A. Implementation Details

A.1. Model Implementation

Text encoder. We adopt the Contrastive Language–Image Pre-training (CLIP) [31] model as the text encoder throughout, which supports variable-length text labels by design. The pre-trained and frozen CLIP-ViT-B/32 model was used, which produces a 512-dimensional embedding vector for each text label. In this paper, we simply use the class names (mostly containing 1 or 2 words, *e.g.*, person, traffic sign) defined in the datasets.

Type-I (SegFormer). We adopt the SegFormer [39] implementation in the `MMSegmentation` [47] codebase to perform whole-to-part segmentation for Type-I requests. Nota that SegFormer can be freely replaced by any other visual feature extractors. We additionally provide an illustration of generating language-based queries and performing vision-language interaction for Type-I requests, as shown in Figure 8. SegFormer uses a series of Mix Transformer encoders (MiT) with different sizes as the visual backbones. In this paper, we mainly adopt the lightweight model (MiT-B0) for fast development and the largest model (MiT-B5) towards better performance. The output feature maps have 512 channels, which can be directly interacted with the text embedding vectors. We additionally apply a projection module (four fully-connected layers) on the text features for better feature alignment. Note that the categorical logits after vision-language feature interaction is usually divided by a temperature parameter τ, $\mathbf{u}_{w,h} = (\mathbf{E}^\top \cdot \mathbf{f}_{w,h})/\tau$, where τ controls the concentration level of the distribution [20, 31]. We follow CLIP to set τ as a learnable parameter, which is initialize to be 0.07 in the start of the training stage.

Type-II (CondInst). We adopt the CondInst [35] implementation in the `AdelaiDet` [51] codebase to perform instance segmentation. Nota that CondInst can also be replaced by other instance segmentation models (*e.g.*, Mask R-CNN [12] or SOLOv2 [53]) with marginal modification. CondInst treats all feature locations on multiple feature pyramids as instance proposals, which is naturally compatible with the design of the probing-based inference. We additionally provide an illustration of how Type-II requests are processed with CondInst, as shown in Figure 9. Since the output feature maps (from the last layer of classification branch) have 256 channels, we apply a linear projection to transform the dimension of text embedding from 512 to 256 to perform feature interaction. We mainly adopt ResNet-50 as the visual backbone of CondInst. We empirically observed that the results were improved slightly ($\sim 0.3\%$ mAP) with a larger backbone, ResNet-101 – we conjecture that the limited improvement lies in the small dataset size of CPP and ADE20K. In addition, we observed that the instance segmentation model is more sensitive to the choice of τ. For example, using $\tau = 0.07$ leads to sub-optimal results. We empirically found that setting $\tau = 1.0$ and adding a learnable bias term (partially following GLIP [22]) works consistently better.

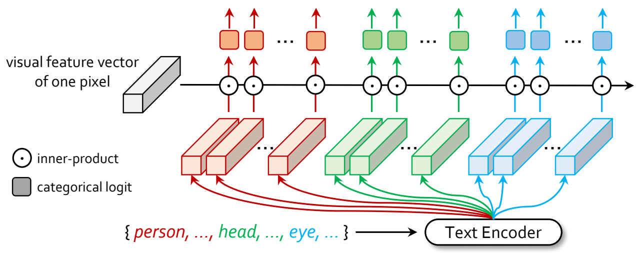
visual feature vector of one pixel
inner-product
categorical logit
{ person, ..., head, ..., eye, ... }
Text Encoder

Figure 8. An illustration of generating language-based queries and performing vision-language interaction for Type-I requests (the case of *one pixel* is plotted for simplicity). The visual feature vector and the text embedding have the same dimension (*e.g.*, 512). The input text labels can be freely replaced by any texts of arbitrary length. See the main texts for more details.

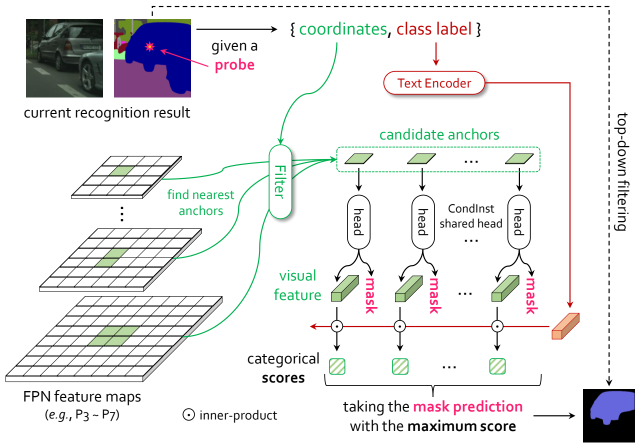

Figure 9. An illustration of how Type-II requests are processed with CondInst. The probe is sampled (or clicked by user) from the semantic region of current recognition result, aiming at segmenting the instance that occupies this probe. The *filter* is used to select the spatially related anchors (or named proposals) by the positional embedding $\mathbf{p}_k$ (equivalents to probe coordinates in our implementation), and the preserved anchors are used for subsequent prediction. Prediction with the highest categorical score (obtained by inner-product with text embedding) is chosen as the final result. See the main texts for more details.

A.2. Model Optimization

For Type-I requests, we almost follow the same training protocol as SegFormer, except that classes of different semantic levels (*e.g.*, objects, object parts, parts of parts) are jointly trained in one model (see Figure 3). The MiT encoder was pre-trained on Imagenet-1K and the decoder was randomly initialized. For data augmentation, random resizing with ratio of $0.5 \sim 2.0$, random horizontal flipping, and random cropping (1024×1024 for CPP and 512×512 for ADE20K) were applied. The model was trained with

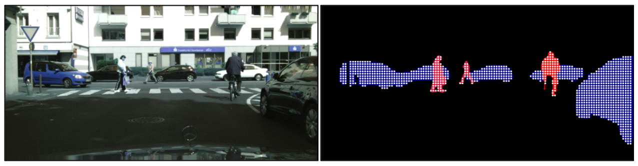
Figure 10. Illustration of sampling probes with *stride* (*e.g.*, 16).

Table 5. Non-probing-based instance segmentation results w.r.t. different *strides*.

AP (%)	**Non-Probing (w.r.t. *stride*)**			
	1	8	16	32
w/o	37.8	37.8	36.8	33.6
w/ mask samp.	38.5	38.5	37.8	35.1

AdamW optimizer for $160K$ iterations on 8 NVIDIA Tesla V100 GPUs. The initial learning rate is 0.00006 and decayed using the poly learning rate policy with the power of 1.0. The batch size is 8 for CPP and 16 for ADE20K.

For Type-II requests, since CondInst [35] have not reported results on Cityscapes and ADE20K, we implemented by ourselves. Due to the small dataset size, we initialized the model with the MS-COCO [25] pre-trained CondInst, which increases the results by about 1% AP. The model was trained with SGD optimizer on 8 NVIDIA Tesla V100 GPUs. For CPP, we used the same training configurations as Mask R-CNN on Cityscapes [54] and the model was trained for $24K$ iterations with the batch size of 8. For ADE20K, the maximum number of proposals sampled during training was increased from 500 to 1,000 since more instances appeared in an image than MS-COCO. The model was trained for $40K$ iterations with the initial learning rate being 0.01 and decayed at $30K$ iterations. During training, random resizing with ratio of $0.5 \sim 2.0$ and random cropping (640×640) were applied. Other configurations are the same as the original CondInst.

A.3. Non-Probing Segmentation

The non-probing-based inference is used to fairly compare against the conventional instance segmentation approaches, *i.e.*, finding all instances *at once*. For this purpose, we densely sample a set of probing pixels based on the semantic segmentation results. Specifically, we regularly sample points with a fixed *stride* in the whole image, and keep the points inside the corresponding semantic regions, as illustrated in Figure 10 (white dots indicate the sampled probes). Each sampled point is viewed as a probing pixel to produce a candidate instance prediction (see Appendix A.4 and Figure 9 for details). Finally, the results are filtered with NMS (using a threshold of 0.6 as the same as CondInst). Intuitively, the above procedure can be viewed as an improved CondInst by replacing the builtin classification branch as a standalone pixel-wise classification model. We present the results with respect to different *strides* in Table 5. As shown, the denser the sampled probes, the higher the mask AP. In addition, enabling mask sampling during training further improves the results (see Appendix A.4 for details of mask sampling). We mainly use the stride of 16 in experiments.

A.4. Probing Segmentation

The probing-based inference, a more flexible setting, is used to simulate the user click to place a probing pixel that lies within the intersection of the predicted semantic region and the ground-truth instance region – if the intersection is empty, the instance is lost (*i.e.*, IoU is 0%). Specifically, we first compute the mass center (a_{mass}, b_{mass}) and the bounding box $\mathcal{B} = (a_0, b_0, a_1, b_1)$ of the intersection region. The actual sampling bounding box $\hat{\mathcal{B}} = (\hat{a}_0, \hat{b}_0, \hat{a}_1, \hat{b}_1)$ is determined by a hyper-parameter $\gamma \in [0, 1]$, which controls the centerness of the probing pixels:

$$\begin{aligned}
\hat{a}_0 &= a_{mass} - \gamma(a_{mass} - a_0), \\
\hat{a}_1 &= a_{mass} + \gamma(a_1 - a_{mass}), \\
\hat{b}_0 &= b_{mass} - \gamma(b_{mass} - b_0), \\
\hat{b}_1 &= b_{mass} + \gamma(b_1 - b_{mass}).
\end{aligned} \tag{2}$$

The probe is randomly sampled from the intersection of $\hat{\mathcal{B}}$ and the original sampling region. If no candidate pixel lies in that region (since the mass center may outside the instance), the probe is instead sampled from $\hat{\mathcal{B}}$ only. The probe lies exactly on the mass center if $\gamma = 0$, and the probe is uniformly distributed on the instance if $\gamma = 1$, otherwise, the sampling strategy lies between two extreme situations.

During inference, for each Type-II request we have a triplet $\{a, b, c\}$ (see Section 4 for details, the subscript k is omitted here for simplicity). To perform instance segmentation *by request*, we first compute the mapped feature locations $(a_f, b_f) = (\lfloor \frac{a}{s_f} \rfloor, \lfloor \frac{b}{s_f} \rfloor)$ for each FPN level, where s_f is the stride of the f-th FPN level, and $s_f \in \{8, 16, 32, 64, 128\}$ for CondInst. The five mapped feature locations are viewed as candidates for producing subsequent predictions. Finally, we only choose one prediction with the highest categorical score, where the scores are computed by the inner-product of the pixel-wise feature vectors (extracted from the FPN feature maps) and the text embedding vector of the target class c (generated by the text encoder). This process is illustrated in Figure 9. For probing-based inference, there is at most one prediction for an instance, which naturally eliminates the need of non-maximum suppression (NMS).

In Table 6, we present the mask AP with respect to to different γ. As shown, AP increases consistently with lower γ (*i.e.*, closer to the mass center). We observed that AP decreases dramatically with lager γ if the mask sampling

Table 6. Instance segmentation results with non-probing-based and probing-based inference on the CPP dataset. Results are produced using the semantic mask prediction of SegFormer-B5. The hyper-parameter γ controls the centerness of the probes.

AP (%)	**Non-Probing**	**Probing (w.r.t. γ)**			
		0.0	0.2	0.5	1.0
CondInst (R50) [35]	36.6	–	–	–	–
w/ CLIP	36.8	39.3	39.0	34.6	24.5
w/ CLIP & mask samp.	**37.8**	**39.4**	**39.1**	**37.4**	**33.5**

Table 7. Overall segmentation results on CPP, using non-probing-based inference and probing-based inference, respectively. * indicates that BPR [34] is used.

HPQ (%)	**Non-Probing**	**Probing (w.r.t. γ)**			
		0.0	0.2	0.5	1.0
SegFormer (B0) + CondInst (R50)	56.0	57.0	56.8	56.7	56.2
SegFormer (B5) + CondInst (R50)*	61.6	62.7	62.6	62.2	61.7

strategy was not used during training (the second row). We diagnosed the issue and found that some probes away from the instance center produced unsatisfying results, as shown in Figure 11. The reason lies in that CondInst only samples positive positions from a small central region of instance by design [35, 36], thus not all possible probes are properly trained. To address this issue, we instead sample positive positions from the entire ground-truth instance masks. The results are presented in Table 6 (the last row) and Figure 11 (the last column). As shown, by enabling mask sampling during training, the results are much more robust to the quality of probes. In addition to probing-based inference, we observed that mask sampling is also helpful for non-probing-based inference, as shown in Table 1.

B. Details of the CPP Experiments

B.1. Data Statistics

The Cityscapes Panoptic Parts (CPP) dataset [5] extends the popular Cityscapes [3] dataset with part-level semantic annotations. There are 9 part classes belonging to 5 scene-level classes are annotated for 2,975 training and 500 validation images in Cityscapes. Specifically, two human classes (person, rider) are labeled with 4 parts (*torso*, *head*, *arm*, *leg*), and three vehicle classes (car, truck, bus) are labeled with 5 parts (*chassis*, *window*, *wheel*, *light*, *license plate*). The CPP dataset provides exhaustive part annotations that each instance mask (belonging to the chosen 5 classes) is completely partitioned into the corresponding part masks. In our experiments, we use 19 semantic classes (8 out of 19 are thing classes for instance segmentation) and 9 non-duplicate part classes in total. Results on the CPP validation set are reported.

B.2. HPQ vs. PartPQ in CPP

In the scenario of CPP (only two hierarchies), the only difference between PartPQ and HPQ lies in computing the accuracy of the objects that have parts. PartPQ directly averages the mask IoU values of all parts, while HPQ calls for a recursive mechanism. CPP is a two-level dataset (*i.e.*, parts cannot have parts), and all parts are semantically labeled (*i.e.*, no instances are labeled on parts although some of them, *e.g.*, wheel of car, can be labeled at the instance level). In this scenario, (1) the HPQ of a part (as a leaf node) is directly defined as its mask IoU if the corresponding prediction is a true positive (IoU is no smaller than 0.5) and HPQ equals zero otherwise, and (2) since each part has only one unit, the recognition of each part has either a true positive (IoU is no smaller than 0.5) or a false positive plus a false negative (IoU is smaller than 0.5) – that said, the denominator of HPQ is a constant, equaling to the number of parts. As a result, the values of HPQ are usually lower than PartPQ (see Table 2).

B.3. Sampling incomplete annotations in CPP

In Table 3, we report the results of learning from incomplete annotations, where only a subset of annotations are preserved for training. For the setting (1), we preserve all semantic and instance annotations but randomly choose a certain portion of part annotations. Specifically, there are in total 161,182 annotated part-level masks in the CPP training set. We randomly sample a certain ratio (*e.g.*, $15\% \sim 50\%$) of part-level masks for training. For the setting (2), we first randomly sample a certain ratio (*e.g.*, $30\% \sim 75\%$) of scene-level masks (34,723 in total), and further sample the same ratio of part-level masks within the preserved scene-level masks, which is a more incomplete scenario. For both settings, evaluation was conducted on the validation set with complete part annotations. We find that ViRReq, without any modification, adapts to both settings easily.

B.4. More Quantitative Results

We additionally report the HPQ results on the CPP dataset with probing-based inference in Table 7 (while only non-probing-based results are reported in Table 2). We have by default used the instance segmentation results with CLIP and mask sampling strategy. Interestingly, although the non-probing-based setting surpasses the probing-based settings with large γ values in instance AP, it reports lower HPQ values because probing-based tests usually generate some false positives with low confidence scores – HPQ, compared to AP, penalizes more on these prediction errors.

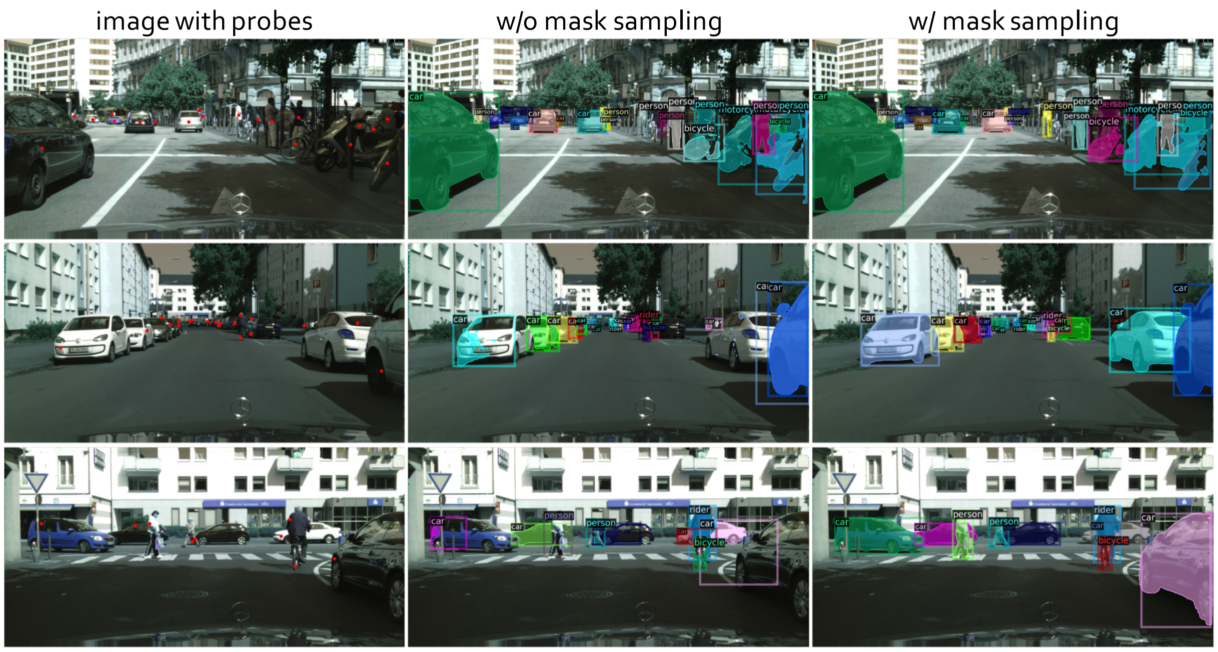

Figure 11. Results of using different types of *probes* (from top to bottom): mass center of instance ($\gamma = 0$), random points on the instance ($\gamma = 1$), handcrafted low-quality probes that closing to the instance boundaries. *Best viewed digitally with color.*

B.5. Details of Competitors

We introduce the details of the competitors [5, 23] compared in Table 2, which are the only two existing works that have reported results on CPP. PPS [5] first provided the CPP dataset and the PartPQ metric for the task of part-aware panoptic segmentation, and established several baselines by merging results of methods for the subtasks of panoptic and part segmentation. Panoptic-PartFormer [23] is a unified Transformer-based model that predicts different levels of masks jointly, which achieves significant improvement over PPS [5].

Note that although we provide a comparison in Table 2, but this work is essentially different to these methods, and our work is actually not optimized for higher results on the fully-annotated CPP dataset. The most important advantages of ViRReq are the abilities to learn complex hierarchies from incomplete annotations and adapt to new concepts easily, while the competitors [5, 23] do not have. The numerical differences (or benefits) in comparison may originate from various aspects (*e.g.*, backbone, training time), and it is difficult to compare with the competitors in a completely fair manner (*e.g.*, keeping the parameters/FLOPs/training epochs in a similar level) since both the methods are highly complicated.

B.6. Qualitative Results

We visualize some segmentation results of ViRReq on CPP, as shown in Figure 12. The results are obtained through probing-based inference, where the mass center was used as the probe (*i.e.*, $\gamma = 0$). The probing pixels are omitted in the figure for simplicity.

B.7. Open-Domain Recognition

We provide some open-domain segmentation results on CPP, as shown in Figure 13. The top three rows indicate anomaly segmentation, *i.e.*, finding unknown concepts in an image. Example images are taken from the Fishyscapes [46] dataset (an anomaly segmentation benchmark). These anomalies (*e.g.*, dog, box, *etc.*) were usually not detected by the regular segmentation model (see Figure 2 of [46]), but were found by our approach which was not specifically designed for this task. The last row involves compositional segmentation, *i.e.*, transferring part-level knowledge from one class (*e.g.*, car) to others (*e.g.*, caravan or trailer) without annotating new data but directly copying the sub-knowledge from the old class to new classes.

C. Details of the ADE20K Experiments

C.1. Data Preparation and Statistics

The ADE20K [43] dataset provides pixel-wise annotations for more than 3,000 semantic classes, including instance-level and part-level annotations. SceneParse150 is a widely used subset of ADE20K for semantic segmentation, which consists of 20,210 training and 2,000 valida-

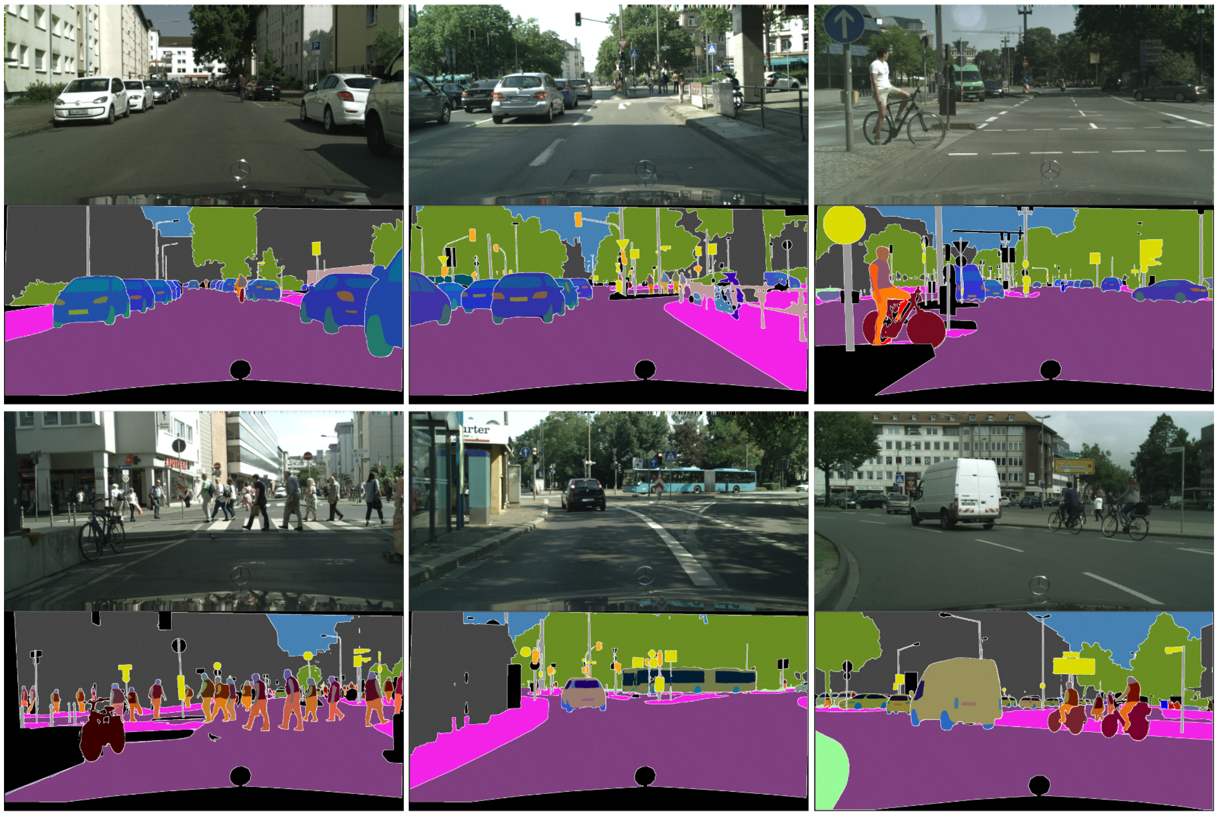

Figure 12. Visualization results of ViRReq on CPP. Results of different requests are merged together. *Best viewed digitally with color.*

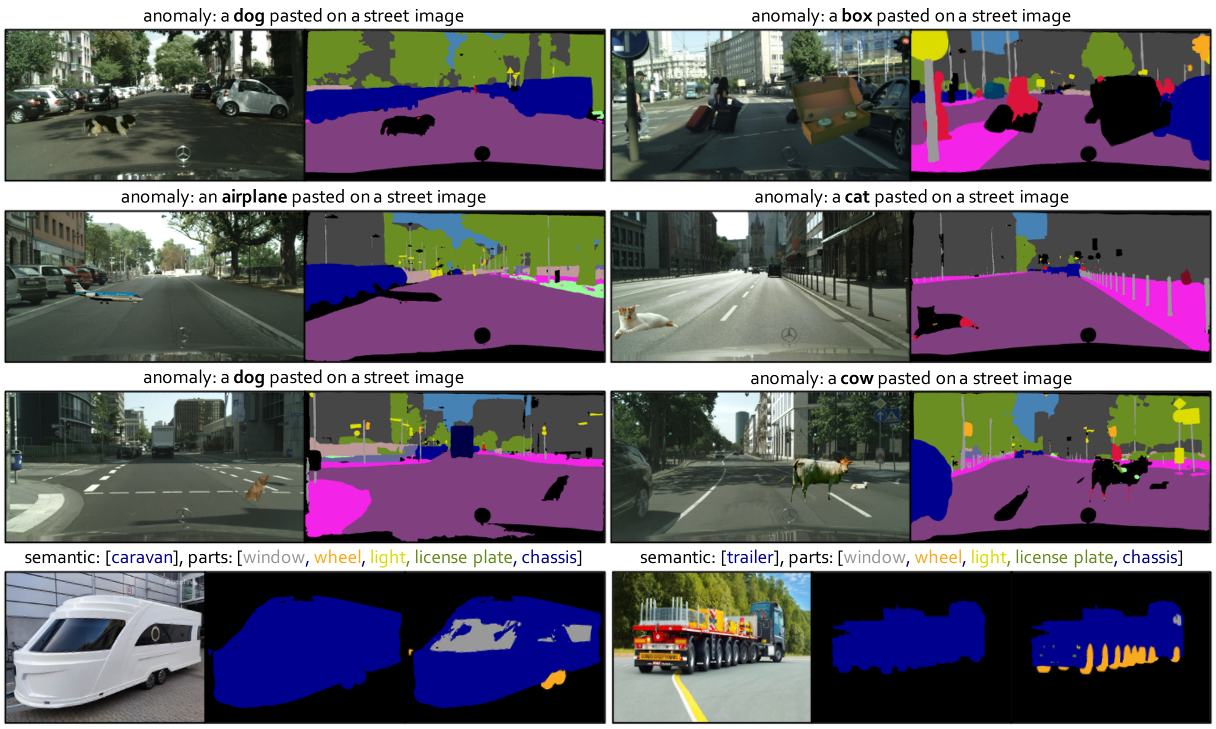
anomaly: a **dog** pasted on a street image
anomaly: a **box** pasted on a street image
anomaly: an **airplane** pasted on a street image
anomaly: a **cat** pasted on a street image
anomaly: a **dog** pasted on a street image
anomaly: a **cow** pasted on a street image
semantic: [caravan], parts: [window, wheel, light, license plate, chassis]
semantic: [trailer], parts: [window, wheel, light, license plate, chassis]

Figure 13. Examples of open-domain recognition on the CPP dataset: anomaly segmentation (top three rows), and compositional segmentation (the last row), respectively. Black region corresponds to others.

tion images covering 150 most frequent classes. For instance segmentation, 100 foreground object classes are chosen from 150 classes, termed InstSeg100 (see Section 3.4 of [43]), while few works have reported results on InstSeg100. As for part segmentation, we found the annotations are significantly sparse and incomplete [2]. There are 289 non-duplicate part classes belonging to the 100 instance classes [3]. The labeling ratio for part classes is pretty low: only 15% of instances have part annotations on averaged (0.03% $\sim$ 69.3% for each instance class individually). In our experiments, we first filter the part-of-objects classes that the number of occurrences is no fewer than 100, resulting in 82 non-duplicate part-level classes belonging to 40 instance classes. Then, we further filter the part-of-parts classes that the number of occurrences is no fewer than 100, resulting in 29 non-duplicate part-of-parts classes belonging to 17 upper-level part classes. We conjecture that the sparse annotation property is the main reason that no prior works have reported qualitative results for part-level segmentation on this dataset. Results on the ADE20K validation set are reported. For the vocabulary used in the experiments, semantic and instance classes can be easily found in the original dataset [43], and we additionally list the part-of-object classes as follows in the format of *instance class name (the number of part classes): [part class names]*.

- bed (4): [footboard, headboard, leg, side rail]
- windowpane (5): [pane, upper sash, lower sash, sash, muntin]
- cabinet (7): [drawer, door, side, front, top, skirt, shelf]
- person (13): [head, right arm, right hand, left arm, right leg, left leg, right foot, left foot, left hand, neck, gaze, torso, back]
- door (5): [door frame, knob, handle, pane, door]
- table (4): [drawer, top, leg, apron]
- chair (7): [back, seat, leg, arm, stretcher, apron, seat cushion]
- car (9): [mirror, door, wheel, headlight, window, license plate, taillight, bumper, windshield]
- painting (1): [frame]
- sofa (7): [arm, seat cushion, seat base, leg, back pillow, skirt, back]
- shelf (1): [shelf]
- mirror (1): [frame]
- armchair (9): [back, arm, seat, seat cushion, seat base, earmuffs, leg, back pillow, apron]
- desk (1): [drawer]
- wardrobe (2): [door, drawer]
- lamp (9): [canopy, tube, shade, light source, column, base, highlight, arm, cord]
- bathtub (1): [faucet]
- chest of drawers (1): [drawer]
- sink (2): [faucet, tap]
- refrigerator (1): [door]
- pool table (3): [corner pocket, side pocket, leg]
- bookcase (1): [shelf]
- coffee table (2): [top, leg]
- toilet (3): [cistern, lid, bowl]
- stove (3): [stove, oven, button panel]
- computer (4): [monitor, keyboard, computer case, mouse]
- swivel chair (3): [back, seat, base]
- bus (1): [window]
- light (5): [shade, light source, highlight, aperture, diffusor]
- chandelier (4): [shade, light source, bulb, arm]
- airplane (1): [landing gear]
- van (1): [wheel]
- stool (1): [leg]
- microwave (1): [door]
- sconce (5): [shade, arm, light source, highlight, backplate]
- traffic light (1): [housing]
- fan (1): [blade]
- monitor (1): [screen]
- glass (4): [opening, bowl, base, stem]
- clock (1): [face]

The part-of-parts classes are listed as follows in the format of *part class name (the number of part-of-parts classes): [part-of-parts class names]*.

- upper sash (3): [pane, stile, muntin]
- lower sash (4): [rail, stile, pane, muntin]
- sash (4): [pane, rail, stile, muntin]
- drawer (2): [knob, handle]
- door (7): [hinge, knob, handle, pane, mirror, window, muntin]
- head (5): [eye, mouth, nose, ear, hair]
- back (3): [rail, spindle, stile]
- arm (4): [inside arm, outside arm, arm panel, arm support]
- wheel (1): [rim]
- window (3): [muntin, pane, shutter]

[2]The statistics are conducted based on the newest version of ADE20K from the official website.

[3]In our definition, only instance classes have part-level annotations.

Table 8. Semantic segmentation (Type-I) accuracy of ADE20K on Level-1 (*i.e.*, scene classes, *e.g.*, car) and Level-2 (*i.e.*, part-of-object classes, *e.g.*, wheel) classes, Level-3 (*i.e.*, part-of-parts classes, *e.g.*, rim).

mIoU (%)	**Lv-1**	**Lv-2**	**Lv-3**
SegFormer (B0) [39]	**37.85**	–	–
w/ CLIP & parts (ours)	36.38	43.08	51.56
SegFormer (B5) [39]	**49.13**	–	–
w/ CLIP & parts (ours)	48.52	55.13	63.40

Table 9. Instance segmentation results with non-probing-based and probing-based inference on the ADE20K dataset. Results are produced using the semantic mask prediction of SegFormer-B5. The hyper-parameter γ controls the centerness of the probes.

AP (%)	**Non-Probing**	**Probing (w.r.t. γ)**			
		0.0	0.2	0.5	1.0
CondInst (R50) [35]	24.6	–	–	–	–
w/ CLIP & mask samp.	**25.2**	**29.8**	**29.4**	**27.6**	**24.0**

Table 10. Overall segmentation results on ADE20K, using non-probing-based inference and probing-based inference respectively.

HPQ (%)	**Non-Probing**	**Probing (w.r.t. γ)**			
		0.0	0.2	0.5	1.0
SegFormer (B0) + CondInst (R50)	27.2	34.2	34.2	33.8	32.9
SegFormer (B5) + CondInst (R50)	33.9	39.1	38.9	38.5	37.9

- column (2): [shaft, capital]
- base (1): [wheel]
- stove (1): [burner]
- oven (1): [door]
- button panel (1): [dial]
- monitor (1): [screen]
- face (1): [hand]

C.2. More Quantitative Results

We report the individual segmentation accuracy of Type-I and Type-II requests on ADE20K, as shown in Tables 8 and 9, respectively. We additionally report the HPQ results on the ADE20K dataset with probing-based inference in Table 10 (while only non-probing-based results are reported in Table 4). We have by default used the instance segmentation results with CLIP and mask sampling strategy. Similar to the observations on CPP, although the non-probing-based setting achieves comparable instance AP, it reports lower HPQ values due to the penalty on false positives. Besides, all these values are significantly lower than the values reported on CPP, indicating that ADE20K is much more challenging in terms of the richness of semantics and the granularity of annotation. Overall, there is much room for improvement in such a complex, multi-level, and sparsely annotated segmentation dataset.

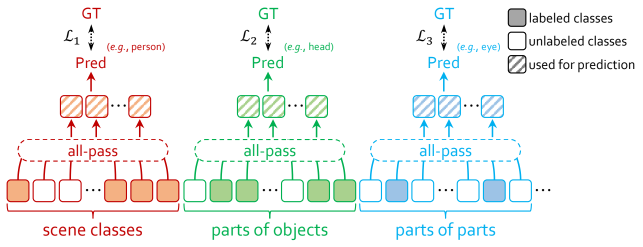

Figure 14. An illustration of how Type-I requests are processed for *one pixel* with the **conventional method**. Each slot indicates the predicted logit of one class. All classes are used during training. See Figure 3 for detailed differences.

C.3. Details of Competitor

We introduce the details of the competitor (*i.e*, conventional method) compared in Table 4. Since no prior works have ever reported quantitative results for part-aware segmentation on ADE20K, we provide a preliminary solution of adapting conventional methods to ADE20K (see Section 5.3). Figure 14 illustrates how the competitor method deals with Type-I requests, *i.e.*, joint training in a multi-task manner and simply ignoring all unlabeled pixels. There are several main differences compared to ViRReq: (i) using fixed class ID and learnable classifier for each class instead of language-based queries, (ii) predicting different levels of results simultaneously instead of iteratively, and (iii) how to handle unlabeled pixels. See Figure 3 for detailed differences.

C.4. More Qualitative Results

We visualize more segmentation results of ViRReq on ADE20K, as shown in Figure 15.

C.5. Details of Few-shot Incremental Learning

ViRReq has the ability of learning new visual concepts (objects and/or parts) from a few training samples. To show this ability, we manually select 50 scene-level and 19 part-level long-tailed classes of ADE20K and add them to the original knowledge base, as listed below. For each new concept, 20 instances are labeled for few-shot learning.

- **scene-level classes:** spotlight, wall socket, fluorescent, jar, shoe, candlestick, switch, air conditioner, telephone, mug, container, board, candle, cup, pitcher, deck chair, light bulb, coffee maker, teapot, partition, shrub, figurine, magazine, can, umbrella, bucket, napkin, text, gravestone, pane, patty, manhole, hat, doorframe, curb, loudspeaker, snow, pool ball, hedge, pipe,

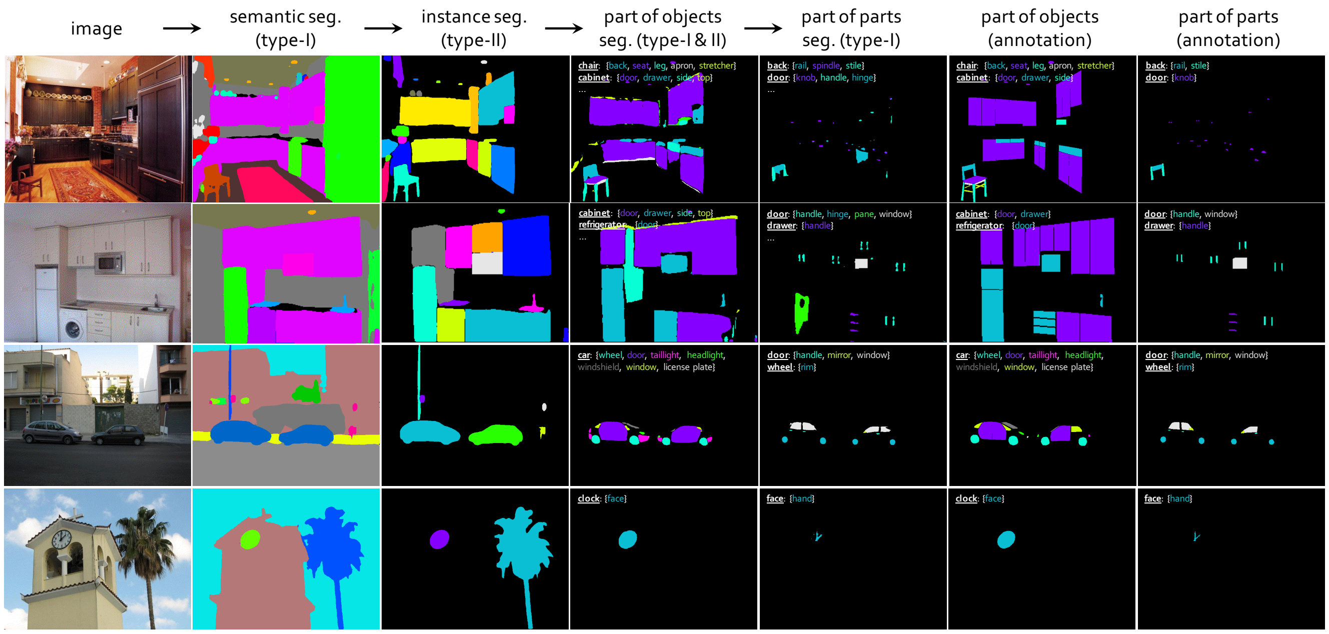
image
semantic seg. (type-I)
instance seg. (type-II)
part of objects seg. (type-I & II)
part of parts seg. (type-I)
part of objects (annotation)
part of parts (annotation)
chair: {back, seat, leg, apron, stretcher}
cabinet: {door, drawer, side, top}
back: {rail, spindle, stile}
door: {knob, handle, hinge}
chair: {back, seat, leg, apron, stretcher}
cabinet: {door, drawer, side}
back: {rail, stile}
door: {knob}
cabinet: {door, drawer, side, top}
refrigerator: {door}
door: {handle, hinge, pane, window}
drawer: {handle}
cabinet: {door, drawer}
refrigerator: {door}
door: {handle, window}
drawer: {handle}
car: {wheel, door, taillight, headlight, windshield, window, license plate}
door: {handle, mirror, window}
wheel: {rim}
car: {wheel, door, taillight, headlight, windshield, window, license plate}
door: {handle, mirror, window}
wheel: {rim}
clock: {face}
face: {hand}
clock: {face}
face: {hand}

Figure 15. More visualization results of ViRReq on ADE20K. Black areas in prediction indicate the others (*i.e.*, unknown) class. The corresponding sub-knowledge is listed in the blank area for reference. *Best viewed in color and by zooming in for details.*

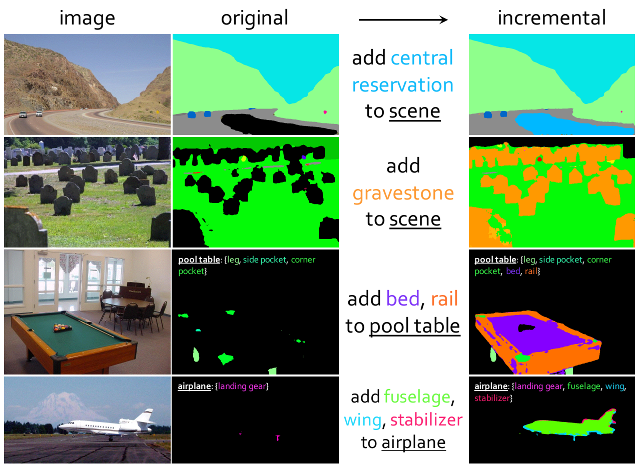
image
original
incremental
add central reservation to scene
add gravestone to scene
add bed, rail to pool table
add fuselage, wing, stabilizer to airplane
pool table: {leg, side pocket, corner pocket}
pool table: {leg, side pocket, corner pocket, bed, rail}
airplane: {landing gear}
airplane: {landing gear, fuselage, wing, stabilizer}

Figure 16. More qualitative results of few-shot incremental learning on objects (top two rows), and object parts (bottom two rows). Prior to incremental learning, these new concepts are recognized as others (the black areas) which is as expected.

central reservation, booklet, grill, place mat, faucet, notebook, document, fish, jacket, price tag

- **part-level classes:** bedpost, sill, casing, rail, stile, gas cap, chain, capital, shaft, bed, burner, dial, speaker, piston, wing, stabilizer, fuselage, turbine engine, motor

We noticed that existing few-shot semantic segmentation approaches [48–50, 52] usually focus on finding visual correspondences between query and support images for mask prediction (*e.g.*, the prototype-based methods), without updating the original models learned on base classes. However, we believe that it is more essential to update (*e.g.*, fine-tune) the model to learn knowledge of the new visual concepts. Inspired by the recent works on CLIP-based few-shot classification [10, 44], we mostly follow CLIP-Adapter [10] to integrate an additional bottleneck layer (*e.g.*, two 1×1 convolutional layers) for feature alignment. Only this bottleneck layer gets updated during the fine-tuning, *i.e.*, learning to project the original trained visual features to a new feature space.

Note that, in our setting, the existing training images may contain unlabeled pixels of these new classes (expect to recognize as others originally). To avoid possible conflict, we associate each training image to the knowledge base that was used for annotation (which we call data versioning, see Section 3.2). For example, one image may have multiple copies in the training set that associated with different version of knowledge base (*e.g.*, annotating for the newly added concepts or not).

Specifically, we mix the original and new training images and fine-tune the trained SegFormer-B5 model for 40K iterations (1/4 of base training iterations). Since the number of new training images (*e.g.*, $20 \times 50 = 1000$ images for scene-level incremental learning) is significantly lower than the original (*e.g.*, 20,210 training images), these new images are repeated multiple times during training to alleviate the imbalance problem. Other configurations are the same as the base training (see Appendix A.2). We show more qualitative results in Figure 16.

Quantitatively, the mIoUs of the newly added 50 scene-level and 19 part-level classes are 7.0% and 16.6%, respectively. The situations vary across classes – some results (*e.g.*, 34.8% for jacket) are reasonable, but others (*e.g.*, 0.5% for hat) are unstable because ADE20K contains very few yet style-diversified test cases.

C.6. Limitations

One of the limitations is error propagation, which occurs between adjacent levels of segmentation, for example, inaccurate scene-level semantic segmentation results may affect the subsequent instance segmentation since we expect that the predicted instance mask should strictly be inside the predicted semantic region. For example, some chairs in Figure 4 are recognized incompletely, so the corresponding parts (*e.g.*, legs) are missing. We empirically found that, by replacing the semantic prediction with the corresponding ground-truth (*i.e.*, eliminating error propagation) on CPP, the instance segmentation results could be improved by $2 \sim 3\%$ AP. Note that the performance gap is moderate because the semantic results on CPP are usually superior ($75 \sim 85\%$ mIoU), and the performance gap would be more significant on ADE20K. Additionally, we believe that applying advanced mask fusion methods [21,24,30,32] could mitigate the error accumulation effects. Besides error propagation, the limitations of ViRReq also include using separate models for two request types, the manual construction of knowledge base, *etc.*, which we leave for future work.

References

[46] Hermann Blum, Paul-Edouard Sarlin, Juan Nieto, Roland Siegwart, and Cesar Cadena. The fishyscapes benchmark: Measuring blind spots in semantic segmentation. *International Journal of Computer Vision*, 129(11):3119–3135, 2021. 14, 12

[47] MMSegmentation Contributors. MMSegmentation: Openmmlab semantic segmentation toolbox and benchmark. https://github.com/open-mmlab/mmsegmentation, 2020. 11, 9

[48] Nanqing Dong and Eric P. Xing. Few-shot semantic segmentation with prototype learning. In *British Machine Vision Conference*, 2018. 18, 16

[49] Yongfei Liu, Xiangyi Zhang, Songyang Zhang, and Xuming He. Part-aware prototype network for few-shot semantic segmentation. *arXiv preprint arXiv:2007.06309*, 2020. 18, 16

[50] Juhong Min, Dahyun Kang, and Minsu Cho. Hypercorrelation squeeze for few-shot segmenation. In *International Conference on Computer Vision*, 2021. 18, 16

[51] Zhi Tian, Hao Chen, Xinlong Wang, Yuliang Liu, and Chunhua Shen. AdelaiDet: A toolbox for instance-level recognition tasks. https://github.com/aim-uofa/AdelaiDet, 2019. 11, 9

[52] Kaixin Wang, Jun Hao Liew, Yingtian Zou, Daquan Zhou, and Jiashi Feng. Panet: Few-shot image semantic segmentation with prototype alignment. In *International Conference on Computer Vision*, 2019. 18, 16

[53] Xinlong Wang, Rufeng Zhang, Tao Kong, Lei Li, and Chunhua Shen. Solov2: Dynamic and fast instance segmentation. In *Advances in Neural Information Processing Systems*, 2020. 11, 9

[54] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019. 12, 10