Knowledge Distillation of Large Language Models

A Simple and Effective Pruning Approach for Large Language Models

SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot

OWQ: Lessons learned from activation outliers for weight quantization in large language models

SqueezeLLM: Dense-and-Sparse Quantization

第二日

OBQ的原理

SAM（调研）

数据驱动的machine learning

跑wanda

Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning

$$\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2 \quad \text{s.t.} \quad \mathcal{C}(\widehat{\mathbf{W}}_\ell) > C.$$

$$w_p = \text{argmin}_{w_p} \frac{w_p^2}{[\mathbf{H}^{-1}]_{pp}}, \quad \boldsymbol{\delta_p} = -\frac{w_p}{[\mathbf{H}^{-1}]_{pp}} \cdot \mathbf{H}^{-1}_{:,p},$$

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2}\delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O\left(\|\delta \mathbf{w}\|^3\right)$$

$$\varepsilon(\delta \mathbf{w}, \lambda) = \frac{1}{2}\delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + \lambda\left(\mathbf{e}_m^T \cdot \delta \mathbf{w} + w_m\right)$$

$$\delta \mathbf{w} = -\frac{w_m}{\left[\mathbf{H}^{-1}\right]_{mm}} \cdot \mathbf{H}^{-1}_{:,m} \text{ incurring error } \varepsilon_m = \frac{w_m^2}{2\left[\mathbf{H}^{-1}\right]_{mm}}.$$

$$\sum_{i=1}^{d_{\text{row}}} ||\mathbf{W}_{\mathbf{i},:}\mathbf{X} - \widehat{\mathbf{W}}_{\mathbf{i},:}\mathbf{X}||_2^2.$$

更新海森矩阵

更新海森逆矩阵

$$\mathbf{H}_{-p}^{-1} = \left(\mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{pp}}\mathbf{H}_{:,p}^{-1}\mathbf{H}_{p,:}^{-1}\right)_{-p},$$

$$w_p = \text{argmin}_{w_p} \frac{w_p^2}{[\mathbf{H}^{-1}]_{pp}}, \quad \boldsymbol{\delta_p} = -\frac{w_p}{[\mathbf{H}^{-1}]_{pp}} \cdot \mathbf{H}_{:,p}^{-1},$$

$$w_p = \text{argmin}_{w_p} \frac{(\text{quant}(w_p) - w_p)^2}{[\mathbf{H}^{-1}]_{pp}}, \quad \boldsymbol{\delta_p} = -\frac{w_p - \text{quant}(w_p)}{[\mathbf{H}^{-1}]_{pp}} \cdot \mathbf{H}_{:,p}^{-1}.$$

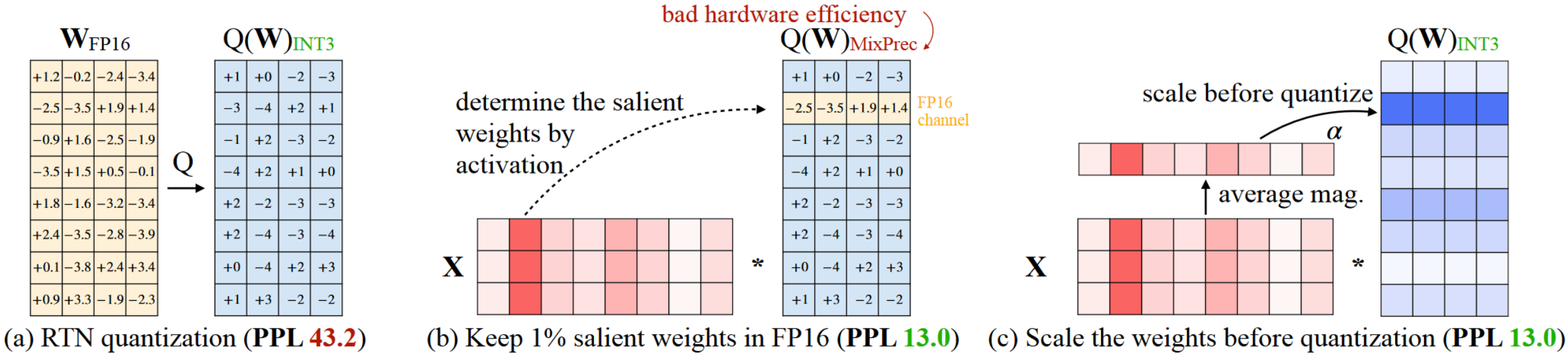
GPTQ

1. Arbitrary Order Insight
2. Lazy Batch-Updates
3. Cholesky Reformulation

Algorithm 1 Quantize $\mathbf{W}$ given inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize B.

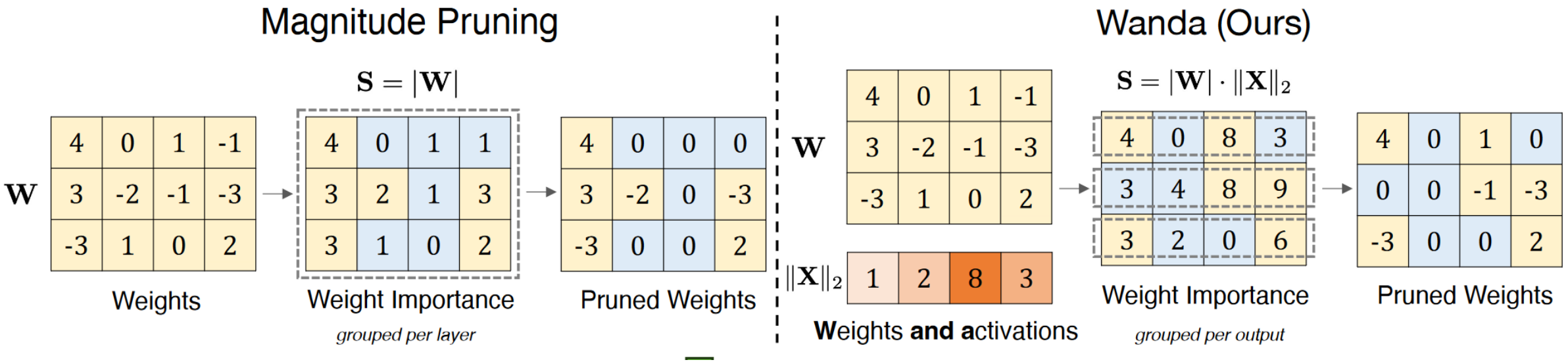
$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ *// quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ *// Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
  for $j = i, \ldots, i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ *// quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) \,/\, [\mathbf{H}^{-1}]_{jj}$ *// quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ *// update weights in block*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ *// update all remaining weights*
end for

AWQ:Activation-aware Weight Quantization for LLM Compression and Acceleration



(a) RTN quantization (**PPL 43.2**) (b) Keep 1% salient weights in FP16 (**PPL 13.0**) (c) Scale the weights before quantization (**PPL 13.0**)

A Simple and Effective Pruning Approach for Large Language Models



Algorithm 1 PyTorch code for Wanda

```
# W: weight matrix (C_out, C_in);
# X: input matrix (N * L, C_in);
# s: desired sparsity level, between 0 and 1;

def prune(W, X, s):
  metric = W.abs() * X.norm(p=2, dim=0)

  _, sorted_idx = torch.sort(metric, dim=1)
  pruned_idx = sorted_idx[:,:int(C_in * s)]
  W.scatter_(dim=1, index=pruned_idx, src=0)
  return W
```

Method	Weight Update	Sparsity	LLaMA 7B	LLaMA 13B	LLaMA 30B	LLaMA 65B
Dense	-	0%	5.68	5.09	4.77	3.56
Magnitude	✗	50%	17.29	20.21	7.54	5.90
SparseGPT	✓	50%	**7.22**	6.21	5.31	**4.57**
Wanda	✗	50%	7.26	**6.15**	**5.24**	**4.57**
Magnitude	✗	4:8	16.84	13.84	7.62	6.36
SparseGPT	✓	4:8	8.61	**7.40**	6.17	5.38
Wanda	✗	4:8	**8.57**	**7.40**	**5.97**	**5.30**
Magnitude	✗	2:4	42.13	18.37	9.10	7.11
SparseGPT	✓	2:4	**11.00**	**9.11**	7.16	6.28
Wanda	✗	2:4	11.53	9.58	**6.90**	**6.25**

Solver Reconstruction

LoRC is inspired by the employment of low-rank matrix factorization on the quantization error matrix $E := W - \hat{W}$, where W represents the original weight and $\hat{W}$ is the quantized weight. LoRC approximates the error E with $\hat{E} = \hat{U}\hat{V}$ by using two low-rank matrices $\hat{U}$ and $\hat{V}$. This results in a more accurate approximation of the original weight matrix W by $\hat{W}_{\text{lorc}} = \hat{W} + \hat{E}$, thereby reducing quantization errors: $\|W - \hat{W}\| \geq \|W - \hat{W}_{\text{lorc}}\|$. LoRC consists of two steps:

Step I: Implement Singular Value Decomposition (SVD) on the error matrix $E = U\Sigma V$, where $U \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$ and $V \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$ are unitary matrices, and $\Sigma \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ is a diagonal matrix with its diagonal elements ordered in a descending manner.

Step II: We formulate the matrix $\hat{E} = \hat{U}\hat{V}$ where $\hat{U} = U_m(\Sigma_m)^{\frac{1}{2}}$ and $\hat{V} = (\Sigma_m)^{\frac{1}{2}}V_m$. Here, $U_m = U_{:,1:m} \in \mathbb{R}^{d_{\text{in}} \times m}$, $V_m = V_{1:m,:} \in \mathbb{R}^{m \times d_{\text{out}}}$, and $\Sigma_m = \Sigma_{1:m,1:m} \in \mathbb{R}^{m \times m}$.

Bits	**LoRC**	Coarse-grained weight quantization (per-row block-size)					Fine-grained quantization on weight (256 block-size)				
		OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-176b	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-176b
W8A16		11.90	11.22	10.70	10.33	10.90	11.90	11.22	10.70	10.33	10.90
W4A16	✗	12.28	11.42	10.78	10.78	11.02	12.05	11.28	10.74	10.50	10.95
	✓	12.10	11.36	10.76	10.34	10.98	11.99	11.29	10.70	10.29	10.93
W3A16	✗	14.18	12.43	11.28	17.77	49.46	12.79	11.63	10.9	11.34	11.13
	✓	13.00	11.90	11.14	10.63	11.30	12.40	11.57	10.83	10.42	11.08
W2A16	✗	120.56	40.17	25.74	225.45	Explode	23.13	15.55	12.68	308.49	12.64
	✓	24.17	18.53	14.39	13.01	14.15	16.27	14.30	12.37	11.54	12.21

SAM

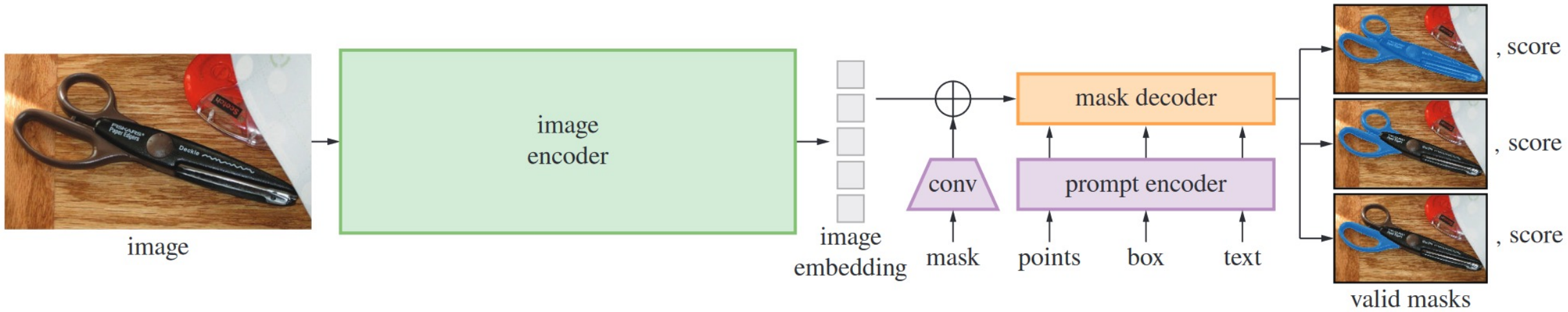


image
image encoder
image embedding
conv
mask
mask decoder
prompt encoder
points
box
text
, score
, score
, score
valid masks

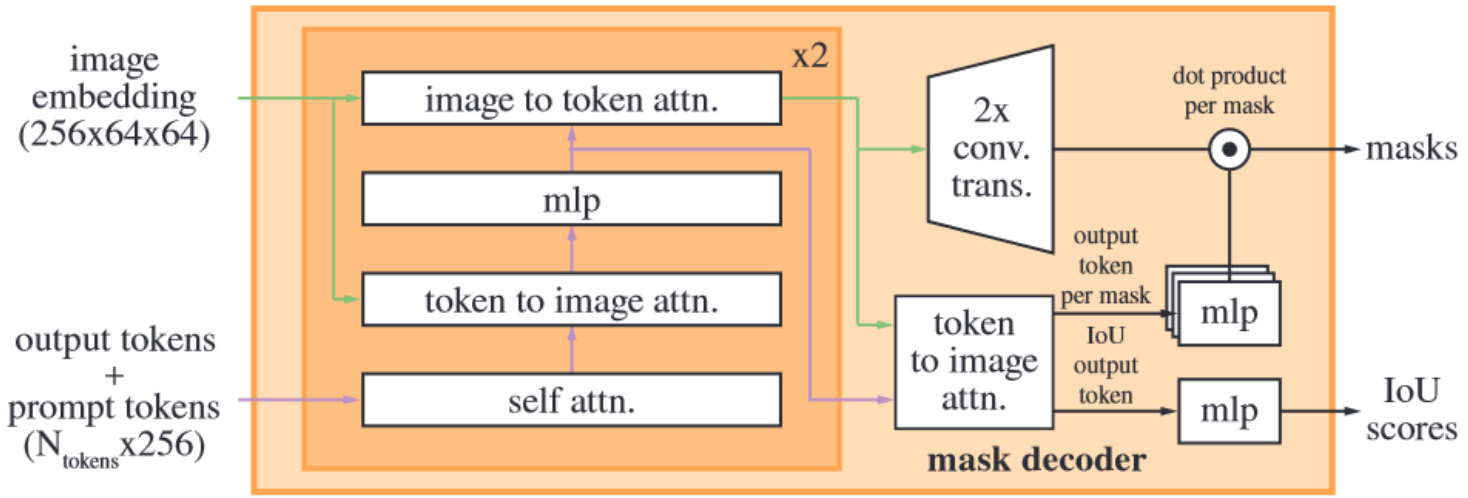
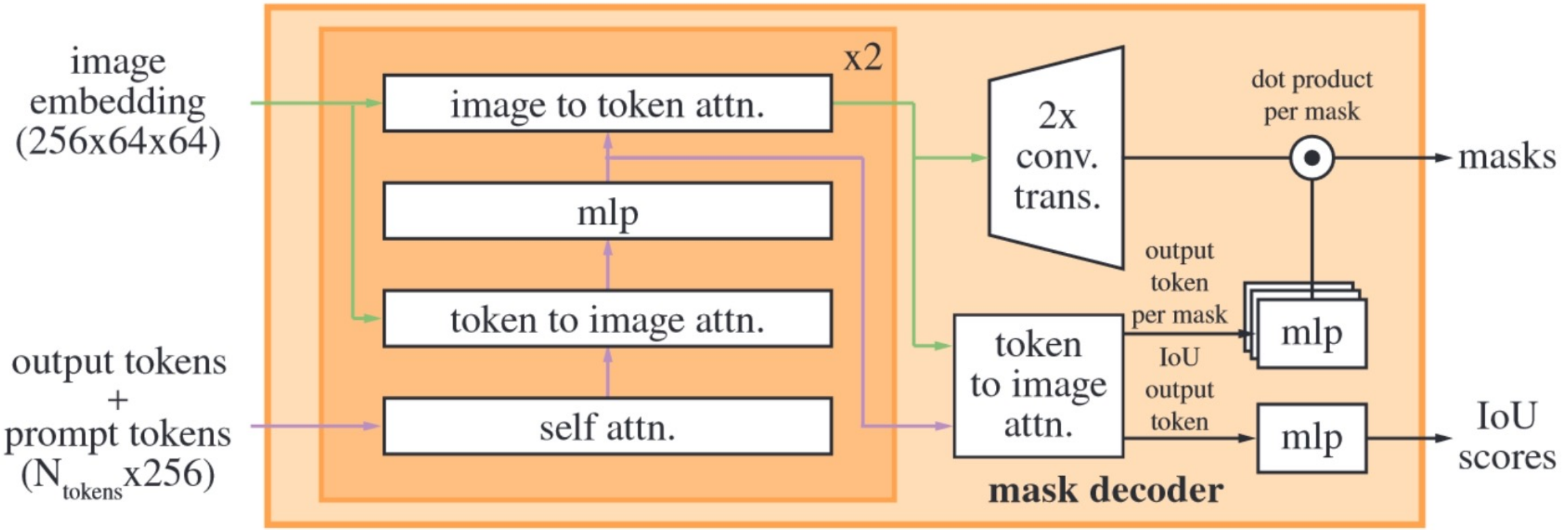


image embedding (256x64x64)
output tokens + prompt tokens (N_{tokens}x256)
x2
image to token attn.
mlp
token to image attn.
self attn.
2x conv. trans.
dot product per mask
masks
token to image attn.
output token per mask
IoU output token
mlp
mlp
IoU scores
mask decoder

```python
def _pe_encoding(self, coords: torch.Tensor) -> torch.Tensor:
    """Positionally encode points that are normalized to [0,1]."""
    # assuming coords are in [0, 1]^2 square and have d_1 x ... x d_n x 2 shape
    coords = 2 * coords - 1
    coords = coords @ self.positional_encoding_gaussian_matrix
    coords = 2 * np.pi * coords
    # outputs d_1 x ... x d_n x C shape
    return torch.cat([torch.sin(coords), torch.cos(coords)], dim=-1)
```

```python
point_embedding[labels == -1] += self.not_a_point_embed.weight
# 对于标签显示为0，也就是背景上的点，将之前的位置编码加上第一个点的嵌入
point_embedding[labels == 0] += self.point_embeddings[0].weight
# 对于是前景上的点，加上第二个点的嵌入
point_embedding[labels == 1] += self.point_embeddings[1].weight
return point_embedding # 返回点提示的嵌入
```

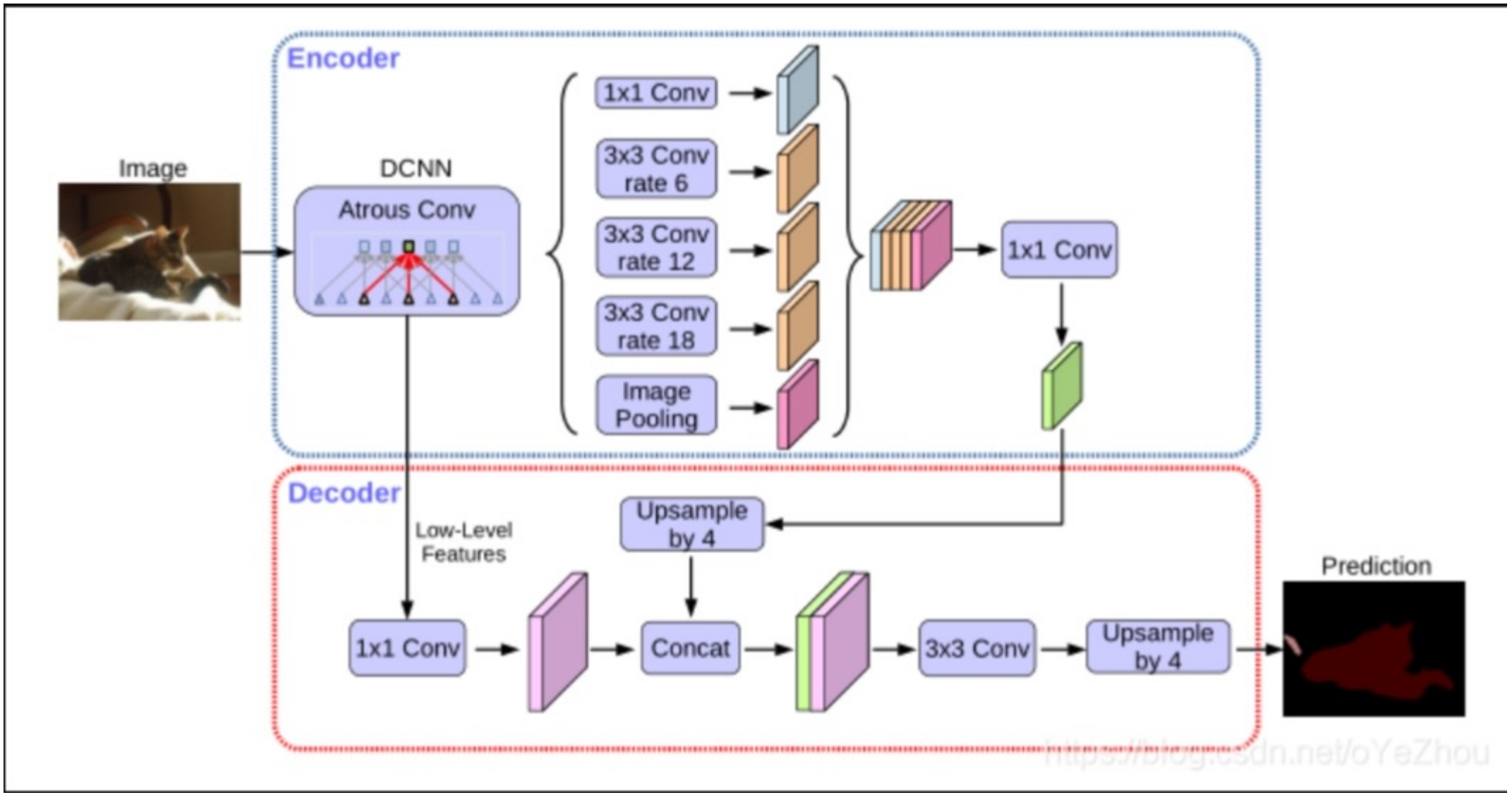
```python
boxes = boxes + 0.5  # Shift to center of pixel
coords = boxes.reshape(-1, 2, 2)
corner_embedding = self.pe_layer.forward_with_coords(coords, self.input_image_size)
corner_embedding[:, 0, :] += self.point_embeddings[2].weight
corner_embedding[:, 1, :] += self.point_embeddings[3].weight
return corner_embedding
```




```python
if points_per_side is not None:
    self.point_grids = build_all_layer_point_grids(
        points_per_side,
        crop_n_layers,
        crop_n_points_downscale_factor,
    )
elif point_grids is not None:
    self.point_grids = point_grids
else:
    raise ValueError("Can't have both points_per_side and point_grid be None.")
```

automatic_mask_generator.py

Class SamAutomaticMaskGenerator

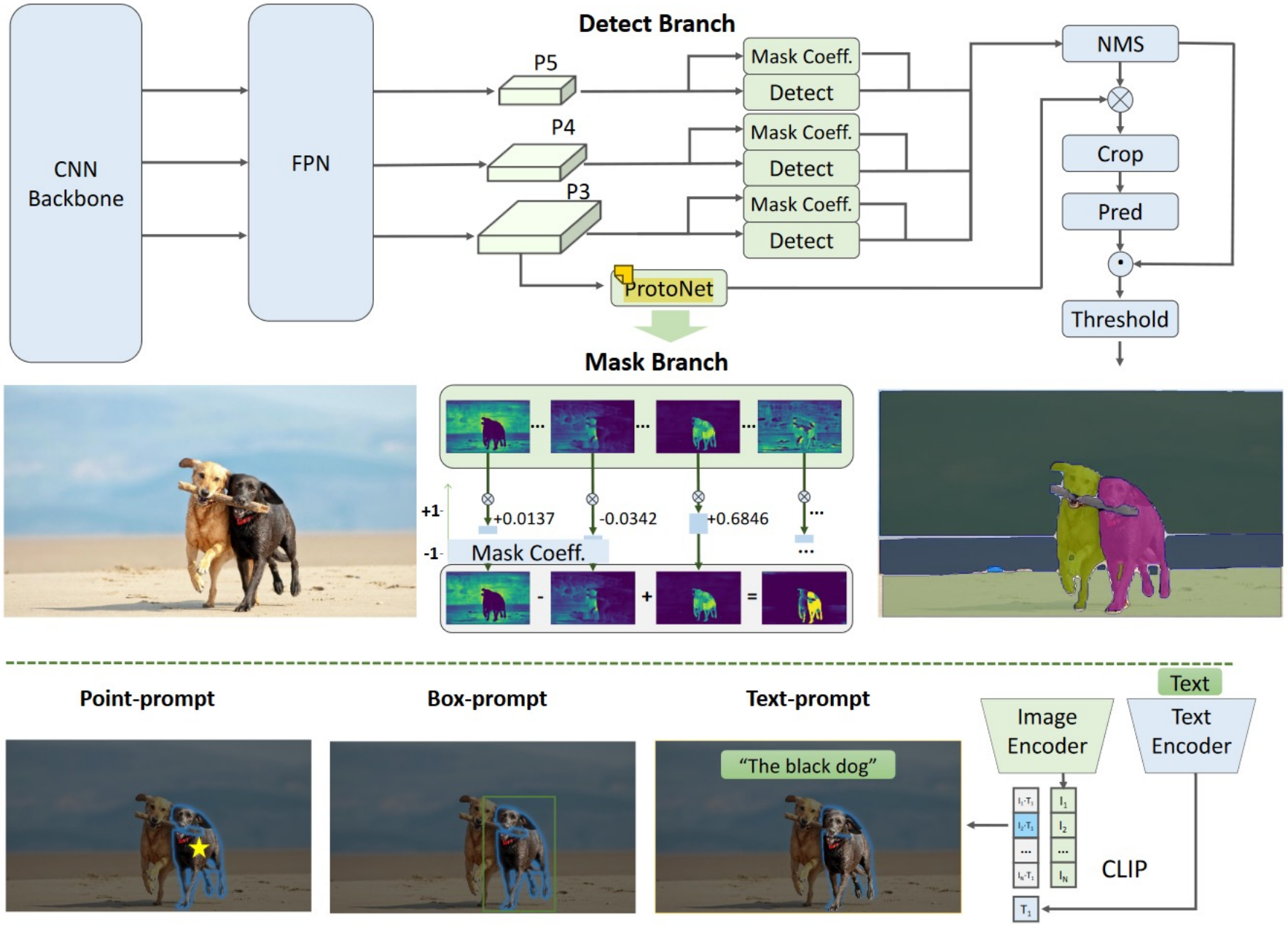

Encoder
Image
DCNN
Atrous Conv
1x1 Conv
3x3 Conv rate 6
3x3 Conv rate 12
3x3 Conv rate 18
Image Pooling
1x1 Conv
Decoder
Low-Level Features
Upsample by 4
1x1 Conv
Concat
3x3 Conv
Upsample by 4
Prediction

```
images, bboxes, gt_masks = data
batch_size = images.size(0)
pred_masks, iou_predictions = model(images, bboxes)
```

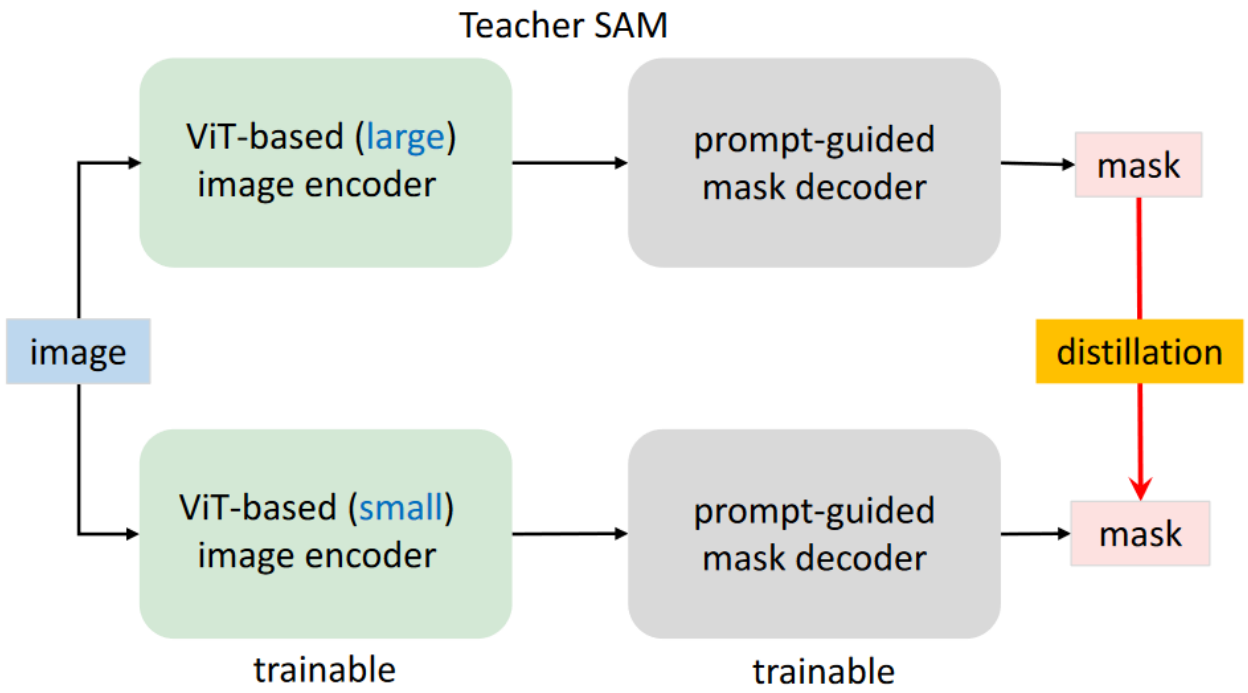
Using bbox prompt, the table below shows the mask quality of the original and fine-tuned models.

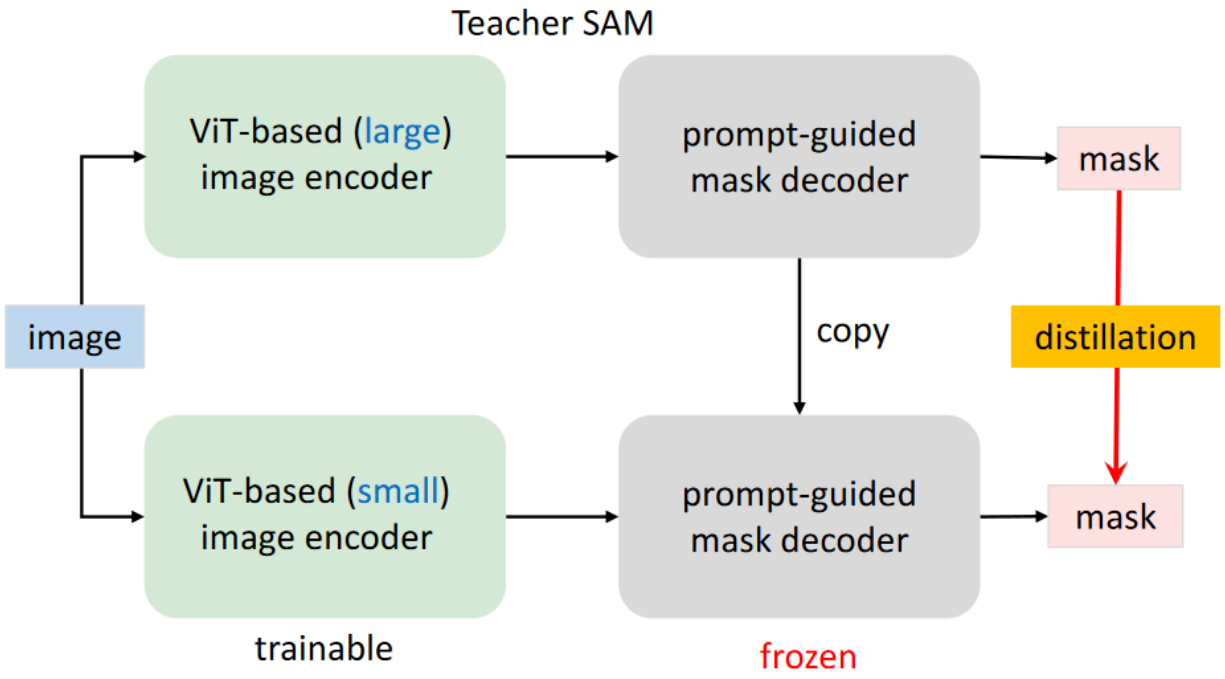
Dataset	Type	Mean IoU	Mean F1	Epoch
COCO2017	Original SAM	0.7978	0.8749	
COCO2017	Fine-tuned	0.8070	0.8816	2
TrashCan 1.0	Original SAM	0.6643	0.7808	
TrashCan 1.0	Fine-tuned	0.7888	0.8738	5
TrashCan 1.0	Fine-tuned	0.8000	0.8795	14
WGISD	Original SAM	0.8536	0.9178	
WGISD	Fine-tuned	0.8732	0.9298	19

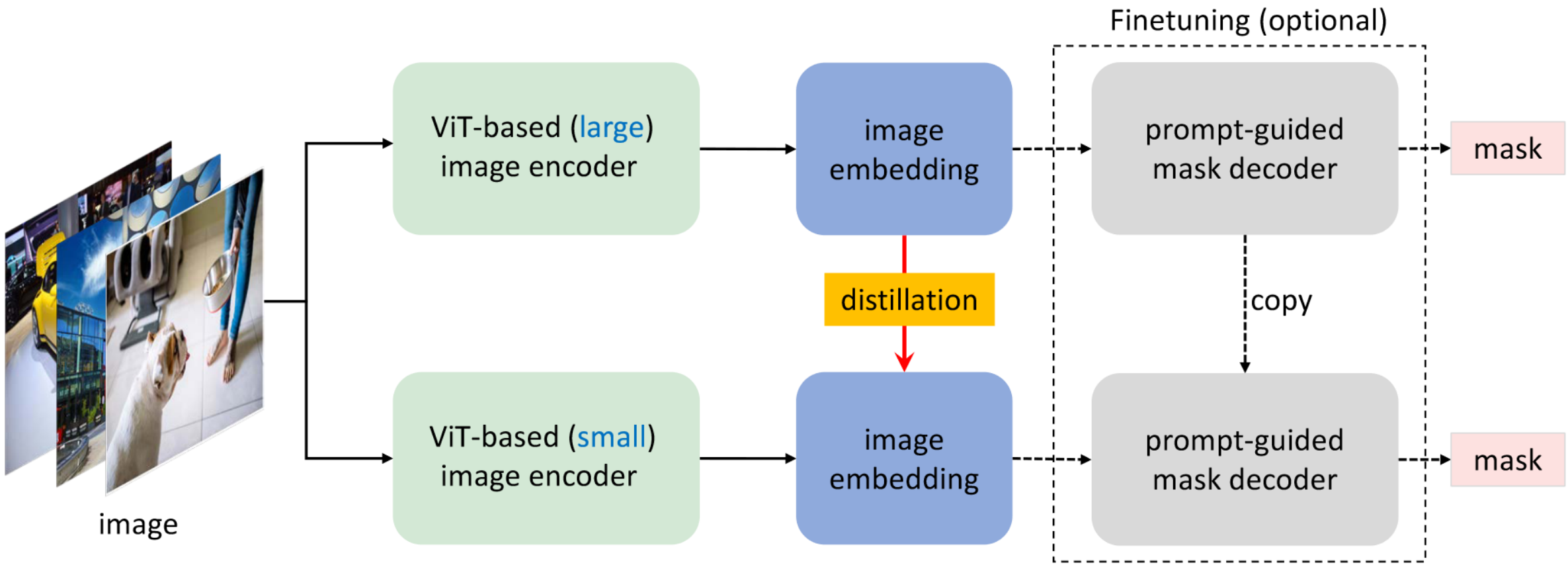
Fast SAM


CNN Backbone
FPN
Detect Branch
P5
P4
P3
Mask Coeff.
Detect
Mask Coeff.
Detect
Mask Coeff.
Detect
ProtoNet
NMS
Crop
Pred
Threshold
Mask Branch
+1
-1
+0.0137
-0.0342
+0.6846
Mask Coeff.
Point-prompt
Box-prompt
Text-prompt
"The black dog"
Text
Image Encoder
Text Encoder
CLIP

Faster SAM


Teacher SAM
ViT-based (large) image encoder
prompt-guided mask decoder
mask
image
distillation
ViT-based (small) image encoder
prompt-guided mask decoder
mask
trainable
trainable

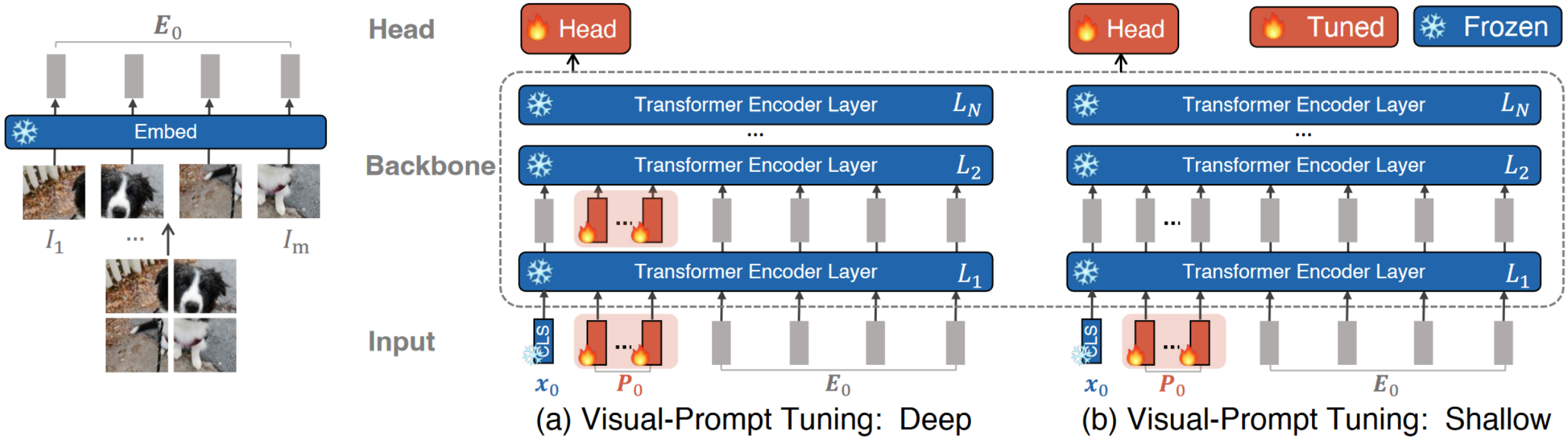

Teacher SAM
ViT-based (large) image encoder
prompt-guided mask decoder
mask
image
copy
distillation
ViT-based (small) image encoder
prompt-guided mask decoder
mask
trainable
frozen



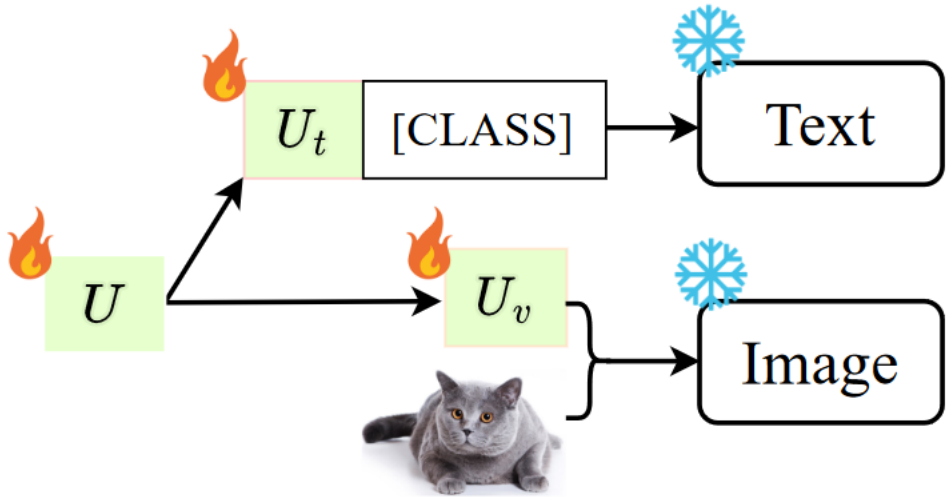
⭐ **How is MobileSAM trained?** MobileSAM is trained on a single GPU with 100k datasets (1% of the original images) for less than a day. The training code will be available soon.

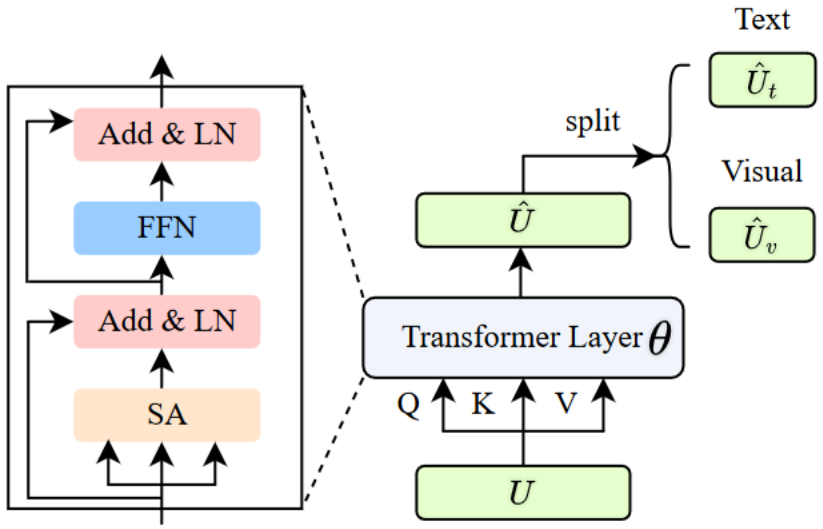
⭐ **How to Adapt from SAM to MobileSAM?** Since MobileSAM keeps exactly the same pipeline as the original SAM, we inherit pre-processing, post-processing, and all other interfaces from the original SAM. Therefore, by assuming everything is exactly the same except for a smaller image encoder, those who use the original SAM for their projects can **adapt to MobileSAM with almost zero effort**.

Visual prompt tuning

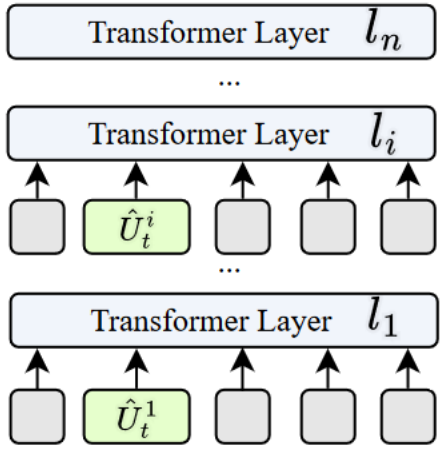


UNIFIED VISION AND LANGUAGE PROMPT LEARNING (V-L model tuning)

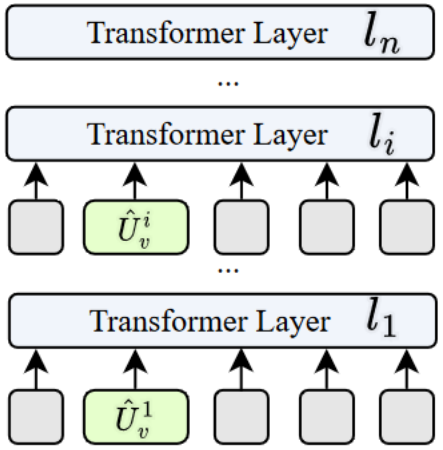




(a) Unified Prompt

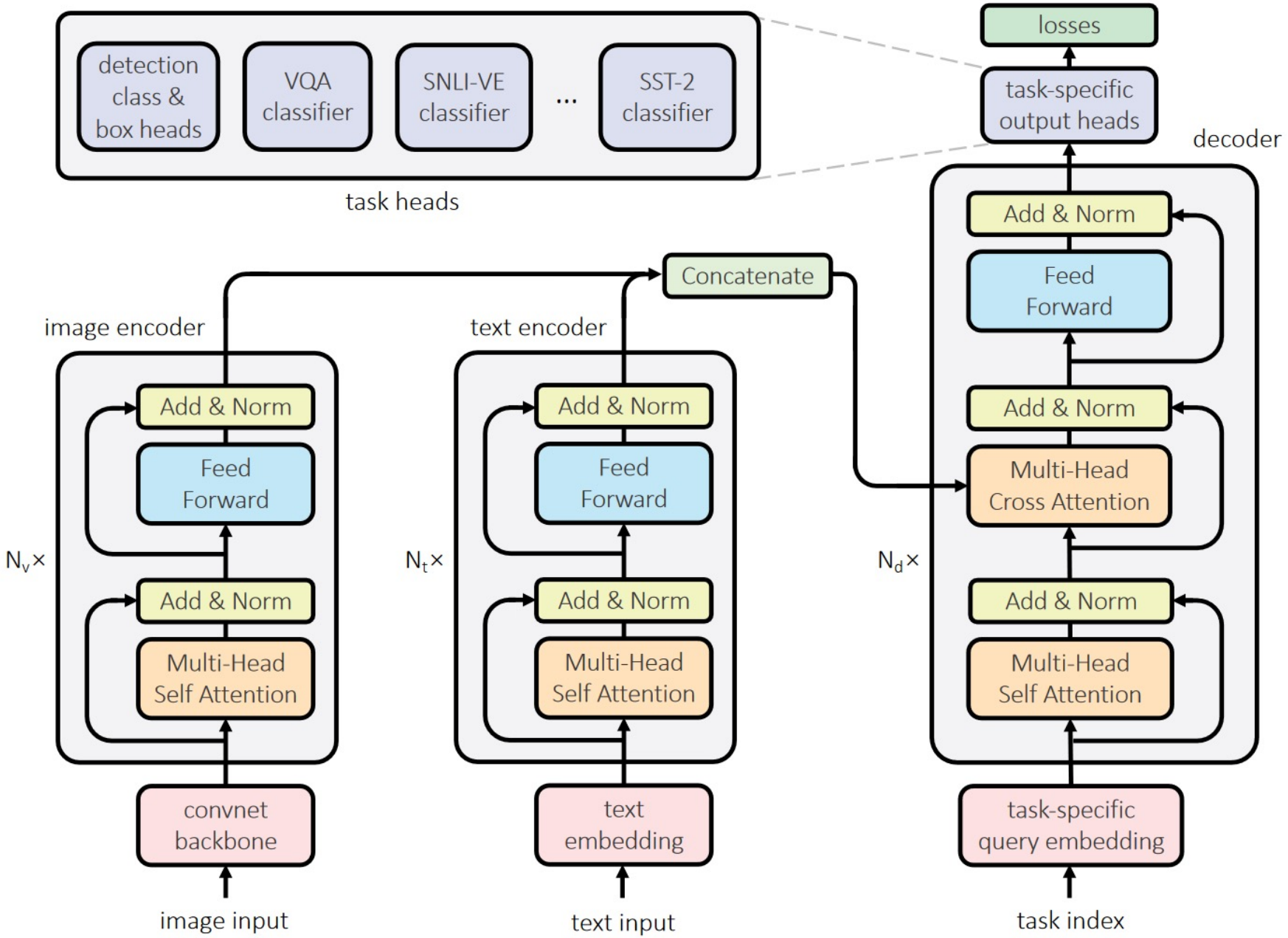


(b) Text Encoder

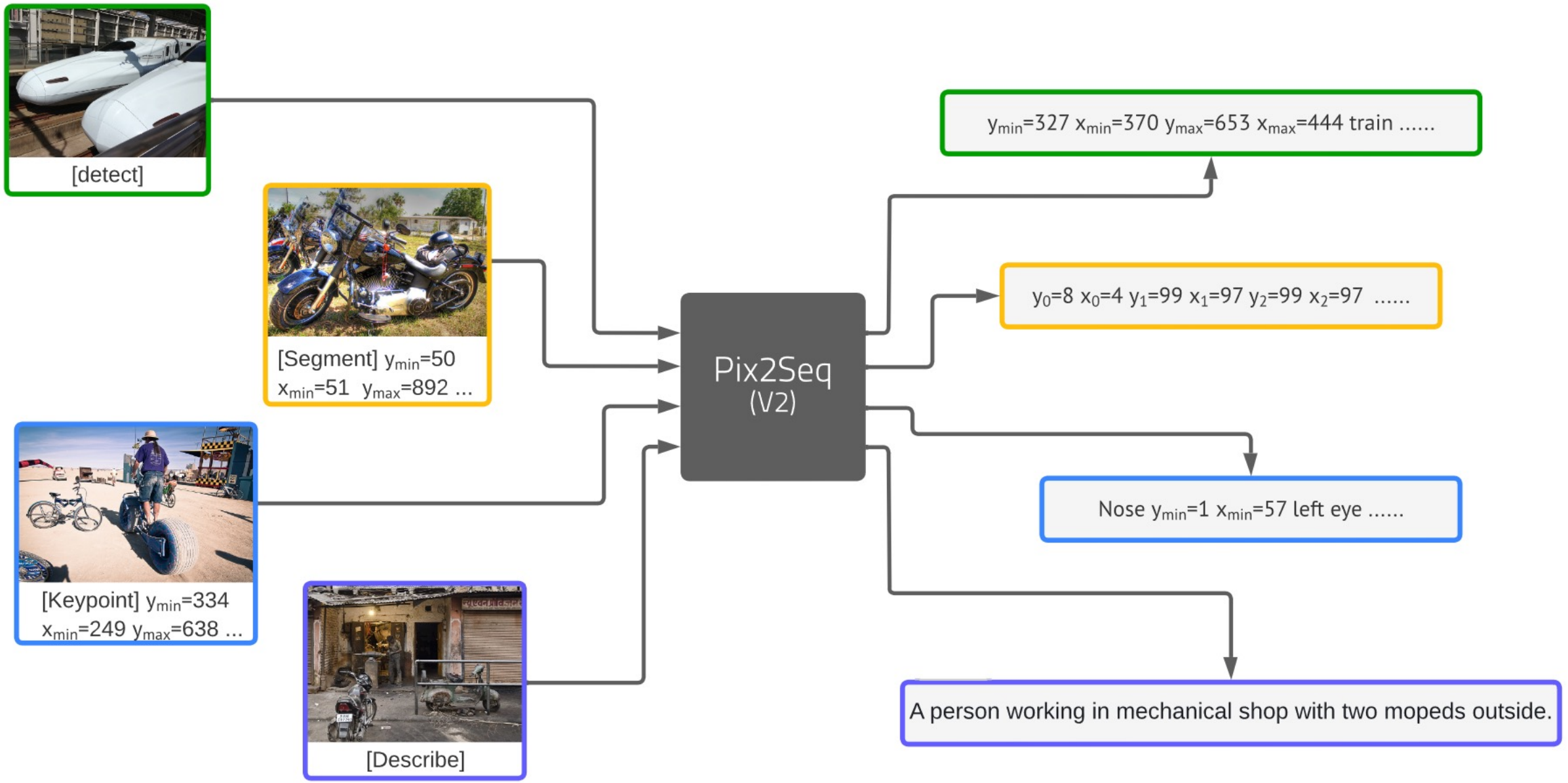


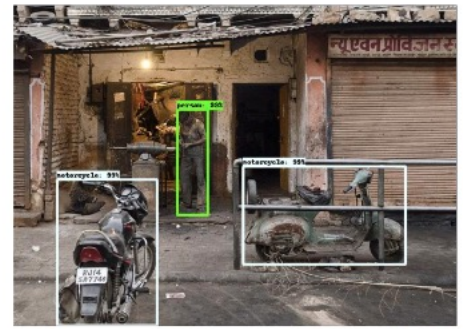
(c) Image Encoder

UniT: Multimodal Multitask Learning with a Unified Transformer



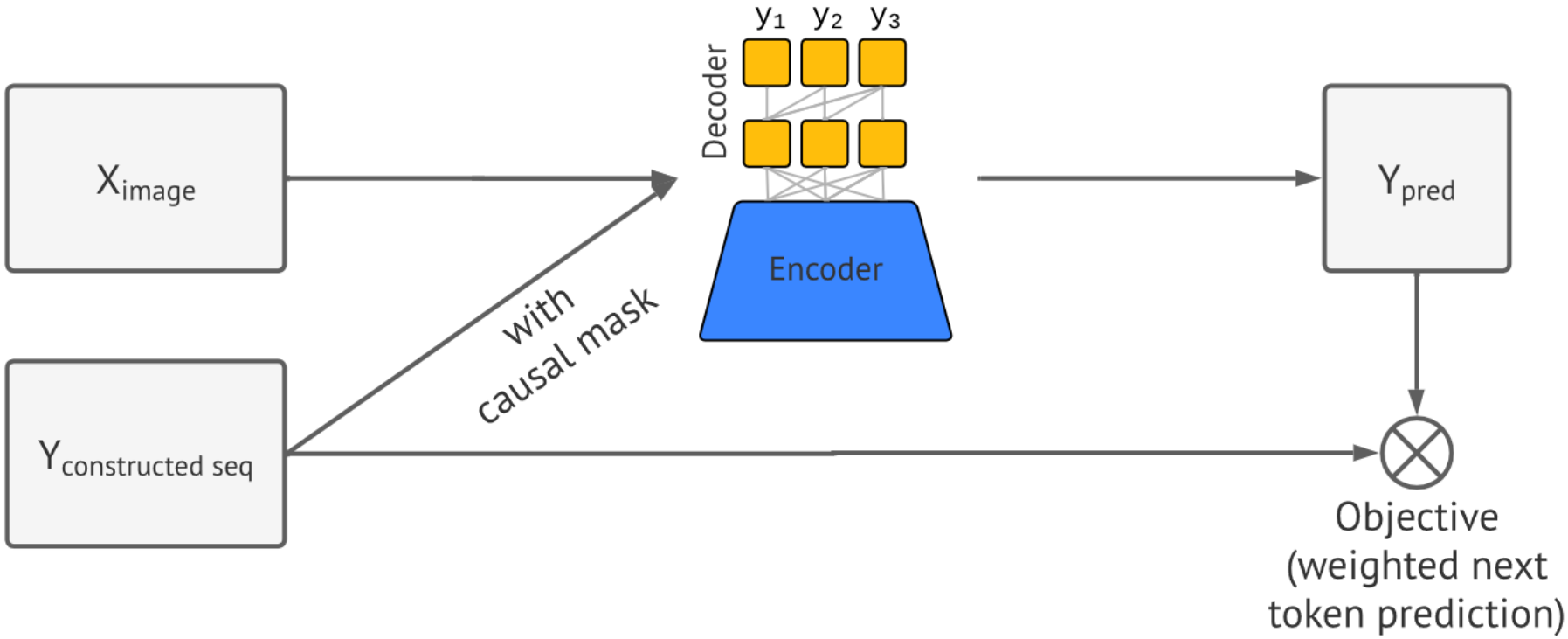
Pix2seq A Unified Sequence Interface for Vision Tasks



	Task prompt	Task output	Output visualization
Input image 	[Detect]	y_{min}=327 x_{min}=370 y_{max}=653 x_{max}=444 person	
	[Segment] y_{min}=503 x_{min}=518 y_{max}=805 y_{max}=892 Motocycle	y_0=553 x_0=599 y_1=788 y_1=664	
	[Keypoint] y_{min}=327 x_{min}=370 y_{max}=653 x_{max}=444 person	Nose y_{min}=1 x_{min}=57 left eye	
	[Describe]	A person working in mechanical shop with two mopeds outside.	

- For object detection, we follow [7] and convert bounding boxes and object descriptions into a sequence of discrete tokens by quantizing the continuous image coordinates. Specifically, an object is represented as a sequence of five discrete tokens, i.e. $[y_{\text{min}}, x_{\text{min}}, y_{\text{max}}, x_{\text{max}}, c]$, and multiple objects are randomly ordered each time a training image is sampled and serialized into a single sequence.
- For instance segmentation, instead of per-pixel masks, we predict the polygon [5] corresponding to the instance masks as a sequence of image coordinates conditioned on a given object instance. Again, we quantize the coordinates into discrete tokens. And to turn polygon into a sequence, we randomly select a starting point for the start token each time a training image is sampled. If there are multiple polygons for the same instance, we concatenate sequences of individual polygons with a separator token in between, so that every instance has a single corresponding sequence.
- For keypoint prediction, we predict a set of keypoints as a sequence of quantized image coordinates conditioned on a given person instance. Specifically, the sequence of keypoints can be encoded as $[y_{\text{keypoint 1}}, x_{\text{keypoint 1}}, y_{\text{keypoint 2}}, x_{\text{keypoint 2}}, \cdots]$. One may also use a keypoint label (e.g., nose, let eye, right eye) before each (y, x)-coordinates so their ordering does not need to be fixed but we opt for simplicity given that there are only a small fixed set of 14 person keypoints in the COCO dataset we consider. When certain keypoints are occluded, their coordinate tokens are replaced with a special occlusion token.

- For bounding boxes, following [7], we split predicted sequences into tuples of 5 tokens to get coordinate tokens and a class token, and dequantize coordinate tokens to get the bounding boxes.
- For instance segmentation, we dequantize the coordinate tokens corresponding to each polygon, and then convert them into dense masks. The model is not trained with any geometry-specific regularizers per se, and as such the output polygonal masks can be somewhat noisy. To reduce the noise we find it helpful to sample multiple sequences and then average the masks, followed by a simple threshold to obtain a single binary mask.
- For keypoint detection, we directly dequantize the image coordinate tokens of the keypoints.
- For captioning, we directly map the predicted discrete tokens into text.

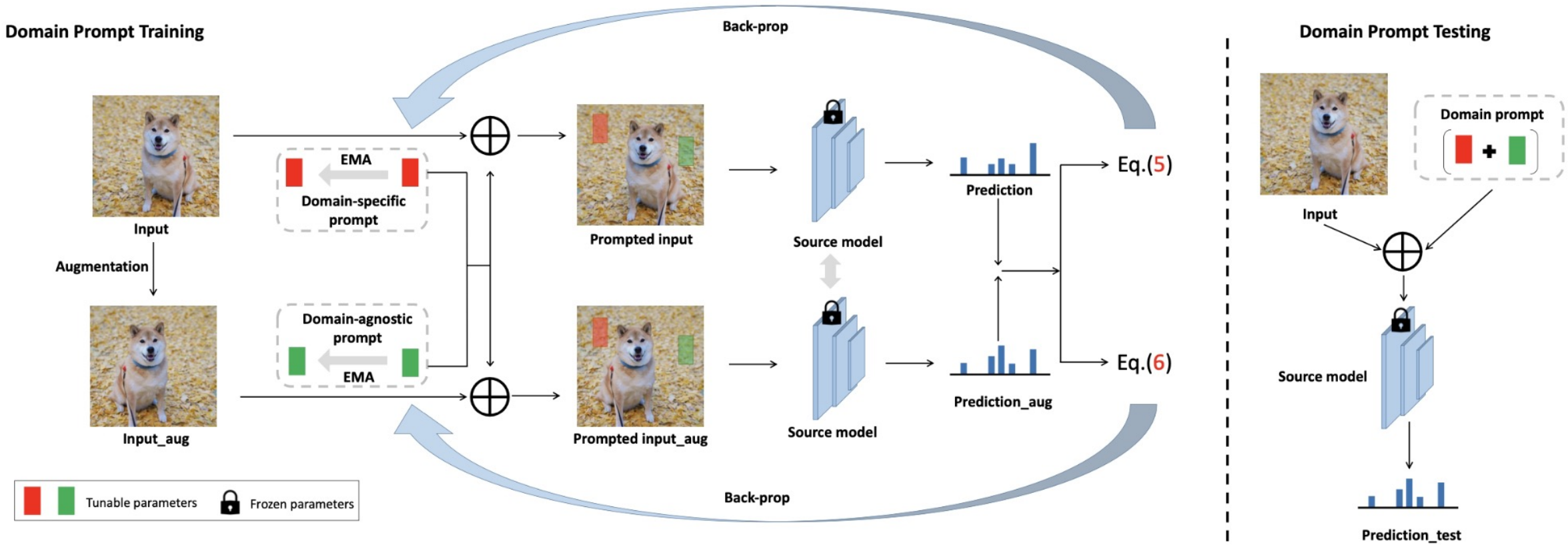


$$\text{maximize} \sum_{j=1}^{L} \boldsymbol{w}_j \log P(\boldsymbol{y}_j | \boldsymbol{x}, \boldsymbol{y}_{1:j-1}) \ ,$$

Image encoder 蒸馏stu

Stu visual prompt

1.如何蒸馏 pkd
2.Visual prompt， 将visual prompt 加到图片上



$$\mathcal{L}_{\omega_\phi}(x_p^T) = -\sum_C f'_{\theta_t}(h(x_p^T))(log f_{\theta_t}(x_p^T)), \quad (5)$$

$$\mathcal{L}_{\psi_\delta}(x_p^T) = -\sum_C f'_{\theta_t}(h(x_p^T))(log f_{\theta_t}(x_p^T)) + \mathcal{L}(\psi_\delta), \quad (6)$$

$$\mathcal{L}_o(x_p^T) = \mathcal{L}_{\psi_\delta}(x_p^T) + \mathcal{L}_{\omega_\phi}(x_p^T), \quad (7)$$

$$\Lambda_i^\tau = \sum_{v<\tau} \frac{\eta_i^\nu}{(\delta_i^\nu)^2 + \xi},$$

$$\mathcal{L}(\psi_\delta) = \alpha \sum_{\theta \in \Theta} \Lambda_i^\tau ||\theta - \theta^*||_2^2,$$