YOLACT
Real-time Instance Segmentation

Daniel Bolya Chong Zhou Fanyi Xiao Yong Jae Lee

University of California, Davis

{dbolya, cczhou, fyxiao, yongjaelee}@ucdavis.edu

Organized by: Xi Fang

Date: 08/05/2020

Motivation

- State-of-the-art approaches to instance segmentation directly build off of advances in object detection like Faster R-CNN and R-FCN.
- These methods focus primarily on performance over speed, leaving the scene devoid of instance segmentation parallels to real-time object detectors like SSD and YOLO.
- To fill that gap with a fast, one-stage instance segmentation model in the same way that SSD and YOLO fill that gap for object detection

Contribution

- They proposes first real-time (> 30 fps) instance segmentation algorithm with competitive results on the challenging MS COCO dataset.
- They analyzes the emergent behavior of YOLACT's prototypes and provide experiments to study the speed vs. performance trade-offs.
- They provide a novel Fast NMS approach that is 12ms faster than traditional NMS with a negligible performance penalty.

Related Works

- Instance Segmentation
  - Mask-RCNN is a representative two-stage instance segmentation approach that first generates candidate region-of-interests (ROIs) and then classifies and segments those ROIs in the second stage.
  - These two-stage methods require re-pooling features for each ROI and processing them with subsequent computations.
- Real-time Instance Segmentation
  - Straight to Shapes and Box2Pix can perform instance segmentation in real-time, but their accuracies are far from that of modern baselines.
- Prototypes
  - Learning prototypes (aka vocabulary or codebook) has been extensively explored in computer vision.
  - They learn prototypes that are specific to each image, rather than global prototypes shared across the entire dataset.

Approach

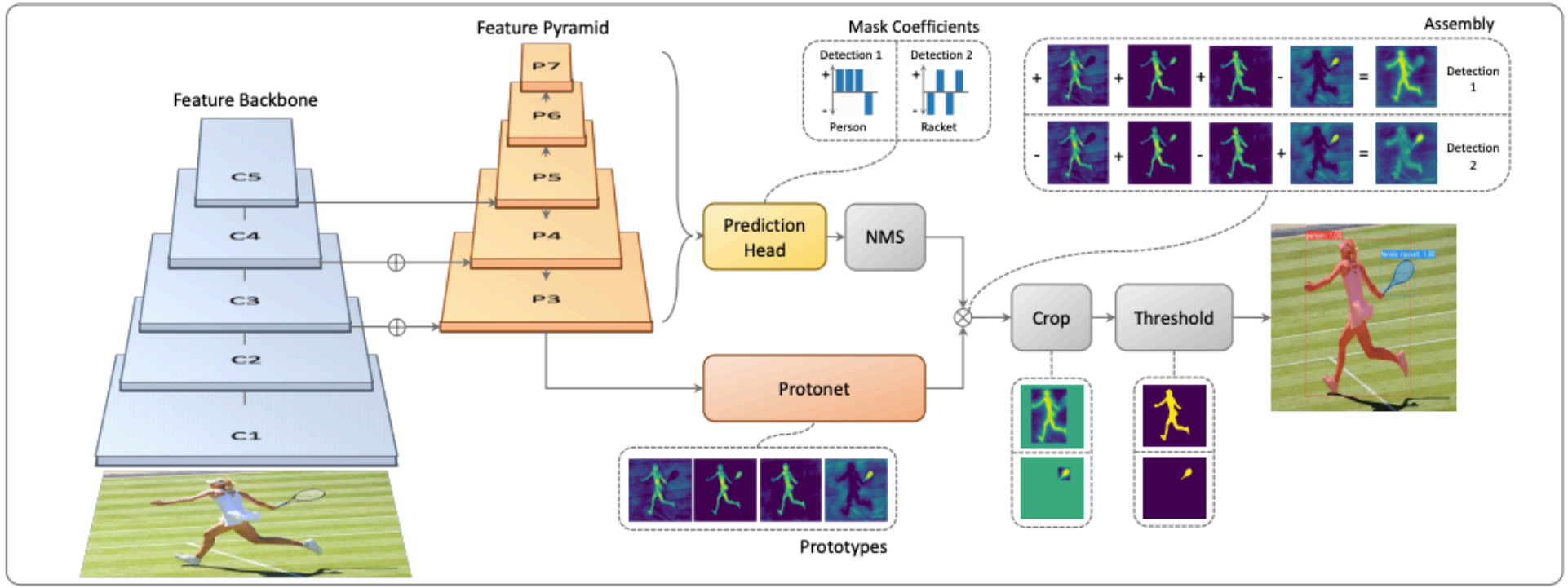

Figure 2: **YOLACT Architecture** Blue/yellow indicates low/high values in the prototypes, gray nodes indicate functions that are not trained, and $k = 4$ in this example. We base this architecture off of RetinaNet [27] using ResNet-101 + FPN.

Approach

- They break up the complex task of instance segmentation into two simpler, parallel tasks that can be assembled to form the final masks
- The first branch uses an FCN to produce a set of image-sized "prototype masks" that do not depend on any one instance.
- The second adds an extra head to the object detection branch to predict a vector of "mask coefficients" for each anchor in the prototype space.
- Finally, for each instance that survives NMS, they construct a mask for that instance by linearly combining the work of these two branches.

Prototype Generation

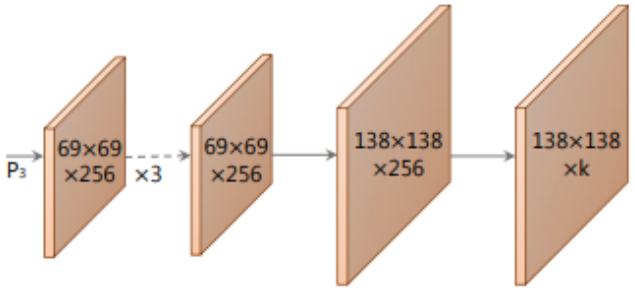

Figure 3: **Protonet Architecture** The labels denote feature size and channels for an image size of 550×550. Arrows indicate 3×3 *conv* layers, except for the final *conv* which is 1×1. The increase in size is an upsample followed by a *conv*. Inspired by the mask branch in [18].

- The prototype generation branch (protonet) predicts a set of k prototype masks for the entire image.
- This formulation is similar to standard semantic segmentation, but no explicit loss on the prototypes.
- Higher resolution prototypes result in both higher quality masks and better performance on smaller objects. They use FPN because its largest feature layers.

Mask Coefficients

- Typical anchor-based object detectors have two branches, one branch to predict c class, and the other to predict bounding box
- For mask coefficient prediction, we simply add a third branch in parallel that predicts k mask coefficients, one corresponding to each prototype
- They apply tanh to the k mask coefficients, which produces more stable outputs over no nonlinearity

Mask Assembly

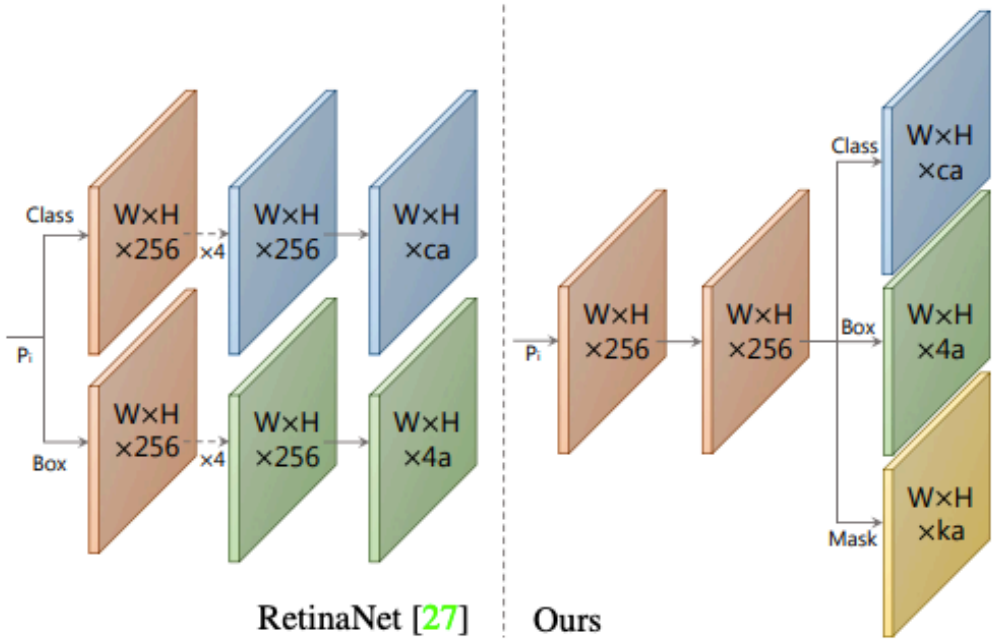

Figure 4: **Head Architecture** We use a shallower prediction head than RetinaNet [27] and add a mask coefficient branch. This is for c classes, a anchors for feature layer P_i, and k prototypes. See Figure 3 for a key.

To produce instance masks, we combine the work of the prototype branch and mask coefficient branch, using a linear combination of the former with the latter as coefficients. We then follow this by a sigmoid nonlinearity to produce the final masks. These operations can be implemented efficiently using a single matrix multiplication and sigmoid:

$$M = \sigma(PC^T) \tag{1}$$

where P is an $h \times w \times k$ matrix of prototype masks and C is a $n \times k$ matrix of mask coefficients for n instances surviving NMS and score thresholding. Other, more complicated combination steps are possible; however, we keep it simple (and fast) with a basic linear combination.

Mask Assembly

Losses We use three losses to train our model: classification loss L_{cls}, box regression loss L_{box} and mask loss L_{mask} with the weights 1, 1.5, and 6.125 respectively. Both L_{cls} and L_{box} are defined in the same way as in [30]. Then to compute mask loss, we simply take the pixel-wise binary cross entropy between assembled masks M and the ground truth masks M_{gt}: $L_{mask} = \text{BCE}(M, M_{gt})$.

Cropping Masks We crop the final masks with the predicted bounding box during evaluation. During training, we instead crop with the ground truth bounding box, and divide L_{mask} by the ground truth bounding box area to preserve small objects in the prototypes.

Emergent Behavior

- The approach might seem surprising, as the general consensus around instance segmentation is that because FCNs are translation invariant.
- Methods like FCIS and Mask R-CNN try to explicitly add translation variance.
- YOLACT learns how to localize instances on its own via different activations in its prototypes.
- The network has to play a balancing act to produce the right coefficients, and adding more prototypes makes this harder

Emergent Behavior

- The mask coefficient branch can learn which situations call for which functionality. For instance, in Figure 5, prototype 2 is a partitioning prototype but also fires most strongly on instances in the bottom-left corner. Prototype 3 is similar but for instances on the right.

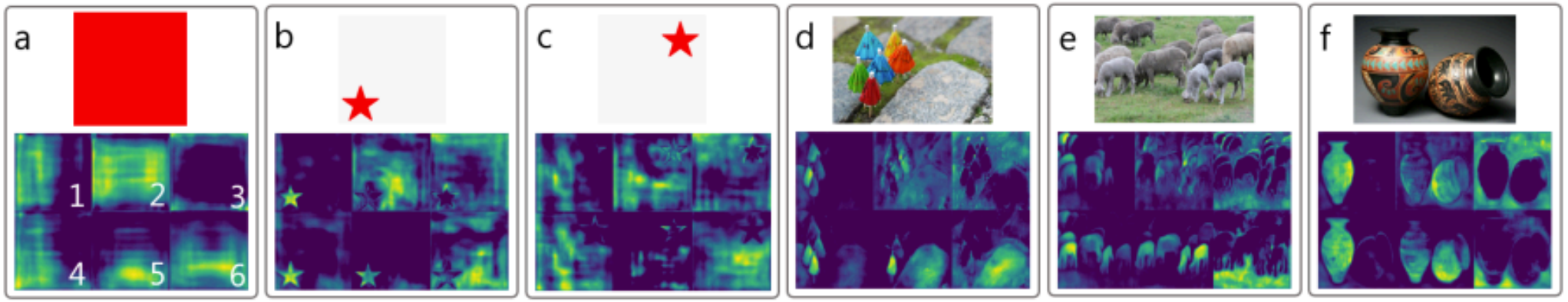

Figure 5: **Prototype Behavior** The activations of the same six prototypes across different images. Prototypes 1, 4, and 5 are partition maps with boundaries clearly defined in image a, prototype 2 is a bottom-left directional map, prototype 3 segments out the background and provides instance contours, and prototype 6 segments out the ground.

Fast NMS

To perform Fast NMS, we first compute a $c \times n \times n$ pairwise IoU matrix X for the top n detections sorted descending by score for each of c classes. Batched sorting on the GPU is readily available and computing IoU can be easily vectorized. Then, we remove detections if there are any higher-scoring detections with a corresponding IoU greater than some threshold t. We efficiently implement this by first setting the lower triangle and diagonal of X to 0: $X_{kij} = 0,\ \forall k, j, i \geq j$, which can be performed in one batched `triu` call, and then taking the column-wise max:

$$K_{kj} = \max_i(X_{kij}) \qquad \forall k, j \tag{2}$$

to compute a matrix K of maximum IoU values for each detection. Finally, thresholding this matrix with t ($K < t$) will indicate which detections to keep for each class.

Because of the relaxation, Fast NMS has the effect of removing slightly too many boxes. However, the performance hit caused by this is negligible compared to the stark increase in speed (see Table 2a). In our code base, Fast NMS is 11.8 ms faster than a Cython implementation of traditional NMS while only reducing performance by 0.1 mAP. In the Mask R-CNN benchmark suite [18], Fast NMS is 15.0 ms faster than their CUDA implementation of traditional NMS with a performance loss of only 0.3 mAP.

Experiements

We report instance segmentation results on MS COCO [28] and Pascal 2012 SBD [16] using the standard metrics. For MS COCO, we train on `train2017` and evaluate on `val2017` and `test-dev`.

Implementation Details We train all models with batch size 8 *on one GPU* using ImageNet [10] pretrained weights. We find that this is a sufficient batch size to use batch norm, so we leave the pretrained batch norm unfrozen but do not add any extra *bn* layers. We train with SGD for 800k iterations starting at an initial learning rate of 10^{-3} and divide by 10 at iterations 280k, 600k, 700k, and 750k, using a weight decay of 5×10^{-4}, a momentum of 0.9, and all data augmentations used in SSD [30]. For Pascal, we train for 120k iterations and divide the learning rate at 60k and 100k. We also multiply the anchor scales by $4/3$, as objects tend to be larger. Training takes 4-6 days (depending on config) on one Titan Xp for COCO and less than 1 day on Pascal.

Results

Figure 6: **YOLACT** evaluation results on COCO's `test-dev` set. This base model achieves 29.8 mAP at 33.0 fps. All images have the confidence threshold set to 0.3.

Method	NMS	AP	FPS	Time
YOLACT	Standard	**30.0**	23.8	42.1
	Fast	29.9	**33.0**	**30.3**
Mask R-CNN	Standard	**36.1**	8.6	116.0
	Fast	35.8	**9.9**	**101.0**

(a) **Fast NMS** Fast NMS performs only slightly worse than standard NMS, while being around 12 ms faster. We also observe a similar trade-off implementing Fast NMS in Mask R-CNN.

k	AP	FPS	Time
32	27.7	**32.4**	**30.9**
64	**27.8**	31.7	31.5
128	27.6	31.5	31.8
256	27.7	29.8	33.6

(b) **Prototypes** Choices for k in our method. YOLACT is robust to varying k, so we choose the fastest ($k = 32$).

Method	AP	FPS	Time
FCIS w/o Mask Voting	27.8	9.5	105.3
Mask R-CNN (550 × 550)	**32.2**	13.5	73.9
fc-mask	20.7	25.7	38.9
YOLACT-550 (Ours)	29.9	**33.0**	**30.3**

(c) **Accelerated Baselines** We compare to other baseline methods by tuning their speed-accuracy trade-offs. *fc*-mask is our model but with 16 × 16 masks produced from an *fc* layer.

Table 2: **Ablations** All models evaluated on COCO `val2017` using our servers. Models in Table 2b were trained for 400k iterations instead of 800k. Time in milliseconds reported for convenience.

Results

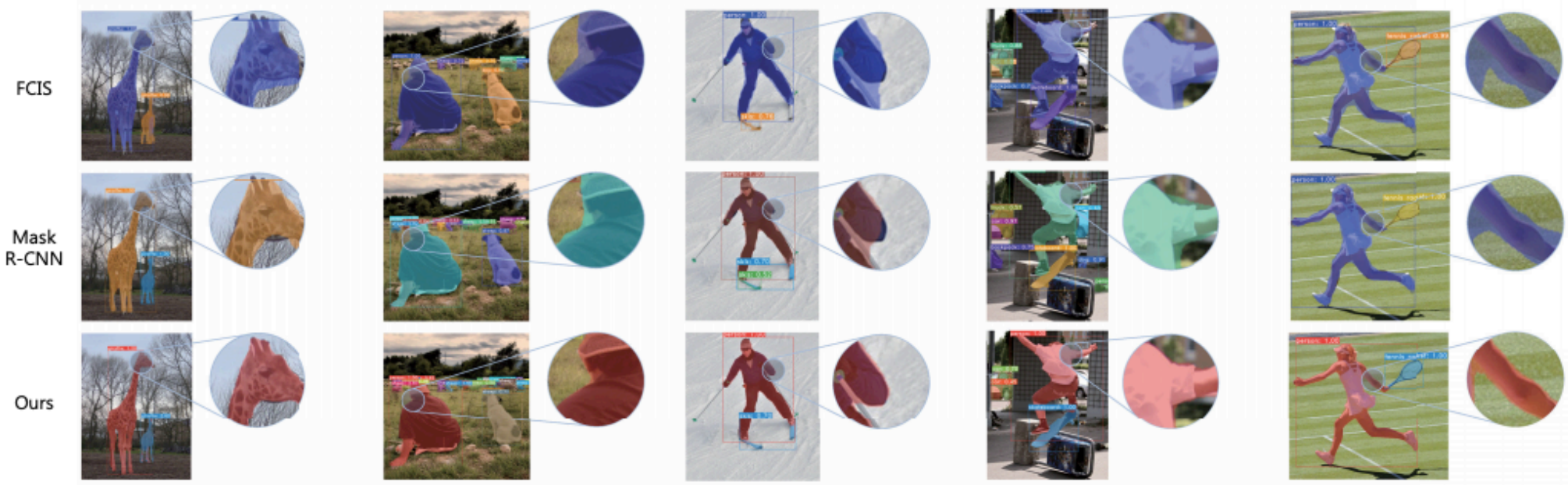

Figure 7: **Mask Quality** Our masks are typically higher quality than those of Mask R-CNN [18] and FCIS [24] because of the larger mask size and lack of feature repooling.

Results

Method	Backbone	FPS	Time	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
PA-Net [29]	R-50-FPN	4.7	212.8	36.6	58.0	39.3	16.3	38.1	53.1
RetinaMask [14]	R-101-FPN	6.0	166.7	34.7	55.4	36.9	14.3	36.7	50.5
FCIS [24]	R-101-C5	6.6	151.5	29.5	51.5	30.2	8.0	31.0	49.7
Mask R-CNN [18]	R-101-FPN	8.6	116.3	35.7	58.0	37.8	15.5	38.1	52.4
MS R-CNN [20]	R-101-FPN	8.6	116.3	**38.3**	58.8	41.5	17.8	40.4	54.4
YOLACT-550	R-101-FPN	**33.5**	**29.8**	29.8	48.5	31.2	9.9	31.3	47.7
YOLACT-400	R-101-FPN	45.3	22.1	24.9	42.0	25.4	5.0	25.3	45.0
YOLACT-550	R-50-FPN	45.0	22.2	28.2	46.6	29.2	9.2	29.3	44.8
YOLACT-550	D-53-FPN	40.7	24.6	28.7	46.8	30.0	9.5	29.6	45.5
YOLACT-700	R-101-FPN	23.4	42.7	31.2	50.6	32.8	12.1	33.3	47.1

Table 1: **MS COCO [28] Results** We compare to state-of-the-art methods for mask mAP and speed on COCO `test-dev` and include several ablations of our base model, varying backbone network and image size. We denote the backbone architecture with `network-depth-features`, where `R` and `D` refer to ResNet [19] and DarkNet [36], respectively. Our base model, YOLACT-550 with ResNet-101, is 3.9x faster than the previous fastest approach with competitive mask mAP.

Conclusion

Localization Failure If there are too many objects in one spot in a scene, the network can fail to localize each object in its own prototype. In these cases, the network will output something closer to a foreground mask than an instance segmentation for some objects in the group; e.g., in the first image in Figure 6 (row 1 column 1), the blue truck under the red airplane is not properly localized.

Leakage Our network leverages the fact that masks are cropped after assembly, and makes no attempt to suppress noise outside of the cropped region. This works fine when the bounding box is accurate, but when it is not, that noise can creep into the instance mask, creating some "leakage"

Understanding the AP Gap However, localization failure and leakage alone are not enough to explain the almost 6 mAP gap between YOLACT's base model and, say, Mask R-CNN. Indeed, our base model on COCO has just a 2.5 mAP difference between its test-dev mask and box mAP (29.8 mask, 32.3 box), meaning our base model would only gain a few points of mAP even with perfect masks. Moreover, Mask R-CNN has this same mAP difference (35.7 mask, 38.2 box), which suggests that the gap between the two methods lies in the relatively poor performance of our detector and not in our approach to generating masks.