to encode the
            image into image embeddings that allow for efficient mask prediction.
          prompt_encoder (PromptEncoder): Encodes various types of input prompts.
          mask_decoder (MaskDecoder): Predicts masks from the image embeddings
            and encoded prompts.
          pixel_mean (list(float)): Mean values for normalizing pixels in the input image.
          pixel_std (list(float)): Std values for normalizing pixels in the input image.
        """
```

```
@torch.no_grad()
def forward(
    self,
    batched_input: List[Dict[str, Any]],
    multimask_output: bool,
) -> List[Dict[str, torch.Tensor]]:
    """
    Predicts masks end-to-end from provided images and prompts.
    If prompts are not known in advance, using SamPredictor is
    recommended over calling the model directly.

    Arguments:
      batched_input (list(dict)): A list over input images, each a
        dictionary with the following keys. A prompt key can be
        excluded if it is not present.
          'image': The image as a torch tensor in 3xHxW format,
            already transformed for input to the model.
          'original_size': (tuple(int, int)) The original size of
            the image before transformation, as (H, W).
          'point_coords': (torch.Tensor) Batched point prompts for
            this image, with shape BxNx2. Already transformed to the
            input frame of the model.
          'point_labels': (torch.Tensor) Batched labels for point prompts,
            with shape BxN.
          'boxes': (torch.Tensor) Batched box inputs, with shape Bx4.
            Already transformed to the input frame of the model.
          'mask_inputs': (torch.Tensor) Batched mask inputs to the model,
            in the form Bx1xHxW.
      multimask_output (bool): Whether the model should predict multiple
        disambiguating masks, or return a single mask.
```

```
Returns:
  (list(dict)): A list over input images, where each element is
    as dictionary with the following keys.
      'masks': (torch.Tensor) Batched binary mask predictions,
        with shape BxCxHxW, where B is the number of input prompts,
        C is determined by multimask_output, and (H, W) is the
        original size of the image.
      'iou_predictions': (torch.Tensor) The model's predictions
        of mask quality, in shape BxC.
      'low_res_logits': (torch.Tensor) Low resolution logits with
        shape BxCxHxW, where H=W=256. Can be passed as mask input
        to subsequent iterations of prediction.
"""
```

```python
    input_images = torch.stack([self.preprocess(x["image"]) for x in batched_input], dim=0)
    image_embeddings = self.image_encoder(input_images)

    outputs = []
    for image_record, curr_embedding in zip(batched_input, image_embeddings):
        if "point_coords" in image_record:
            points = (image_record["point_coords"], image_record["point_labels"])
        else:
            points = None
        sparse_embeddings, dense_embeddings = self.prompt_encoder(
            points=points,
            boxes=image_record.get("boxes", None),
            masks=image_record.get("mask_inputs", None),
        )
        low_res_masks, iou_predictions = self.mask_decoder(
            image_embeddings=curr_embedding.unsqueeze(0),
            image_pe=self.prompt_encoder.get_dense_pe(),
            sparse_prompt_embeddings=sparse_embeddings,
            dense_prompt_embeddings=dense_embeddings,
            multimask_output=multimask_output,
        )
        masks = self.postprocess_masks(
            low_res_masks,
            input_size=image_record["image"].shape[-2:],
            original_size=image_record["original_size"],
        )
        masks = masks > self.mask_threshold
        outputs.append(
            {
                "masks": masks,
                "iou_predictions": iou_predictions,
                "low_res_logits": low_res_masks,
            }
        )
    return outputs
```

Code – image encoder

```python
class ImageEncoderViT(nn.Module):
            LayerNorm2d(out_chans),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.patch_embed(x)
        if self.pos_embed is not None:
            x = x + self.pos_embed

        for blk in self.blocks:
            x = blk(x)

        x = self.neck(x.permute(0, 3, 1, 2))

        return x
```

Code – prompt encoder

If it is point (works similarly for box)

```
def _embed_points(
    self,
    points: torch.Tensor,
    labels: torch.Tensor,
    pad: bool,
) -> torch.Tensor:
    """Embeds point prompts."""
    points = points + 0.5  # Shift to center of pixel
    if pad:
        padding_point = torch.zeros((points.shape[0], 1, 2), device=points.device)
        padding_label = -torch.ones((labels.shape[0], 1), device=labels.device)
        points = torch.cat([points, padding_point], dim=1)
        labels = torch.cat([labels, padding_label], dim=1)
    point_embedding = self.pe_layer.forward_with_coords(points, self.input_image_size)
    point_embedding[labels == -1] = 0.0
    point_embedding[labels == -1] += self.not_a_point_embed.weight
    point_embedding[labels == 0] += self.point_embeddings[0].weight
    point_embedding[labels == 1] += self.point_embeddings[1].weight
    return point_embedding
```

Code – prompt encoder

If it is mask

```
def _embed_masks(self, masks: torch.Tensor) -> torch.Tensor:
    """Embeds mask inputs."""
    mask_embedding = self.mask_downscaling(masks)
    return mask_embedding
```

```
self.mask_downscaling = nn.Sequential(
    nn.Conv2d(1, mask_in_chans // 4, kernel_size=2, stride=2),
    LayerNorm2d(mask_in_chans // 4),
    activation(),
    nn.Conv2d(mask_in_chans // 4, mask_in_chans, kernel_size=2, stride=2),
    LayerNorm2d(mask_in_chans),
    activation(),
    nn.Conv2d(mask_in_chans, embed_dim, kernel_size=1),
)
```

Code - decoder

```python
class MaskDecoder(nn.Module):
        self.iou_prediction_head = MLP(
            transformer_dim, iou_head_hidden_dim, self.num_mask_tokens, iou_head_depth
        )

    def forward(
        self,
        image_embeddings: torch.Tensor,
        image_pe: torch.Tensor,
        sparse_prompt_embeddings: torch.Tensor,
        dense_prompt_embeddings: torch.Tensor,
        multimask_output: bool,
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        """
        Predict masks given image and prompt embeddings.

        Arguments:
          image_embeddings (torch.Tensor): the embeddings from the image encoder
          image_pe (torch.Tensor): positional encoding with the shape of image_embeddings
          sparse_prompt_embeddings (torch.Tensor): the embeddings of the points and boxes
          dense_prompt_embeddings (torch.Tensor): the embeddings of the mask inputs
          multimask_output (bool): Whether to return multiple masks or a single
            mask.

        Returns:
          torch.Tensor: batched predicted masks
          torch.Tensor: batched predictions of mask quality
        """
```

```python
"""
masks, iou_pred = self.predict_masks(
    image_embeddings=image_embeddings,
    image_pe=image_pe,
    sparse_prompt_embeddings=sparse_prompt_embeddings,
    dense_prompt_embeddings=dense_prompt_embeddings,
)

# Select the correct mask or masks for output
if multimask_output:
    mask_slice = slice(1, None)
else:
    mask_slice = slice(0, 1)
masks = masks[:, mask_slice, :, :]
iou_pred = iou_pred[:, mask_slice]

# Prepare output
return masks, iou_pred
```

```python
def predict_masks(
    self,
    image_embeddings: torch.Tensor,
    image_pe: torch.Tensor,
    sparse_prompt_embeddings: torch.Tensor,
    dense_prompt_embeddings: torch.Tensor,
) -> Tuple[torch.Tensor, torch.Tensor]:
    """Predicts masks. See 'forward' for more details."""
    # Concatenate output tokens
    output_tokens = torch.cat([self.iou_token.weight, self.mask_tokens.weight], dim=0)
    output_tokens = output_tokens.unsqueeze(0).expand(sparse_prompt_embeddings.size(0), -1, -1)
    tokens = torch.cat((output_tokens, sparse_prompt_embeddings), dim=1)

    # Expand per-image data in batch direction to be per-mask
    src = torch.repeat_interleave(image_embeddings, tokens.shape[0], dim=0)
    src = src + dense_prompt_embeddings
    pos_src = torch.repeat_interleave(image_pe, tokens.shape[0], dim=0)
    b, c, h, w = src.shape

    # Run the transformer
    hs, src = self.transformer(src, pos_src, tokens)
    iou_token_out = hs[:, 0, :]
    mask_tokens_out = hs[:, 1 : (1 + self.num_mask_tokens), :]

    # Upscale mask embeddings and predict masks using the mask tokens
    src = src.transpose(1, 2).view(b, c, h, w)
    upscaled_embedding = self.output_upscaling(src)
    hyper_in_list: List[torch.Tensor] = []
    for i in range(self.num_mask_tokens):
        hyper_in_list.append(self.output_hypernetworks_mlps[i](mask_tokens_out[:, i, :]))
    hyper_in = torch.stack(hyper_in_list, dim=1)
    b, c, h, w = upscaled_embedding.shape
    masks = (hyper_in @ upscaled_embedding.view(b, c, h * w)).view(b, -1, h, w)

    # Generate mask quality predictions
    iou_pred = self.iou_prediction_head(iou_token_out)
```

```python
self.iou_prediction_head = MLP(
    transformer_dim, iou_head_hidden_dim, self.num_mask_tokens, iou_head_depth
)
```

Application - Zero shot edge detection

One Example Semantic Segmentation

- Just one example image of target class.
- No training phase involved.
- Potential uses.

What We Have?

- One example image
- One query/test image
- SAM model

Use SAM for our problem (ideal version)

- Stitch/concatenate the example and the query/test image.

- Treat the concatenated image as a single image and feed to SAM. Generate prompts using the example image.

- And SAM produces output for the entire concatenated image.

Use SAM for our problem (ideal version)

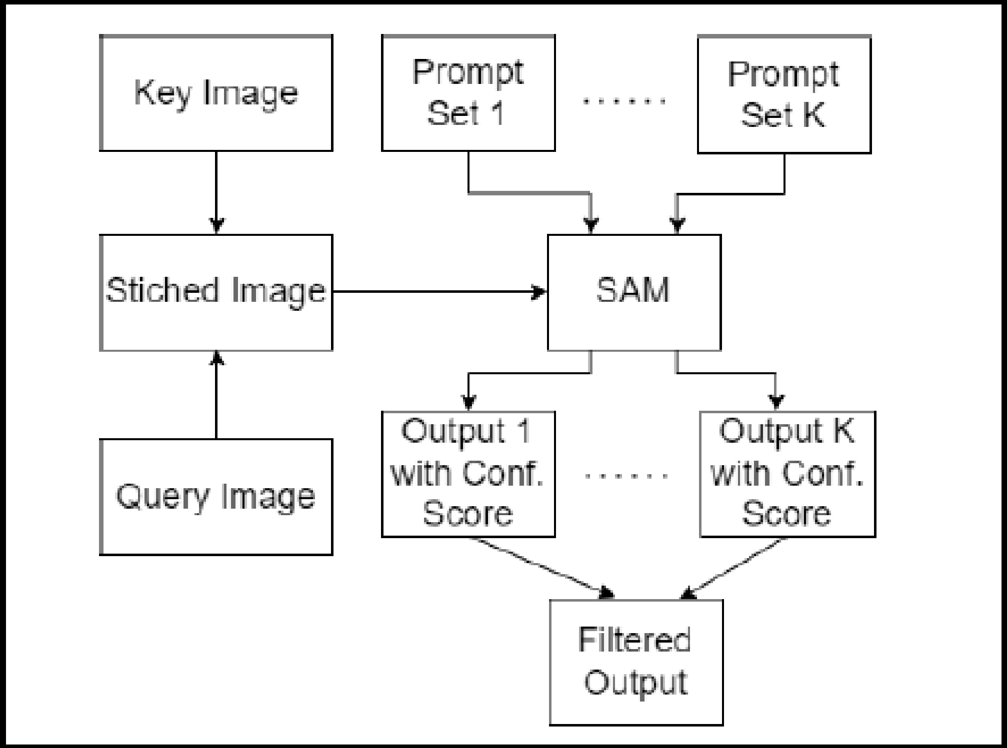
- Does not work.

- Spatial bias.

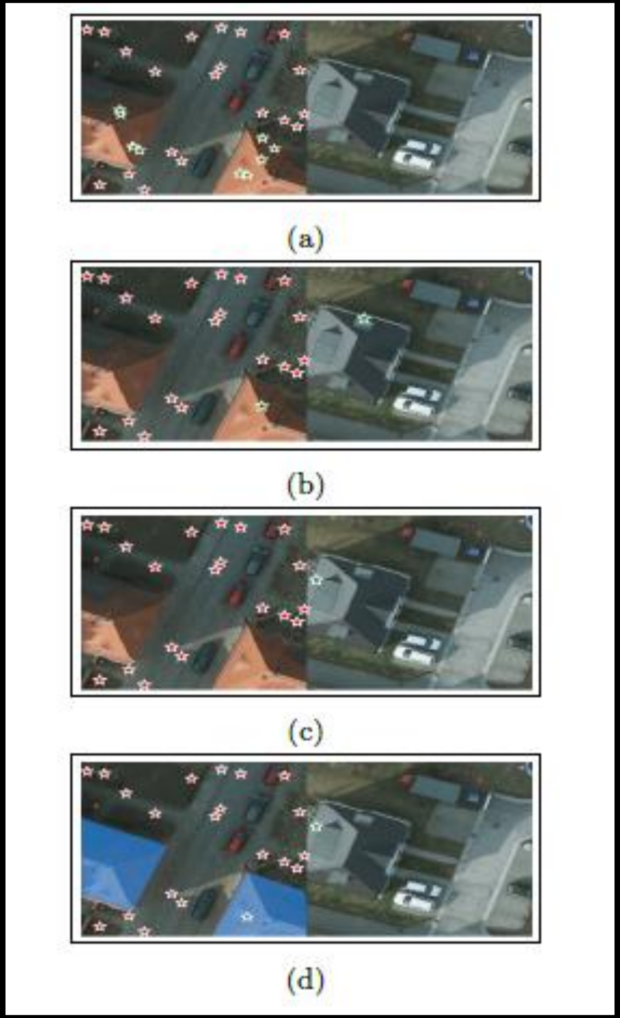
How to Solve

- Generate some positive prompts in the query/test half of the concatenated image as well.
- But how? We do not know the segmentation mask of the concatenated image at all!

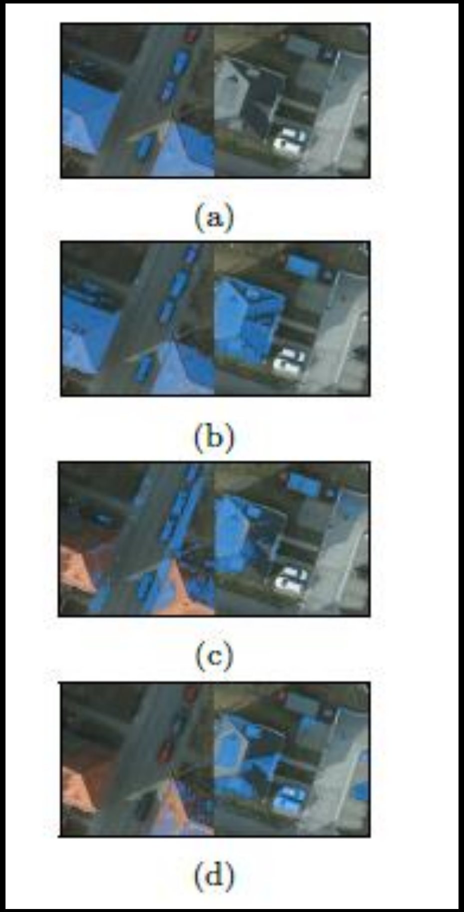
Run Many Times and Filter by Confidence



Four Prompt Design Techniques



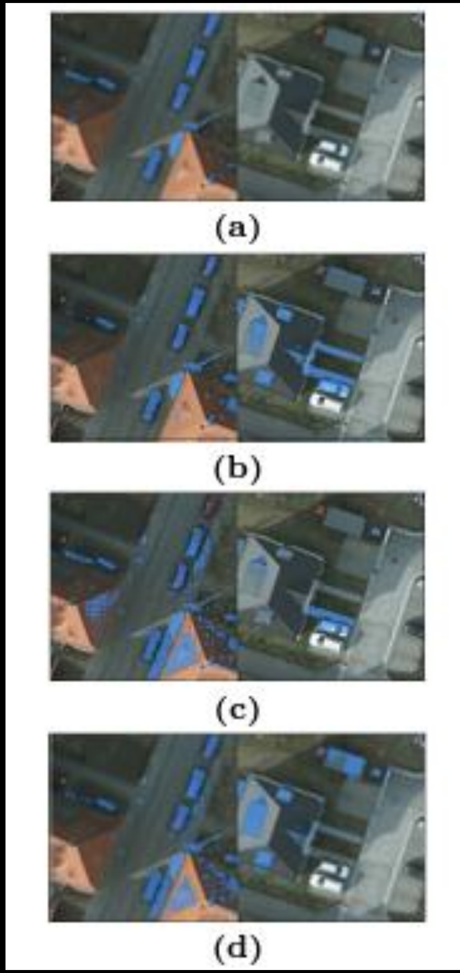
Building Detection Example



Numerical result

Method	IoU
Prompt 1 / SAM	0.002
Prompt 2	0.693
Prompt 3	0.393
Prompt 4	0.480

Vehicle Detection Example


(a)
(b)
(c)
(d)