Appendix

In this Appendix, we first provide more training details of VITA (Appendix A). In addition, the inference procedure is explained in Appendix B.

A Training Details

A.1 Implementation

We use 4 NVIDIA A100 GPUs with 40GB of memory (8 A100 GPUs when using a Swin [22] backbone), and activate Automatic Mixed Precision (AMP) provided by PyTorch. Our training pipline is two-stage. We first pretrain the model for image instance segmentation on COCO [20] `train` set using the batch size of 16 and by setting the number of input frames to $T = 1$. Then, we finetune the pretrained model on the VIS `train` sets (YouTube-VIS 2019 [32], YouTube-VIS 2021, and OVIS [24]) with pseudo-videos augmented from COCO images (see Appendix A.1.1). We use the batch size of 8 and set each input clip to be length of $T = 6$. Considering the difficulty and varying number of training videos included in each dataset, we set up different training iterations for each VIS dataset - 130k, 160k, 110k with decay of learning rates at 75k, 100k, 50k for YouTube-VIS 2019, 2021, and OVIS, respectively. And both Object Encoder and Object Decoder in VITA follow the standard Transformer encoder and decoder architectures suggested in DETR [5]. However, we just switch the order of self- and cross-attention in Object Decoder to make video queries learnable, and eliminate dropouts to make computation more efficient, as discussed in Mask2Former [7].

A.1.1 Pseudo-video generation

During training, we follow SeqFormer [30] to generate pseudo-videos from a single image. Given a single image, we first resize the short side of an image to one of the 400, 500 and 600 pixels while maintaining its ratio. Then, the image is randomly cropped T times to a size in the range [384, 600] to create a pseudo-video of length T. Finally, the cropped images are resized to a shorter edge to be randomly chosen from [288, 512] pixels with a step of 32 pixels.

A.2 Loss function

The final loss function of our frame-level detector [7], denoted by $\mathcal{L}_f$ in the main paper, is largely composed of two terms: mask-related loss and categorical loss. The mask-related loss is again consists of $\mathcal{L}^f_{ce}$ and $\mathcal{L}^f_{dice}$, each representing a binary cross-entropy loss and a dice loss, respectively. Then, the final loss $\mathcal{L}_f$ is a combination of a categorical loss (the cross entropy) and the mask-related loss $\mathcal{L}_f = \lambda_{cls}\mathcal{L}^f_{cls} + \lambda_{ce}\mathcal{L}^f_{ce} + \lambda_{dice}\mathcal{L}^f_{dice}$ and we set $\lambda_{cls} = 2$, $\lambda_{ce} = 2$, and $\lambda_{dice} = 5$, respectively.

For the $\mathcal{L}_v$ calculated from video-level results generated by VITA, we employ the same hyper-parameters as frame-level losses: $\mathcal{L}_v = \lambda_{cls}\mathcal{L}^v_{cls} + \lambda_{ce}\mathcal{L}^v_{ce} + \lambda_{dice}\mathcal{L}^v_{dice}$. Note that, for $\mathcal{L}^v_{ce}$ and $\mathcal{L}^v_{dice}$, we extend the functions of $\mathcal{L}^f_{ce}$ and $\mathcal{L}^f_{dice}$ to the temporal axis, just as IFC [15] did.

A.3 Building VITA on Mask2Former

Mask2Former uses 9 decoder layers where output frame queries from each layer can be used as an input for VITA. However, using the outputs from all 9 layers during training leads to the lack of GPU memory. Therefore, we use the outputs from the last 3 layers for training VITA.

B Inference procedure

In Tab. 4 in the main paper, we measured the maximum number of frames that each model can infer at once. To further specify the process of measuring the numbers, we provide simplified PyTorch-style inference pseudo-codes of both VITA and Mask2Former-VIS in Tab. 8 and Tab. 9 respectively. For fair comparison, we modified the inference procedure of previous methods to collect backbone features of each frame sequentially. The strategy prevents the methods from a memory explosion until entering each VIS prediction module. The most noticeable difference is that VITA collects only `frame_queries` and `mask_features` of each frame from our frame-level detector [7] denoted by the function `mask2former()` (line 2-12 in Tab. 8). Then, the `frame_queries` for the entire video become the input of Object Encoder (line 19 in Tab. 8). On the other hand, previous Transformer-based offline VIS models (*e.g.*, Mask2Former-VIS), first aggregate the backbone features of entire video and takes it as inputs for the VIS model, the function `mask2former_vis()` (line 3-20 in Tab. 9). After that, both of methods generate their video-level predictions by using their `vq` (video queries) and `mask_features`.

Table 8: PyTorch-style inference pseudo-code of VITA.

```
1  def vita(video):
2      frame_queries = []
3      mask_features = []
4
5      for frame in video:
6          feats = backbone(frame)
7          fq, mf = mask2former(
8              feats
9          )
10
11         frame_queries.append(fq)
12         mask_features.append(mf)
13
14     """
15     VITA only aggregates
16     frame queries for its
17     remaining computations.
18     """
19     fq = object_encoder(
20         frame_queries
21     )
22     vq = object_decoder(fq)
23
24     w = mask_head(vq)
25     pred_mask = []
26     for mf in mask_features:
27         # w.shape: (Nv x C)
28         # mf.shape: (C x H x W)
29         _mask = w @ mf
30
31         pred_mask.append(_mask)
32
33     # Nv x (K+1)
34     pred_cls = cls_head(vq)
35
36     # Nv x T x H x W
37     pred_mask = torch.stack(
38         pred_mask, dim=1
39     )
40
41     return pred_cls, pred_mask
```

Table 9: PyTorch-style inference pseudo-code of Mask2Former-VIS [6].

```
1  def previous_methods(video):
2
3      frame_features = []
4
5      for frame in video:
6          feats = backbone(frame)
7          frame_features.append(
8              feats
9          )
10
11     """
12     Previous approaches receive
13     either multi or single scale
14     feature map at once for their
15     encoder/decoder layers.
16     """
17     vq, mask_features =\
18         mask2former_vis(
19             frame_features
20         )
21
22
23
24     w = mask_head(vq)
25     pred_mask = []
26     for mf in mask_features:
27         # w.shape: (Nv x C)
28         # mf.shape: (C x H x W)
29         _mask = w @ mf
30
31         pred_mask.append(_mask)
32
33     # Nv x (K+1)
34     pred_cls = cls_head(vq)
35
36     # Nv x T x H x W
37     pred_mask = torch.stack(
38         pred_mask, dim=1
39     )
40
41     return pred_cls, pred_mask
```