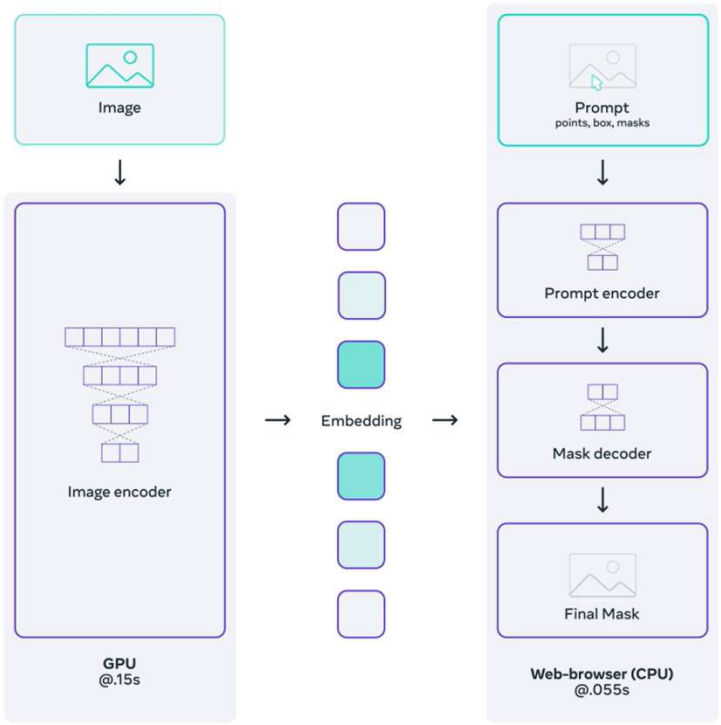
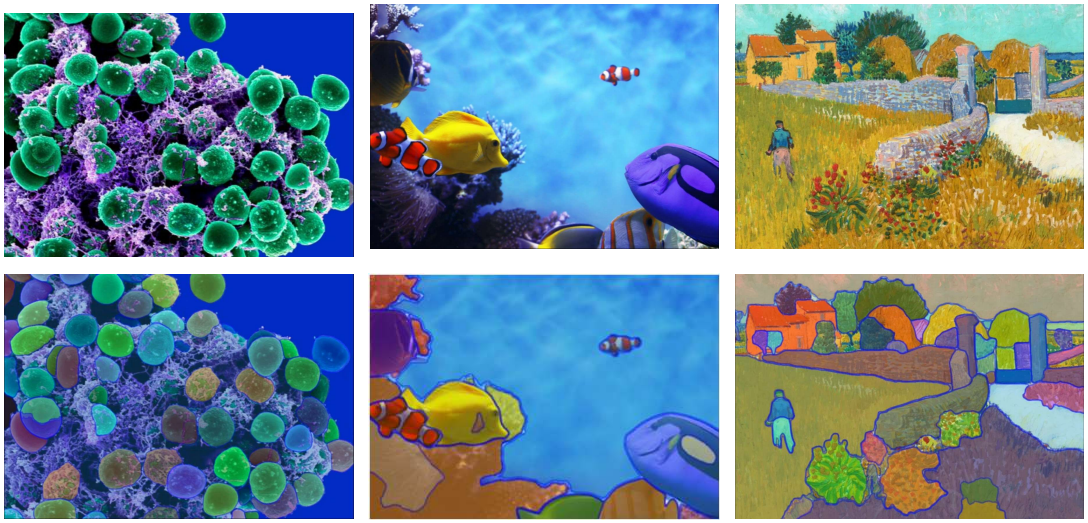


Images taken from https://segment-anything.com/



Segment Anything

Umar Jamil

What is Image Segmentation?

Image segmentation is the process of partitioning a digital image into multiple regions (or segments), such that pixels belonging to the same region share some (semantic) characteristics.

It has applications in many fields:

1. Medical imaging (locate tumors)
2. Object detection (pedestrian detection, object detection in satellite images)
3. Content-based image retrieval (find all images that contain a cat/dog/pizza)
4. …

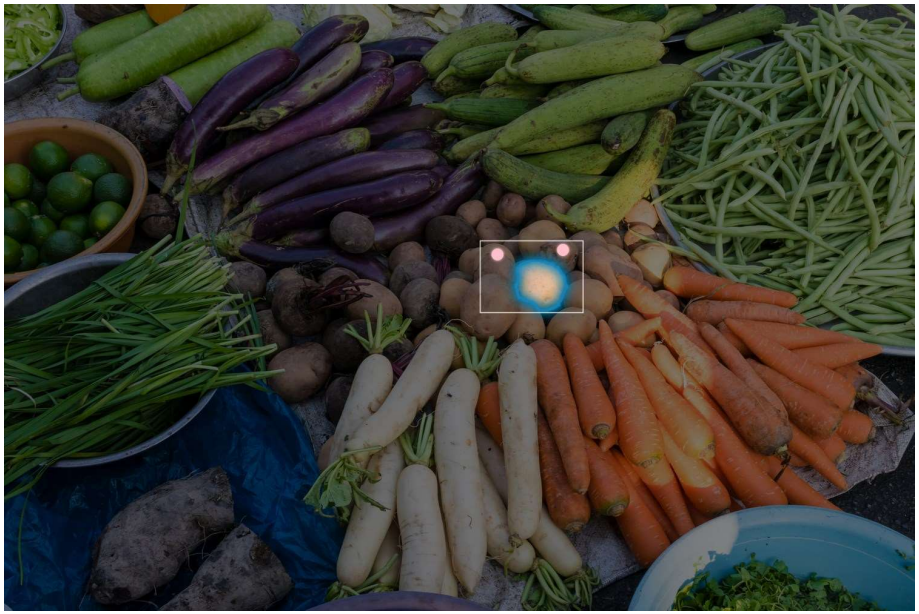
But also has many **challenges**:

1. Difficult and expensive to label datasets (the operator needs to create a pixel-perfect region)
2. Models are usually application-specific (for example only trained for a certain type of medical application, and can't be applied to other fields like pedestrian detection)
3. Previous models usually not prompt-able, that is, we can't tell the model to only segment people, cars or dogs.

Segment Anything

Segment Anything introduces three innovations:

1. Promptable Segmentation Task: allows to find masks given:
    1. Points (for example points selected by the user by mouse click)
    2. Boxes (a rectangle defined by the user)
    3. Text prompt (for example “find all the dogs in this image”)
    4. A combination of the above (for example a box and a background point)
2. Segment Anything Model
    1. A **fast** encoder-decoder model (takes 50ms in a web browser to generate a mask given a prompt)
    2. Ambiguity-aware: for example given a point, it may correspond to multiple masks and the model should return the three most likely (the **part**, the **subpart** and the **whole**).
3. Segment Anything Dataset (and its Segment Anything Engine)
    1. 1.1 billion segmentation masks collected with the Segment Anything Engine
    2. All the masks have been generated automatically! (no human supervision!)



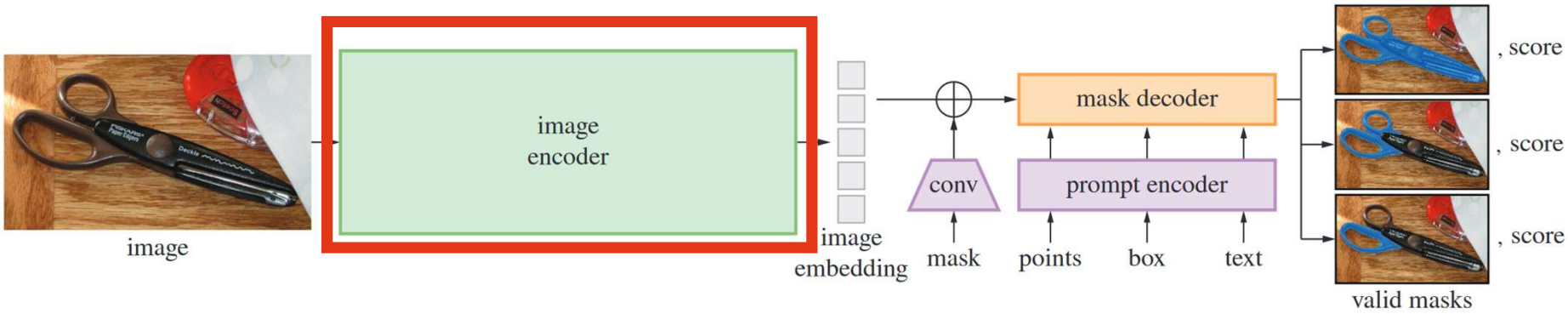
Segment Anything Task

2. Segment Anything Task

We take inspiration from NLP, where the next token prediction task is used for foundation model pre-training *and* to solve diverse downstream tasks via prompt engineering [10]. To build a foundation model for segmentation, we aim to define a task with analogous capabilities.

Task. We start by translating the idea of a prompt from NLP to segmentation, where a prompt can be a set of foreground / background points, a rough box or mask, free-form text, or, in general, any information indicating what to segment in an image. The *promptable segmentation task*, then, is to return a *valid* segmentation mask given any *prompt*. The requirement of a "valid" mask simply means that even when a prompt is *ambiguous* and could refer to multiple objects (*e.g.*, recall the shirt *vs.* person example, and see Fig. 3), the output should be a reasonable mask for at least *one* of those objects. This requirement is similar to expecting a language model to output a coherent response to an ambiguous prompt. We choose this task because it leads to a natural pre-training algorithm *and* a general method for zero-shot transfer to downstream segmentation tasks via prompting.

Segment Anything Model



Segment Anything Model: Image Encoder

Image encoder. Motivated by scalability and powerful pre-training methods, we use an MAE [47] pre-trained Vision Transformer (ViT) [33] minimally adapted to process high resolution inputs [62]. The image encoder runs once per image and can be applied prior to prompting the model.

Vision Transformer

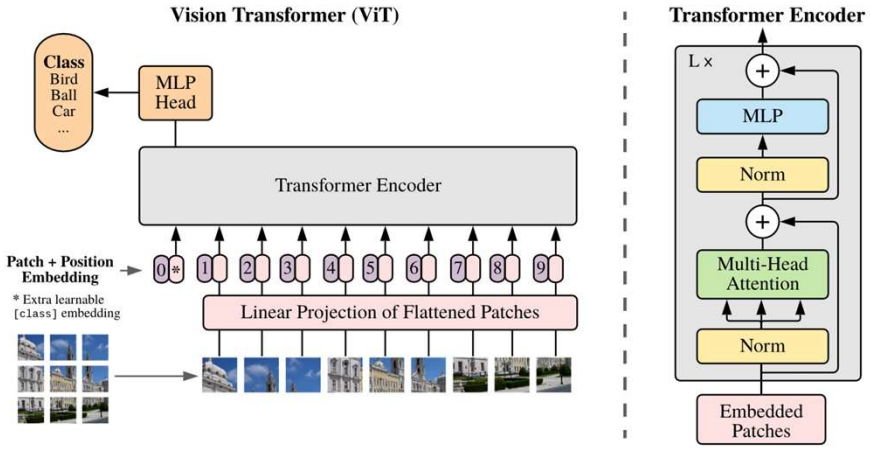


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al (2017).

AN IMAGE IS WORTH 16X16 WORDS:
TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

Alexey Dosovitskiy[*,†], **Lucas Beyer**[*], **Alexander Kolesnikov**[*], **Dirk Weissenborn**[*],
Xiaohua Zhai[*], **Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer,**
Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby[*,†]
[*]equal technical contribution, [†]equal advising
Google Research, Brain Team
{adosovitskiy, neilhoulsby}@google.com

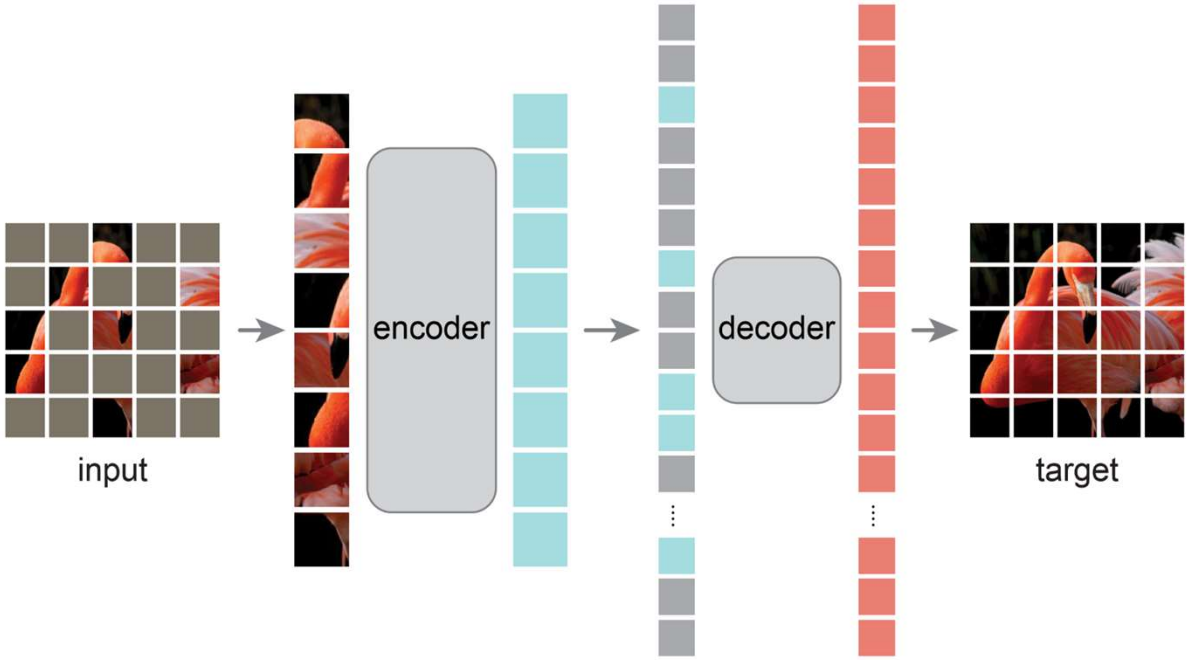
Masked Autoencoder Vision Transformer

Masked Autoencoders Are Scalable Vision Learners

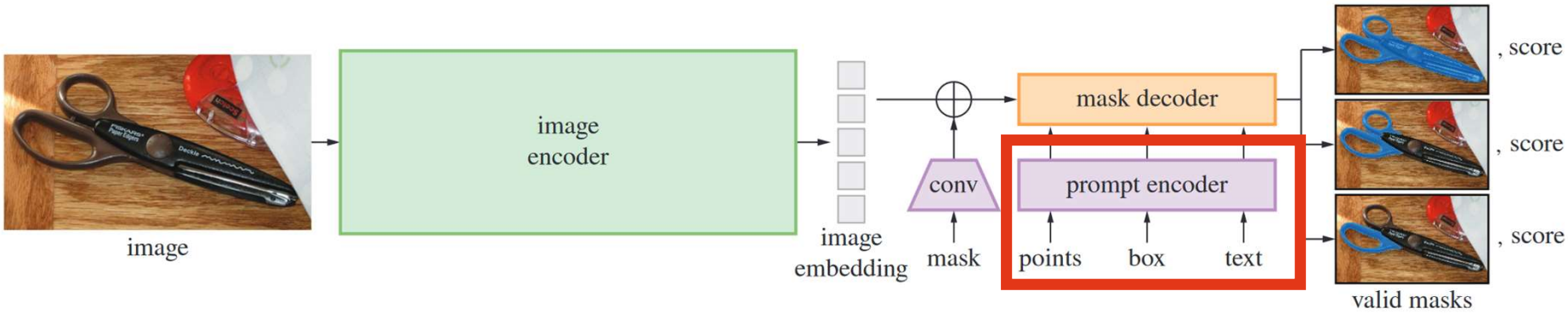
Kaiming He*,† Xinlei Chen* Saining Xie Yanghao Li Piotr Dollár Ross Girshick

*equal technical contribution †project lead

Facebook AI Research (FAIR)



Segment Anything Model



Segment Anything Model: Prompt Encoder

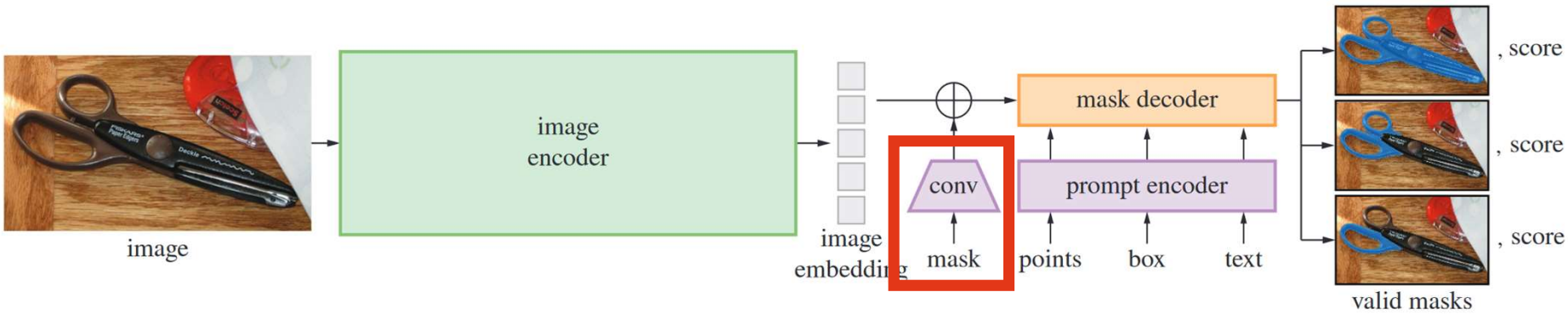
Prompt encoder. We consider two sets of prompts: *sparse* (points, boxes, text) and *dense* (masks). We represent points and boxes by positional encodings [95] summed with learned embeddings for each prompt type and free-form text with an off-the-shelf text encoder from CLIP [82]. Dense prompts (*i.e.*, masks) are embedded using convolutions and summed element-wise with the image embedding.

Segment Anything Model: Prompt Encoder

Prompt encoder. Sparse prompts are mapped to 256-dimensional vectorial embeddings as follows. A point is represented as the sum of a positional encoding [95] of the point's location and one of two learned embeddings that indicate if the point is either in the foreground or background. A box is represented by an embedding pair: (1) the positional encoding of its top-left corner summed with a learned embedding representing "top-left corner" and (2) the same structure but using a learned embedding indicating "bottom-right corner". Finally, to represent free-form text we use the text encoder from CLIP [82] (any text encoder is possible in general). We focus on geometric prompts for the remainder of this section and discuss text prompts in depth in §D.5.

```python
def _embed_points(
    self,
    points: torch.Tensor, # Indicates the coordinates of the points
    labels: torch.Tensor, # Indicate if the point is foreground or background
    pad: bool,
) -> torch.Tensor:
    """Embeds point prompts."""
    points = points + 0.5  # Shift to center of pixel
    if pad: # Add padding if needed (to keep the sequence length constant)
        padding_point = torch.zeros((points.shape[0], 1, 2), device=points.device)
        padding_label = -torch.ones((labels.shape[0], 1), device=labels.device) # The -1 label indicats padding
        points = torch.cat([points, padding_point], dim=1) # Append the padding point
        labels = torch.cat([labels, padding_label], dim=1) # Append the padding label
    point_embedding = self.pe_layer.forward_with_coords(points, self.input_image_size) # Obtain the positional encodings
    point_embedding[labels == -1] = 0.0 # zero out padding
    point_embedding[labels == -1] += self.not_a_point_embed.weight # Add special embedding to indicate padding
    point_embedding[labels == 0] += self.point_embeddings[0].weight # Add embedding for background points
    point_embedding[labels == 1] += self.point_embeddings[1].weight # Add embedding for foreground points
    return point_embedding

def _embed_boxes(self, boxes: torch.Tensor) -> torch.Tensor:
    """Embeds box prompts."""
    boxes = boxes + 0.5  # Shift to center of pixel
    coords = boxes.reshape(-1, 2, 2)
    corner_embedding = self.pe_layer.forward_with_coords(coords, self.input_image_size) # Obtain the positional encodings
    corner_embedding[:, 0, :] += self.point_embeddings[2].weight # Special embedding for top-left corner
    corner_embedding[:, 1, :] += self.point_embeddings[3].weight # Special embedding for bottom-right corner
    return corner_embedding
```

Segment Anything Model



Segment Anything Model: Mask Convolution

Dense prompts (*i.e.*, masks) have a spatial correspondence with the image. We input masks at a 4× lower resolution than the input image, then downscale an additional 4× using two 2×2, stride-2 convolutions with output channels 4 and 16, respectively. A final 1×1 convolution maps the channel dimension to 256. Each layer is separated by GELU activations [50] and layer normalization. The mask and image embedding are then added element-wise. If there is no mask prompt, a learned embedding representing "no mask" is added to each image embedding location.

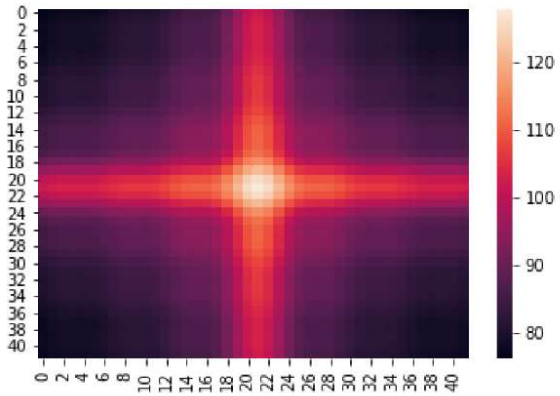
```python
if masks is not None:
    dense_embeddings = self._embed_masks(masks)
else: # If no mask is specified, use a special "no mask" embedding
    dense_embeddings = self.no_mask_embed.weight.reshape(1, -1, 1, 1).expand(
        bs, -1, self.image_embedding_size[0], self.image_embedding_size[1]
    )
```

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

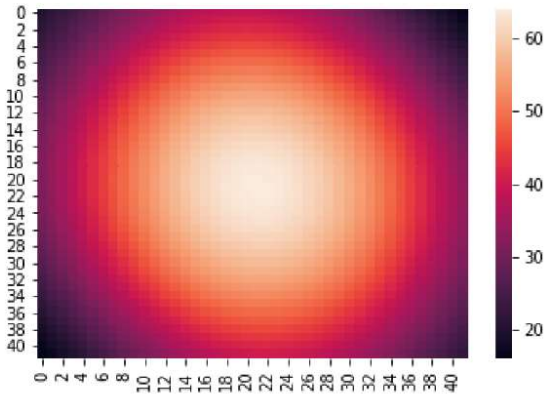
```python
self.mask_downscaling = nn.Sequential(
    nn.Conv2d(1, mask_in_chans // 4, kernel_size=2, stride=2),
    LayerNorm2d(mask_in_chans // 4),
    activation(),
    nn.Conv2d(mask_in_chans // 4, mask_in_chans, kernel_size=2, stride=2),
    LayerNorm2d(mask_in_chans),
    activation(),
    nn.Conv2d(mask_in_chans,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

```python
# 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 # Add mask embeddings to the image
pos_src = torch.repeat_interleave(image_pe, tokens.shape[0], dim=0)
b, c, h, w = src.shape
```

Segment Anything Model: Positional Encodings



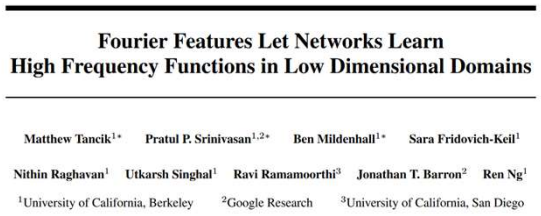
(a) Sinusoidal-concatenation encoding.



(b) 2D Fourier feature PE.

Figure 2: The similarities of the center position to the rest positions on the 2D space, based on the dot product between their positional encoding of each approach.

Used by the Segment Anything Model

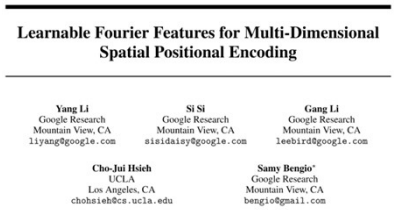


Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains

Matthew Tancik[1*] Pratul P. Srinivasan[1,2*] Ben Mildenhall[1*] Sara Fridovich-Keil[1]

Nithin Raghavan[1] Utkarsh Singhal[1] Ravi Ramamoorthi[3] Jonathan T. Barron[2] Ren Ng[1]

[1]University of California, Berkeley [2]Google Research [3]University of California, San Diego



Learnable Fourier Features for Multi-Dimensional Spatial Positional Encoding

Yang Li
Google Research
Mountain View, CA
liyang@google.com

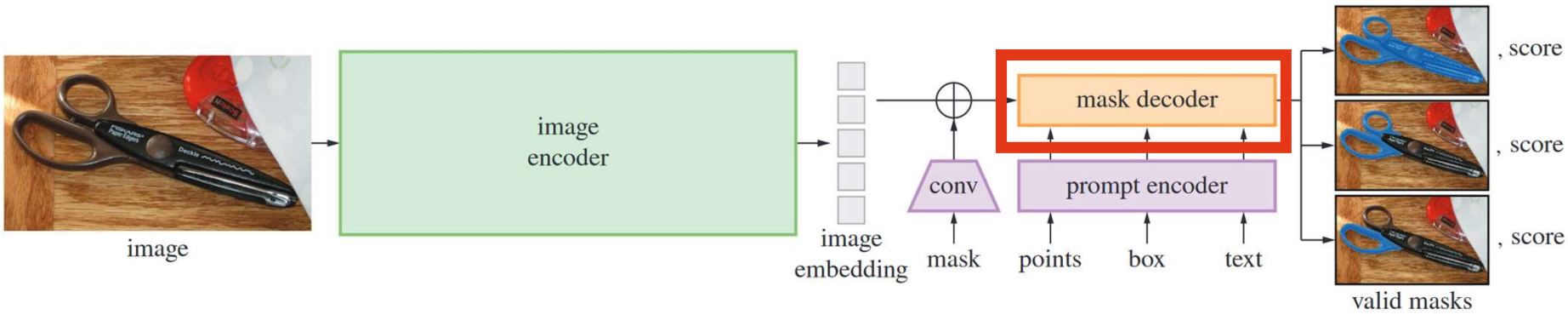
Si Si
Google Research
Mountain View, CA
sisidaisy@google.com

Gang Li
Google Research
Mountain View, CA
leebird@google.com

Cho-Jui Hsieh
UCLA
Los Angeles, CA
chohsieh@cs.ucla.edu

Samy Bengio*
Google Research
Mountain View, CA
bengio@gmail.com

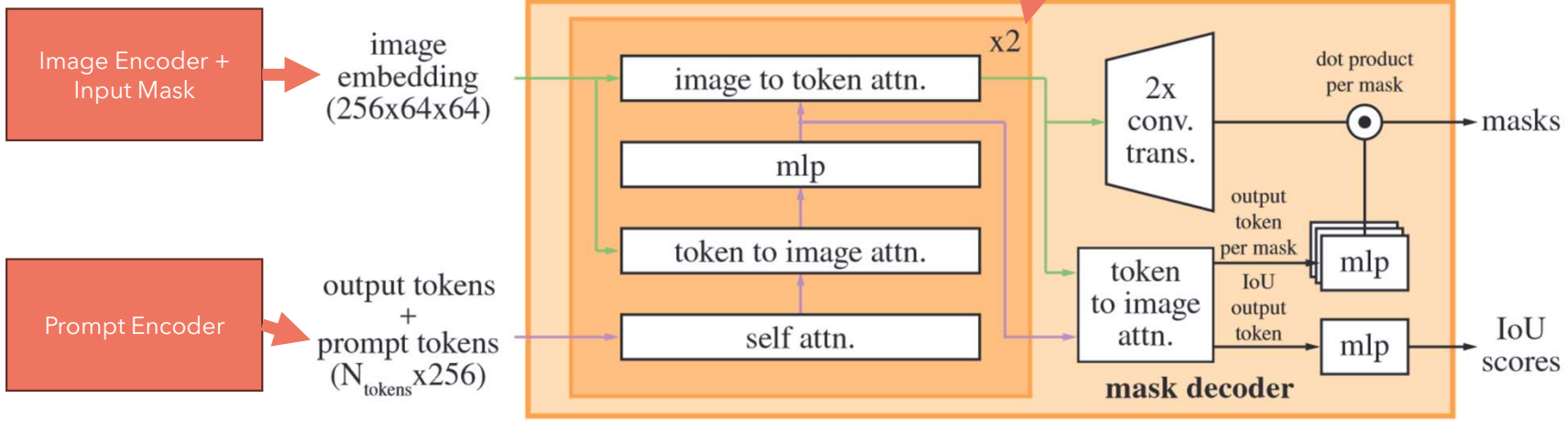
Segment Anything Model



Segment Anything Model: Mask Decoder

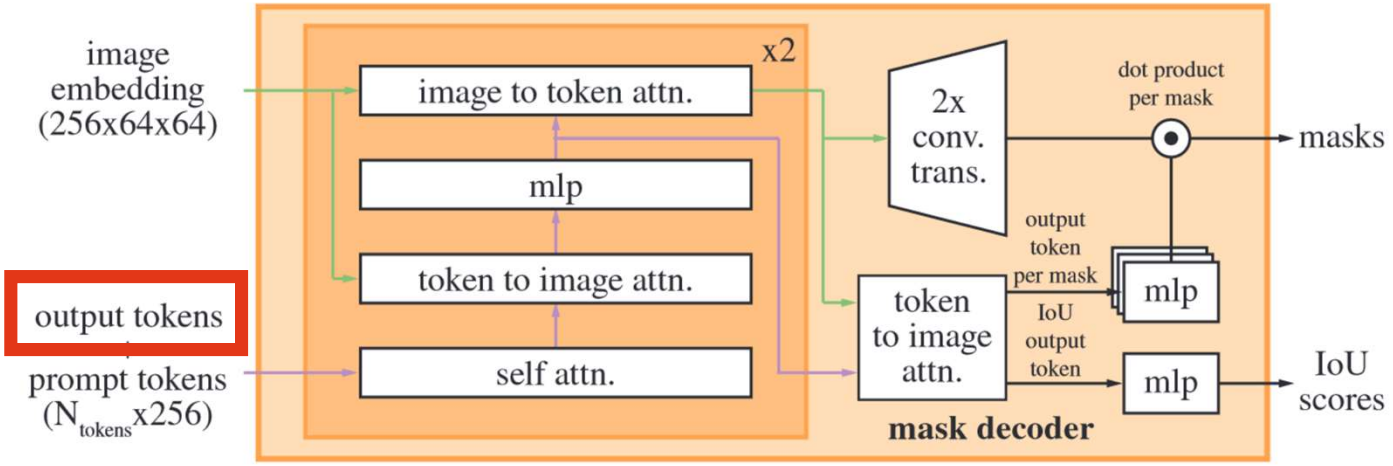
We have two layers, one on top of the other (output of the first is the input of the second).

```python
# Prepare queries
queries = point_embedding
keys = image_embedding

# Apply transformer blocks and final layernorm
for layer in self.layers:
    queries, keys = layer(
        queries=queries,
        keys=keys,
        query_pe=point_embedding,
        key_pe=image_pe,
    )
```



Segment Anything Model: Output Tokens

Lightweight mask decoder. This module efficiently maps the image embedding and a set of prompt embeddings to an output mask. To combine these inputs, we take inspiration from Transformer segmentation models [14, 20] and modify a standard Transformer decoder [103]. Before applying our decoder, we first insert into the set of prompt embeddings a learned output token embedding that will be used at the decoder's output, analogous to the `[class]` token in [33]. For simplicity, we refer to these embeddings (*not* including the image embedding) collectively as "tokens".



One token representing the IoU (Intersection over Union) and three tokens representing the output masks (part, subpart, whole) are added before the sparse token embeddings (points, boxes and text).

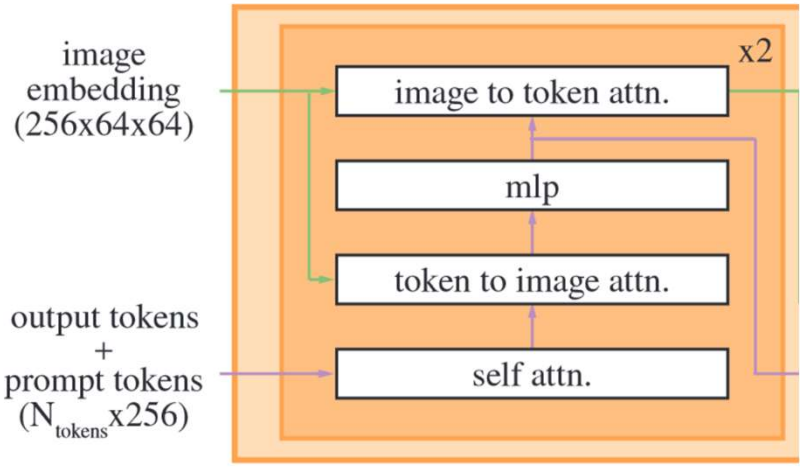
```
# Concatenate output tokens
output_tokens = torch.cat([self.iou_token.weight, self.mask_tokens.weight], dim=0)
output_tokens = output_tokens.unsqueeze(0).expand(sparse_prompt_embeddings.size(0), -1, -1)
tokens = torch.cat((output_tokens, sparse_prompt_embeddings), dim=1)
```

We use WordPiece embeddings (Wu et al., 2016) with a 30,000 token vocabulary. The first token of every sequence is always a special classification token (`[CLS]`). The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks. Sentence pairs are packed together into a

Similar to BERT's [CLS] token and Vision Transformer's [class] token!

Segment Anything Model: Transformer

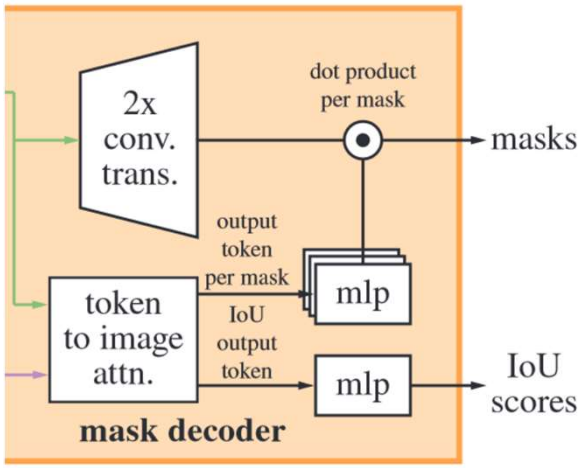
```python
def forward(
    self, queries: Tensor, keys: Tensor, query_pe: Tensor, key_pe: Tensor
) -> Tuple[Tensor, Tensor]:
    # Self attention block
    if self.skip_first_layer_pe:
        queries = self.self_attn(q=queries, k=queries, v=queries)
    else:
        q = queries + query_pe # Add positional encodings to the prompt tokens
        attn_out = self.self_attn(q=q, k=q, v=queries) # Run self-attention on the prompt
        queries = queries + attn_out # Add back the original prompts as indicated in the paper
    queries = self.norm1(queries)

    # Cross attention block, tokens attending to image embedding
    q = queries + query_pe # The queries are the prompt tokens
    k = keys + key_pe # The keys are the image embedding + positional encodings
    # The values are the image embedding (without positional encodings)
    attn_out = self.cross_attn_token_to_image(q=q, k=k, v=keys)
    queries = queries + attn_out
    queries = self.norm2(queries)

    # MLP block
    mlp_out = self.mlp(queries)
    queries = queries + mlp_out
    queries = self.norm3(queries)

    # Cross attention block, image embedding attending to tokens
    q = queries + query_pe # q = prompt tokens + positional encodingss
    k = keys + key_pe # k = image embedding + positional encodings
    # Bad variable naming practices from META!
    attn_out = self.cross_attn_image_to_token(q=k, k=q, v=queries)
    keys = keys + attn_out
    keys = self.norm4(keys)

    return queries, keys
```



To ensure the decoder has access to critical geometric information the positional encodings are added to the image embedding whenever they participate in an attention layer. Additionally, the *entire* original prompt tokens (including their positional encodings) are re-added to the updated tokens whenever they participate in an attention layer. This allows for a strong dependence on both the prompt token's geometric location and type.

Segment Anything Model: Output

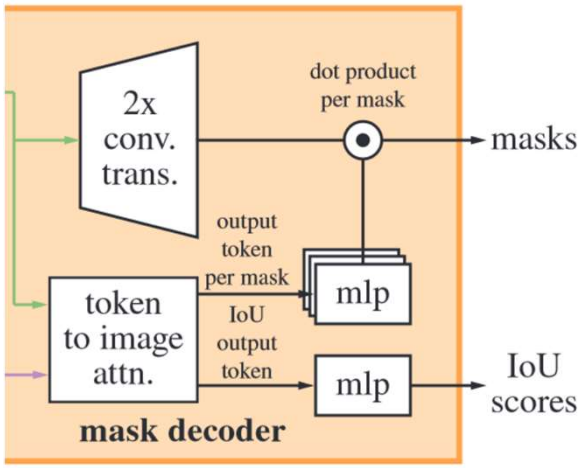
```python
# Run the transformer
# hs is the transformer output for the prompt
# src is the transformer output for the image
hs, src = self.transformer(src, pos_src, tokens)
iou_token_out = hs[:, 0, :] # Output token for the IoU prediction
mask_tokens_out = hs[:, 1 : (1 + self.num_mask_tokens), :] # Output tokens for masks

# 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) # Upscale image
hyper_in_list: List[torch.Tensor] = []
for i in range(self.num_mask_tokens): # Run each token through its MLP
    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
# Dot product of the MLP output for each "output token" and the upscaled image
# (each output token represents a mask)
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)

return masks, iou_pred
```



Segment Anything Model: Output

Making the model ambiguity-aware. As described, a single input prompt may be ambiguous in the sense that it corresponds to multiple valid masks, and the model will learn to average over these masks. We eliminate this problem with a simple modification: instead of predicting a single mask, we use a small number of output tokens and predict multiple masks simultaneously. By default we predict three masks, since we observe that three layers (whole, part, and subpart) are often enough to describe nested masks. During training, we compute the loss (described shortly) between the ground truth and each of the predicted masks, but only backpropagate from the lowest loss. This is a common technique used for models with multiple outputs [15, 45, 64]. For use in applications, we'd like to rank predicted masks, so we add a small head (operating on an additional output token) that estimates the IoU between each predicted mask and the object it covers.

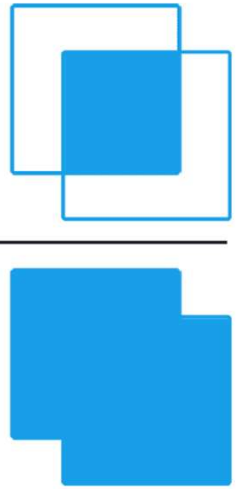
Ambiguity is much rarer with multiple prompts and the three output masks will usually become similar. To minimize computation of degenerate losses at training and ensure the single unambiguous mask receives a regular gradient signal, we only predict a single mask when more than one prompt is given. This is accomplished by adding a fourth output token for an additional mask prediction. This fourth mask is never returned for a single prompt and is the only mask returned for multiple prompts.



IoU (Intersection over Union)

Image from Wikipedia

$$IoU = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$



Segment Anything Model: Loss

Losses. We supervise mask prediction with a linear combination of focal loss [65] and dice loss [73] in a 20:1 ratio of focal loss to dice loss, following [20, 14]. Unlike [20, 14], we observe that auxiliary deep supervision after each decoder layer is unhelpful. The IoU prediction head is trained with mean-square-error loss between the IoU prediction and the predicted mask's IoU with the ground truth mask. It is added to the mask loss with a constant scaling factor of 1.0.

Segment Anything Model: Focal loss

Focal Loss: very used in object detection and it is like Cross Entropy but adjusted for class imbalance.

That is, it allows the model to put more focus on hard examples and less on easy ones.

Why is there class imbalance? Because most of our pixels will be non-mask and only a small percentage will be part of the mask.

3.2. Focal Loss Definition

As our experiments will show, the large class imbalance encountered during training of dense detectors overwhelms the cross entropy loss. Easily classified negatives comprise the majority of the loss and dominate the gradient. While α balances the importance of positive/negative examples, it does not differentiate between easy/hard examples. Instead, we propose to reshape the loss function to down-weight easy examples and thus focus training on hard negatives.

More formally, we propose to add a modulating factor $(1 - p_t)^\gamma$ to the cross entropy loss, with tunable *focusing* parameter $\gamma \geq 0$. We define the focal loss as:

$$\mathrm{FL}(p_t) = -(1 - p_t)^\gamma \log(p_t). \tag{4}$$

Focal Loss for Dense Object Detection

Tsung-Yi Lin Priya Goyal Ross Girshick Kaiming He Piotr Dollár
Facebook AI Research (FAIR)

Segment Anything Model: Dice loss

The Dice score, also known as the Sørensen–Dice coefficient, is a measure of the similarity between two sets of data. In the context of image segmentation, the Dice score can be used to evaluate the similarity between a predicted segmentation mask and the ground truth segmentation mask. The Dice score ranges from 0, indicating no overlap, to 1, indicating perfect overlap. It is equal to the F1 score

$$\text{Dice} = \frac{2\ X\ Area\ of\ overlap}{Total\ area} = \frac{2 \text{ x } \text{Prediction} \cap \text{Ground truth}}{\text{Prediction} + \text{Ground truth}}$$

https://medium.com/mlearning-ai/understanding-evaluation-metrics-in-medical-image-segmentation-d289a373a3f

$$Dice\ \boldsymbol{Loss} = 1\ - Dice\ score$$

3 Dice loss layer

The network predictions, which consist of two volumes having the same resolution as the original input data, are processed through a soft-max layer which outputs the probability of each voxel to belong to foreground and to background. In medical volumes such as the ones we are processing in this work, it is not uncommon that the anatomy of interest occupies only a very small region of the scan. This often causes the learning process to get trapped in local minima of the loss function yielding a network whose predictions are strongly biased towards background. As a result the foreground region is often missing or only partially detected. Several previous approaches resorted to loss functions based on sample re-weighting where foreground regions are given more importance than background ones during learning. In this work we propose a novel objective function based on dice coefficient, which is a quantity ranging between 0 and 1 which we aim to maximise. The dice coefficient D between two binary volumes can be written as

$$D = \frac{2\sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$

V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation

Fausto Milletari[1], Nassir Navab[1,2], Seyed-Ahmad Ahmadi[3]

Segment Anything Model: Interactive training

Training algorithm. Following recent approaches [92, 37], we simulate an interactive segmentation setup during training. First, with equal probability either a foreground point or bounding box is selected randomly for the target mask. Points are sampled uniformly from the ground truth mask. Boxes are taken as the ground truth mask's bounding box, with random noise added in each coordinate with standard deviation equal to 10% of the box sidelength, to a maximum of 20 pixels. This noise profile is a reasonable compromise between applications like instance segmentation, which produce a tight box around the target object, and interactive segmentation, where a user may draw a loose box.

After making a prediction from this first prompt, subsequent points are selected uniformly from the error region between the previous mask prediction and the ground truth mask. Each new point is foreground or background if the error region is a false negative or false positive, respectively. We also supply the mask prediction from the previous iteration as an additional prompt to our model. To provide the next iteration with maximal information, we supply the unthresholded mask logits instead of the binarized mask. When multiple masks are returned, the mask passed to the next iteration and used to sample the next point is the one with the highest predicted IoU.

We find diminishing returns after 8 iteratively sampled points (we have tested up to 16). Additionally, to encourage the model to benefit from the supplied mask, we also use two more iterations where no additional points are sampled. One of these iterations is randomly inserted among the 8 iteratively sampled points, and the other is always at the end. This gives 11 total iterations: one sampled initial input prompt, 8 iteratively sampled points, and two iterations where no new external information is supplied to the model so it can learn to refine its own mask predictions. We note that using a relatively large number of iterations is possible because our lightweight mask decoder requires less than 1% of the image encoder's compute and, therefore, each iteration adds only a small overhead. This is unlike previous interactive methods that perform only one or a few interactive steps per optimizer update [70, 9, 37, 92].

Segment Anything Data Engine

As segmentation masks are not abundant on the internet, we built a data engine to enable the collection of our 1.1B mask dataset, SA-1B. The data engine has three stages: (1) a model-assisted manual annotation stage, (2) a semi-automatic stage with a mix of automatically predicted masks and model-assisted annotation, and (3) a fully automatic stage in which our model generates masks without annotator input. We go into details of each next.

Segment Anything Data Engine: Assisted-Manual

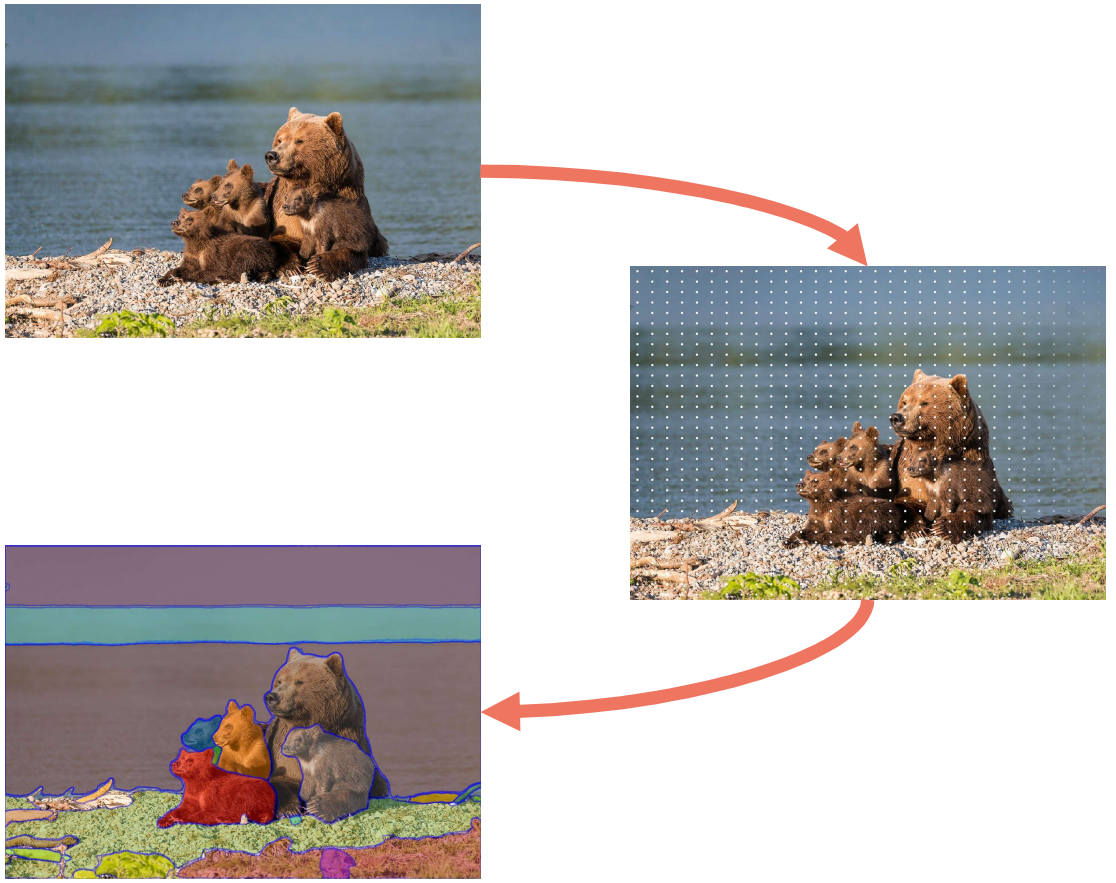
Assisted-manual stage. In the first stage, resembling classic interactive segmentation, a team of professional annotators labeled masks by clicking foreground / background object points using a browser-based interactive segmentation tool powered by SAM. Masks could be refined using pixel-precise "brush" and "eraser" tools. Our model-assisted annotation runs in real-time directly inside a browser (using precomputed image embeddings) enabling a truly interactive experience. We did not impose semantic constraints for labeling objects, and annotators freely labeled both "stuff" and "things" [1]. We suggested annotators label objects they could name or describe, but did not collect these names or descriptions. Annotators were asked to label objects in order of prominence and were encouraged to proceed to the next image once a mask took over 30 seconds to annotate.

Segment Anything Data Engine: Semi-Automatic Stage

Semi-automatic stage. In this stage, we aimed to increase the *diversity* of masks in order to improve our model's ability to segment anything. To focus annotators on less prominent objects, we first automatically detected confident masks. Then we presented annotators with images prefilled with these masks and asked them to annotate any additional unannotated objects. To detect confident masks, we trained a bounding box detector [84] on all first stage masks using a generic "object" category. During this stage we collected an additional 5.9M masks in 180k images (for a total of 10.2M masks). As in the first stage, we periodically retrained our model on newly collected data (5 times). Average annotation time per mask went back up to 34 seconds (excluding the automatic masks) as these objects were more challenging to label. The average number of masks per image went from 44 to 72 masks (including the automatic masks).

Segment Anything Data Engine: Fully-Automatic Stage

Fully automatic stage. In the final stage, annotation was *fully automatic*. This was feasible due to two major enhancements to our model. First, at the start of this stage, we had collected enough masks to greatly improve the model, including the diverse masks from the previous stage. Second, by this stage we had developed the ambiguity-aware model, which allowed us to predict valid masks even in ambiguous cases. Specifically, we prompted the model with a 32×32 regular grid of points and for each point predicted a set of masks that may correspond to valid objects. With the ambiguity-aware model, if a point lies on a part or subpart, our model will return the subpart, part, and whole object. The IoU prediction module of our model is used to select *confident* masks; moreover, we identified and selected only *stable* masks (we consider a mask stable if thresholding the probability map at $0.5 - \delta$ and $0.5 + \delta$ results in similar masks). Finally, after selecting the confident and stable masks, we applied non-maximal suppression (NMS) to filter duplicates. To further improve the quality of smaller masks, we also processed multiple overlapping zoomed-in image crops. For further details of this stage, see §B. We applied fully automatic mask generation to all 11M images in our dataset, producing a total of 1.1B high-quality masks.



NMS (Non Maximal Suppression)

An algorithm to select the best bounding boxes (or masks) among many alternatives.
Input: a list of bounding boxes along with their confidence scores
Parameters: an IoU threshold

Algorithm:

1. Select the bounding box *b1* with the max confidence score and add it to the result R
2. Remove all the other boxes that have an IoU threshold with *b1* higher than the threshold
3. Go back to 1

Output: A list of bounding boxes that survived the process.

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!