# YOLACT++ Better Real-time Instance Segmentation

Daniel Bolya*, Chong Zhou*, Fanyi Xiao, and Yong Jae Lee

**Abstract**—We present a simple, fully-convolutional model for real-time ($> 30$ fps) instance segmentation that achieves competitive results on MS COCO evaluated on a single Titan Xp, which is significantly faster than any previous state-of-the-art approach. Moreover, we obtain this result after training on **only one GPU**. We accomplish this by breaking instance segmentation into two parallel subtasks: (1) generating a set of prototype masks and (2) predicting per-instance mask coefficients. Then we produce instance masks by linearly combining the prototypes with the mask coefficients. We find that because this process doesn't depend on repooling, this approach produces very high-quality masks and exhibits temporal stability for free. Furthermore, we analyze the emergent behavior of our prototypes and show they learn to localize instances on their own in a translation variant manner, despite being fully-convolutional. We also propose Fast NMS, a drop-in 12 ms faster replacement for standard NMS that only has a marginal performance penalty. Finally, by incorporating deformable convolutions into the backbone network, optimizing the prediction head with better anchor scales and aspect ratios, and adding a novel fast mask re-scoring branch, our YOLACT++ model can achieve $34.1$ mAP on MS COCO at $33.5$ fps, which is fairly close to the state-of-the-art approaches while still running at real-time.

**Index Terms**—Instance Segmentation, Real Time

## 1 Introduction

*"Boxes are stupid anyway though, I'm probably a true believer in masks except I can't get YOLO to learn them."*

– Joseph Redmon, YOLOv3 [1]

What would it take to create a real-time instance segmentation algorithm? Over the past few years, the vision community has made great strides in instance segmentation, in part by drawing on powerful parallels from the well-established domain of object detection. State-of-the-art approaches to instance segmentation like Mask R-CNN [2] and FCIS [3] directly build off of advances in object detection like Faster R-CNN [4] and R-FCN [5]. Yet, these methods focus primarily on performance over speed, leaving the scene devoid of instance segmentation parallels to real-time object detectors like SSD [6] and YOLO [1], [7]. In this work, our goal is 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.

However, instance segmentation is hard—much harder than object detection. One-stage object detectors like SSD and YOLO are able to speed up existing two-stage detectors like Faster R-CNN by simply removing the second stage and making up for the lost performance in other ways (e.g., strong data augmentation, anchor clustering, etc.). The same approach is not easily extendable, however, to instance segmentation. State-of-the-art two-stage instance segmentation methods depend heavily on *feature localization* to produce masks. That is, these methods "re-pool" features in some bounding box region (e.g., via RoI-pool/align), and then feed these now localized features to their mask predictor. This approach is inherently sequential and is therefore difficult to accelerate. One-stage methods that perform these steps in parallel like FCIS do exist (e.g., [3], [8], [9]), but they require significant amounts of post-processing after localization, and thus are still far from real-time.

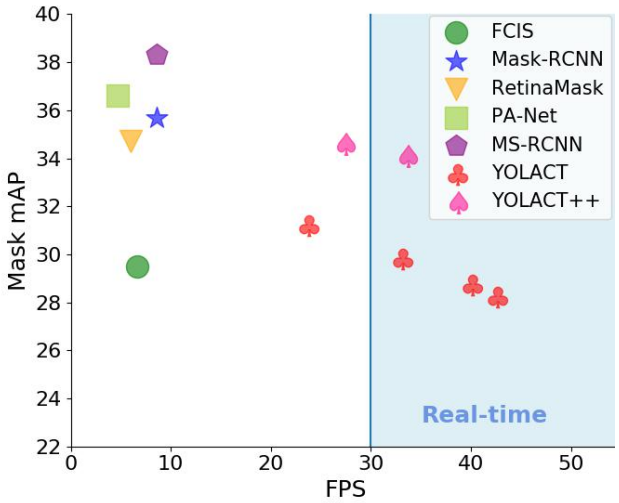



Fig. 1: Speed-performance trade-off for various instance segmentation methods on COCO. To our knowledge, ours is the first *real-time* (above 30 FPS) approach with over 30 mask mAP on COCO `test-dev`.

To address these issues, we propose YOLACT[1], a real-time instance segmentation framework that forgoes an explicit localization step. Instead, YOLACT breaks up instance segmentation into two parallel tasks: (1) generating a dictionary of non-local *prototype masks over the entire image*, and (2) predicting a set of *linear combination coefficients per instance*. Then producing a full-image instance segmentation from these two components is

* *The first two authors contributed equally to this work.*
* *This work was performed at the Computer Vision Lab, Department of Computer Science, University of California, Davis, CA 95616 USA. E-mail: {dbolya, cczhou, fyxiao, yongjaelee}@ucdavis.edu.*

1. **Y**ou **O**nly **L**ook **A**t **C**oefficien**T**s

simple: for each instance, linearly combine the prototypes using the corresponding predicted coefficients and then crop with a predicted bounding box. We show that by segmenting in this manner, *the network learns how to localize instance masks on its own*, where visually, spatially, and semantically similar instances appear different in the prototypes.

Moreover, since the number of prototype masks is independent of the number of categories (e.g., there can be more categories than prototypes), YOLACT learns a distributed representation in which each instance is segmented with a combination of prototypes that are shared across categories. This distributed representation leads to interesting emergent behavior in the prototype space: some prototypes spatially partition the image, some localize instances, some detect instance contours, some encode position-sensitive directional maps (similar to those obtained by hard-coding a position-sensitive module in FCIS [3]), and most do a combination of these tasks (see Figure 5).

This approach also has several practical advantages. First and foremost, it's fast: because of its parallel structure and extremely lightweight assembly process, YOLACT adds only a marginal amount of computational overhead to a one-stage backbone detector, making it easy to reach 30 fps even when using ResNet-101 [10]; in fact, *the entire mask branch takes only ∼5 ms to evaluate*. Second, masks are high-quality: since the masks use the full extent of the image space without any loss of quality from repooling, our masks for large objects are significantly higher quality than those of other methods (see Figure 8). Finally, it's general: the idea of generating prototypes and mask coefficients could be added to almost any modern object detector.

Our main contribution is the first real-time ($>$ 30 fps) instance segmentation algorithm with competitive results on the challenging MS COCO dataset [11] (see Figure 1). In addition, we analyze the emergent behavior of YOLACT's prototypes and provide experiments to study the speed vs. performance trade-offs obtained with different backbone architectures, numbers of prototypes, and image resolutions. We also provide a novel Fast NMS approach that is 12ms faster than traditional NMS with a negligible performance penalty. To further improve the performance of our model over our conference paper version [12], in Section 6, we propose YOLACT++. Specifically, we incorporate deformable convolutions [13], [14] into the backbone network, which provide more flexible feature sampling and strengthening its capability of handling instances with different scales, aspect ratios, and rotations. Furthermore, we optimize the prediction heads with better anchor scale and aspect ratio choices for larger object recall. Finally, we also introduce a novel fast mask re-scoring branch, which results in a decent performance boost with only marginal speed overhead. These improvements are validated in Tables 3, 6, and 7. Apart from these algorithm improvements over our conference paper [12], we also provide more qualitative results (Figure 9), a timing breakdown of each stage (Table 8), and real-time bounding box detection results (Table 5).

The code for YOLACT and YOLACT++ are both available at https://github.com/dbolya/yolact.

## 2 Related Work

**Instance Segmentation** Given its importance, a lot of research effort has been made to push instance segmentation *accuracy*. Mask-RCNN [2] 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. Follow-up works try to improve its accuracy by e.g., enriching the FPN features [15] or addressing the incompatibility between a mask's confidence score and its localization accuracy [16]. These two-stage methods require re-pooling features for each ROI and processing them with subsequent computations, which make them unable to obtain real-time speeds (30 fps) even when decreasing image size (see Table 2c).

One-stage instance segmentation methods generate position sensitive maps that are assembled into final masks with position-sensitive pooling [3], [17] or combine semantic segmentation logits and direction prediction logits [18]. Though conceptually faster than two-stage methods, they still require repooling or other non-trivial computations (e.g., mask voting). This severely limits their speed, placing them far from real-time. In contrast, our assembly step is much more lightweight (only a linear combination) and can be implemented as one GPU-accelerated matrix-matrix multiplication, making our approach very fast.

Finally, some methods first perform semantic segmentation followed by boundary detection [19], pixel clustering [20], [21], CRF inference [22], or learn an embedding to form instance masks [23], [24], [25], [26]. Again, these methods have multiple stages and/or involve expensive clustering procedures, which limits their viability for real-time applications.

**Real-time Instance Segmentation** While real-time object detection [1], [6], [7], [28], and semantic segmentation [29], [30], [31], [32], [33] methods exist, few works have focused on real-time instance segmentation. Straight to Shapes [34] and Box2Pix [35] can perform instance segmentation in real-time (30 fps on Pascal SBD 2012 [36], [37] for Straight to Shapes, and 10.9 fps on Cityscapes [38] and 35 fps on KITTI [39] for Box2Pix), but their accuracies are far from that of modern baselines. While [40] substantially improves instance segmentation accuracy over these prior methods, it runs only at 11 fps on Cityscapes. In fact, Mask R-CNN [2] remains one of the fastest instance segmentation methods on semantically challenging datasets like COCO [11] (13.5 fps on $550^2$ px images; see Table 2c).

**Prototypes** Learning prototypes (aka vocabulary/codebook) has been extensively explored in computer vision. Classical representations include textons [41] and visual words [42], with advances made via sparsity and locality priors [43], [44], [45]. Others have designed prototypes for object detection [**?**], [46], [47]. Though related, these works use prototypes to represent features, whereas we use them to assemble masks for instance segmentation. Moreover, we learn prototypes that are specific to each image, rather than global prototypes shared across the entire dataset like in [22]. Also, unlike the "shape priors" defined in [22], which are fixed shape primitives, our prototypes are per-image feature maps that all masks can draw from.

## 3 YOLACT

Our goal is to add a mask branch to an existing one-stage object detection model in the same vein as Mask R-CNN [2] does to Faster R-CNN [4], but without an explicit feature localization step (e.g., feature repooling). To do this, we 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 [48] to produce a set of image-sized "prototype masks" that do not depend on any one instance. The second adds an

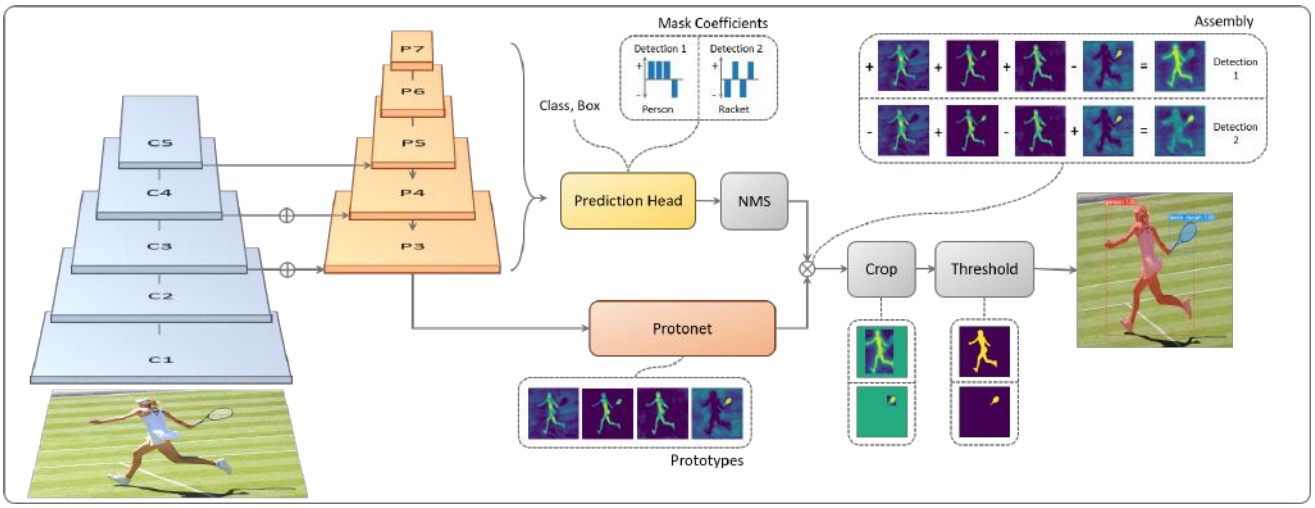



Fig. 2: **YOLACT Architecture** Blue/yellow indicates low/high values in the prototypes, gray nodes indicate functions that are not trained, and $k = 4$ in this example. We base this architecture off of RetinaNet [27] using ResNet-101 + FPN.

extra head to the object detection branch to predict a vector of "mask coefficients" for each anchor that encode an instance's representation in the prototype space. Finally, for each instance that survives box-based NMS, we construct a mask for that instance by linearly combining the work of these two branches.

**Rationale** We perform instance segmentation in this way primarily because masks are spatially coherent; i.e., pixels close to each other are likely to be part of the same instance. While a convolutional (*conv*) layer naturally takes advantage of this coherence, a fully-connected (*fc*) layer does not. That poses a problem, since one-stage object detectors produce class and box coefficients for each anchor as an output of an *fc* layer.[2] Two stage approaches like Mask R-CNN get around this problem by using a localization step (e.g., RoI-Align), which preserves the spatial coherence of the features while also allowing the mask to be a *conv* layer output. However, doing so requires a significant portion of the model to wait for a first-stage RPN to propose localization candidates, inducing a significant speed penalty.

Thus, we break the problem into two parallel parts, making use of *fc* layers, which are good at producing semantic vectors, and *conv* layers, which are good at producing spatially coherent masks, to produce the "mask coefficients" and "prototype masks", respectively. Then, because prototypes and mask coefficients can be computed independently, the computational overhead over that of the backbone detector comes mostly from the assembly step, which can be implemented as a single matrix multiplication. In this way, we can maintain spatial coherence in the feature space while still being one-stage and *fast*.

### 3.1 Prototype Generation

The prototype generation branch (protonet) predicts a set of $k$ prototype masks for the entire image. We implement protonet as an FCN whose last layer has $k$ channels (one for each prototype) and attach it to a backbone feature layer (see Figure 3 for an illustration). While this formulation is similar to standard semantic segmentation, it differs in that we exhibit no explicit loss on the prototypes. Instead, all supervision for these prototypes comes from the final mask loss after assembly.

We note two important design choices: taking protonet from deeper backbone features produces more robust masks, and higher resolution prototypes result in both higher quality masks and better performance on smaller objects. Thus, we use FPN [49] because its largest feature layers ($P_3$ in our case; see Figure 2) are the deepest. Then, we upsample it to one fourth the dimensions of the input image to increase performance on small objects.

Finally, we find it important for the protonet's output to be unbounded, as this allows the network to produce large, overpowering activations for prototypes it is very confident about (e.g., obvious background). Thus, we have the option of following protonet with either a `ReLU` or no nonlinearity. We choose `ReLU` for more interpretable prototypes.

### 3.2 Mask Coefficients

Typical anchor-based object detectors have two branches in their prediction heads: one branch to predict $c$ class confidences, and the other to predict $4$ bounding box regressors. For mask coefficient prediction, we simply add a third branch in parallel that predicts $k$ mask coefficients, one corresponding to each prototype. Thus, instead of producing $4 + c$ coefficients per anchor, we produce $4 + c + k$.

Then for nonlinearity, we find it important to be able to subtract out prototypes from the final mask. Thus, we apply `tanh` to the $k$ mask coefficients, which produces more stable outputs over no nonlinearity. The relevance of this design choice is apparent in Figure 2, as neither mask would be constructable without allowing for subtraction.

### 3.3 Mask Assembly

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

2. To show that this is an issue, we develop an "*fc*-mask" model that produces masks for each anchor as the reshaped output of an *fc* layer. As our experiments in Table 2c show, simply adding masks to a one-stage model as *fc* outputs only obtains 20.7 mAP and is thus very much insufficient.

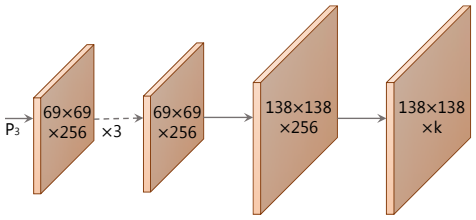



Fig. 3: **Protonet Architecture** The labels denote feature size and channels for an image size of $550 \times 550$. Arrows indicate $3 \times 3$ *conv* layers, except for the final *conv* which is $1 \times 1$. The increase in size is an upsample followed by a *conv*. Inspired by the mask branch in [2].

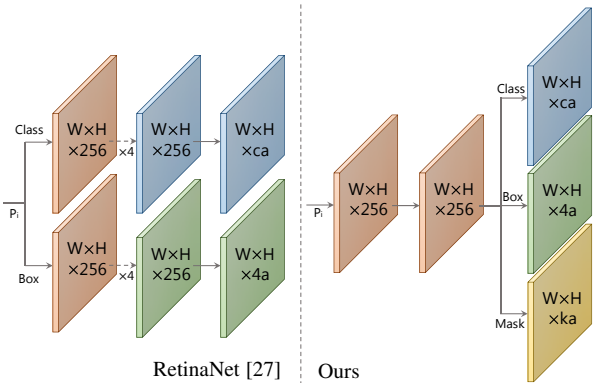



Fig. 4: **Head Architecture** We use a shallower prediction head than RetinaNet [27] and add a mask coefficient branch. This is for $c$ classes, $a$ anchors for feature layer $P_i$, and $k$ prototypes. See Figure 3 for a key.

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.

**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 [6]. 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. Specifically, we assign zero to pixels outside of the box region. During training, we instead crop with the ground truth bounding box, and divide $L_{mask}$ by the ground truth box area to preserve small objects in the prototypes.

### 3.4 Emergent Behavior

Our approach might seem surprising, as the general consensus around instance segmentation is that because FCNs are equivariant with respect to input translations, the task needs position awareness added back in [3]. Thus methods like FCIS [3] and Mask R-CNN [2] try to explicitly add position awareness, whether it be by directional maps and position-sensitive repooling, or by putting the mask branch in the second stage so it does not have to deal with localizing instances. In our method, the only position awareness we add is to crop the final mask with the predicted bounding box. However, we find that our method also works without cropping for medium and large objects, so this is not a result of cropping. Instead, YOLACT *learns how to localize instances on its own* via different activations in its prototypes.

To see how this is possible, first note that the prototype activations for the solid red image (image a) in Figure 5 are actually not possible in an FCN without padding. Because a convolution outputs to a single pixel, if its input everywhere in the image is the same, the result everywhere in the *conv* output will be the same. On the other hand, the consistent rim of padding in modern FCNs like ResNet gives the network the ability to tell how far away from the image's edge a pixel is. Conceptually, one way it could accomplish this is to have multiple layers in sequence spread the padded 0's out from the edge toward the center (e.g., with a kernel like $[1, 0]$). In practice, this leads to massive amount of effective padding for standard ConvNets like ResNets. For instance, ResNet 101 and ResNet 50 have 511px and 239px of padding in each direction respectively [50]. In addition to the large receptive field (e.g., 1027 pixels for ResNet 101), this means ResNet, *is inherently translation variant*, and our method makes heavy use of that property (images b and c exhibit clear translation variance).

We observe many prototypes to activate on certain "partitions" of the image. That is, they only activate on objects on one side of an implicitly learned boundary. In Figure 5, prototypes 1-3 are such examples. By combining these partition maps, the network can distinguish between different (even overlapping) instances of the same semantic class; e.g., in image d, the green umbrella can be separated from the red one by subtracting prototype 3 from prototype 2.

Furthermore, being learned objects, prototypes are compressible. That is, if protonet combines the functionality of multiple prototypes into one, 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. This explains why in practice, the model does not degrade in performance even with as low as $k = 32$ prototypes (see Table 2b).

On the other hand, increasing $k$ is ineffective most likely because predicting coefficients is difficult. If the network makes a large error in even one coefficient, due to the nature of linear combinations, the produced mask can vanish or include leakage from other objects. Thus, the network has to play a balancing act to produce the right coefficients, and adding more prototypes makes this harder. In fact, we find that for higher values of $k$, the network simply adds redundant prototypes with small edge-level variations that slightly increase $AP_{95}$, but not much else.

## 4 Backbone Detector

For our backbone detector we prioritize speed as well as feature richness, since predicting these prototypes and coefficients is a difficult task that requires good features to do well. Thus, the

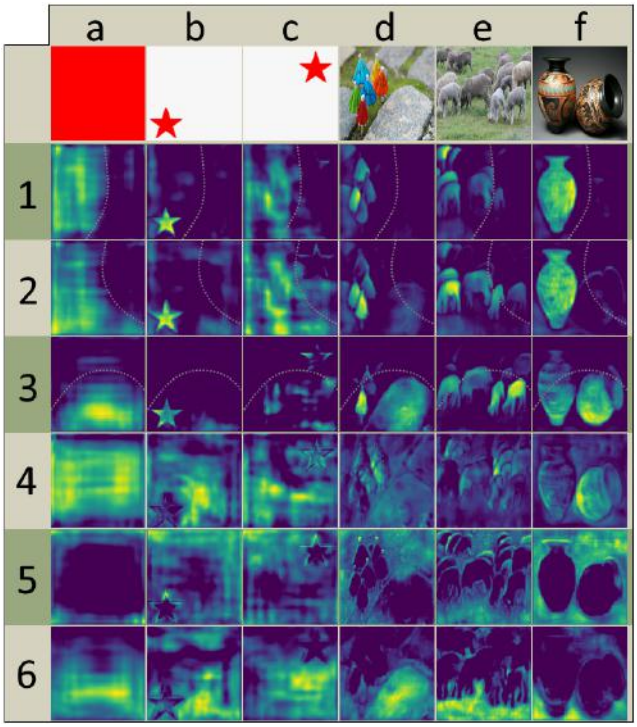



Fig. 5: **Prototype Behavior** The activations of the same six prototypes (y axis) across different images (x axis). Prototypes 1-3 respond to objects to one side of a soft, implicit boundary (marked with a dotted line). Prototype 4 activates on the bottom-left of objects (for instance, the bottom left of the umbrellas in image d); prototype 5 activates on the background and on the edges between objects; and prototype 6 segments what the network perceives to be the ground in the image. These last 3 patterns are most clear in images d-f.

design of our backbone detector closely follows RetinaNet [27] with an emphasis on speed.

**YOLACT Detector** We use ResNet-101 [10] with FPN [49] as our default feature backbone and a base image size of $550 \times 550$. We do not preserve aspect ratio in order to get consistent evaluation times per image, and at least on the almost-square COCO images, we don't observe any benefit for maintaining aspect ratio. Like RetinaNet, we modify FPN by not producing $P_2$ and producing $P_6$ and $P_7$ as successive $3 \times 3$ stride 2 *conv* layers starting from $P_5$ (not $C_5$) and place 3 anchors with aspect ratios $[1, 1/2, 2]$ on each. The anchors of $P_3$ have areas of 24 pixels squared, and every subsequent layer has double the scale of the previous (resulting in the scales $[24, 48, 96, 192, 384]$). For the prediction head attached to each $P_i$, we have one $3 \times 3$ *conv* shared by all three branches, and then each branch gets its own $3 \times 3$ conv in parallel. Compared to RetinaNet, our prediction head design (see Figure 4) is more lightweight and much faster. We apply smooth-$L_1$ loss to train box regressors and encode box regression coordinates in the same way as SSD [6]. To train class prediction, we use softmax cross entropy with $c$ positive labels and 1 background label, selecting training examples using OHEM [51] with a 3:1 neg:pos ratio. Thus, unlike RetinaNet we do not use focal loss, which we found not to be viable in our situation. Finally, we do not do NMS during training, as one ground truth can match to multiple predictions. For each such positive prediction, we train both its box and mask.

With these design choices, we find that this backbone performs better and faster than SSD [6] modified to use ResNet-101 [10], with the same image size.

## 5 Other Improvements

We also discuss other improvements that either increase speed with little effect on performance or increase performance with no speed penalty.

### 5.1 Fast NMS

After producing bounding box regression coefficients and class confidences for each anchor, like most object detectors we perform NMS to suppress duplicate detections. In many previous works [1], [2], [4], [6], [7], [27], NMS is performed sequentially. That is, for each of the $c$ classes in the dataset, sort the detected boxes descending by confidence, and then for each detection remove all those with lower confidence than it that have an IoU overlap greater than some threshold. While this sequential approach is fast enough at speeds of around 5 fps, it becomes a large barrier for obtaining 30 fps (for instance, a 10 ms improvement at 5 fps results in a 0.26 fps boost, while a 10 ms improvement at 30 fps results in a 12.9 fps boost).

To fix the sequential nature of traditional NMS, we introduce Fast NMS, a version of NMS where every instance can be decided to be kept or discarded in parallel. To do this, we simply allow already-removed detections to suppress other detections, which is not possible in traditional NMS. This relaxation allows us to implement Fast NMS entirely in standard GPU-accelerated matrix operations.

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 [2], Fast NMS is 15.0 ms faster than their CUDA implementation of traditional NMS with a performance loss of only 0.3 mAP.

### 5.2 Semantic Segmentation Loss

While Fast NMS trades a small amount of performance for speed, there are ways to increase performance with no speed penalty. One of those ways is to apply extra losses to the model during training using modules not executed at test time. This effectively increases feature richness while at no speed penalty.

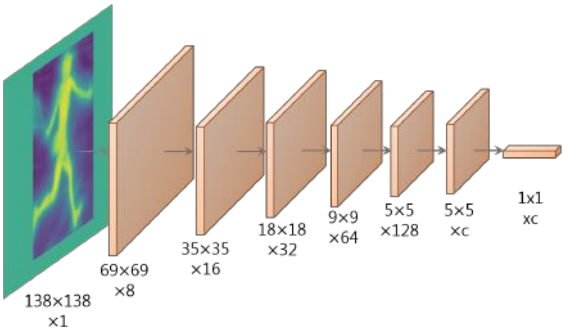



Fig. 6: **Fast Mask Re-scoring Network Architecture** Our mask scoring branch consists of 6 conv layers with ReLU non-linearity and 1 global pooling layer. Since there is no feature concatenation nor any fc layers, the speed overhead is only $\sim$1 ms.

Thus, we apply a semantic segmentation loss on our feature space using layers that are only evaluated during training. Note that because we construct the ground truth for this loss from instance annotations, this does not strictly capture semantic segmentation (i.e., we do not enforce the standard one class per pixel). To create predictions during training, we simply attach a 1x1 *conv* layer with $c$ output channels directly to the largest feature map ($P_3$) in our backbone. Since each pixel can be assigned to more than one class, we use sigmoid and $c$ channels instead of softmax and $c+1$. This loss is given a weight of 1 and results in a $+0.4$ mAP boost.

## 6 YOLACT++

YOLACT, as introduced thus far, is viable for real-time applications and only consumes $\sim$1500 MB of VRAM even with a ResNet-101 backbone. We believe these properties make it an attractive model that could be deployed in low-capacity embedded systems.

We next explore several performance improvements to the original framework, while keeping the real-time demand in mind. Specifically, we first introduce an efficient and fast mask re-scoring network, which re-ranks the mask predictions according to their mask quality. We then identify ways to improve the backbone network with deformable convolutions so that our feature sampling aligns better with instances, which results in a better backbone detector and more precise mask prototypes. We finally discuss better choices for the detection anchors to increase recall.

### 6.1 Fast Mask Re-Scoring Network

As indicated by Mask Scoring R-CNN [16], there is a discrepancy in the model's classification confidence and the quality of the predicted mask (i.e., higher quality mask segmentations don't necessarily have higher class confidences). Thus, to better correlate the class confidence with mask quality, Mask Scoring R-CNN adds a new module to Mask R-CNN that learns to regress the predicted mask to its mask IoU with ground-truth.

Inspired by [16], we introduce a *fast* mask re-scoring branch, which rescores the predicted masks based on their mask IoU with ground-truth. Specifically, our Fast Mask Re-Scoring Network is a 6-layer FCN with ReLU non-linearity per conv layer and a final global pooling layer. It takes as input YOLACT's cropped mask prediction (before thresholding) and outputs the mask IoU for each object category. We rescore each mask by taking the product between the predicted mask IoU for the category predicted by our classification head and the corresponding classification confidence (see Figure 6).

Our method differs from Mask Scoring R-CNN [16] in the following important ways: (1) Our input is only the mask at the full image size (with zeros outside the predicted box region) whereas their input is the ROI repooled mask concatenated with the feature from the mask prediction branch, and (2) we don't have any $fc$ layers. These make our method *significantly* faster. Specifically, the speed overhead of adding the Fast Mask Re-Scoring branch to YOLACT is 1.2 ms, which changes the fps from $34.4$ to $33$ for our ResNet-101 model, while the overhead of incorporating Mask Scoring R-CNN's module into YOLACT is 28 ms (note that the overhead is particularly large for YOLACT as we need to repool features whereas Mask-RCNN could simply reuse ROI features), which would change the fps from $34.4$ to $17.5$. The speed difference mainly comes from MS R-CNN's usage of the ROI align operation, its $fc$ layers, and the feature concatenation in the input.

### 6.2 Deformable Convolution with Intervals

Deformable Convolution Networks (DCNs) [13], [14] have proven to be effective for object detection, semantic segmentation, and instance segmentation due to its replacement of the rigid grid sampling used in conventional convnets with free-form sampling. We follow the design choice made by DCNv2 [14] and replace the 3x3 convolution layer in each ResNet block with a 3x3 deformable convolution layer for $C_3$ to $C_5$. Note that we do not use the modulated deformable modules because we can't afford the inference time overhead that they introduce.

Adding deformable convolution layers into the backbone of YOLACT, leads to a $+1.8$ mask mAP gain with a speed overhead of 8 ms. We believe the boost is due to: (1) DCN can strengthen the network's capability of handling instances with different scales, rotations, and aspect ratios by aligning to the target instances. (2) YOLACT, as a single-shot method, does not have a re-sampling process. Thus, a better and more flexible sampling strategy is more critical to YOLACT than two-stage methods, such as Mask R-CNN because there is no way to recover sub-optimal samplings in our network. In contrast, the ROI align operation in Mask R-CNN can address this problem to some extent by aligning all objects to a canonical reference region.

Even though the performance boost is fairly decent when directly plugging in the deformable convolution layers following the design choice in [14], the speed overhead is quite significant as well (see Table 7). This is because there are 30 layers with deformable convolutions when using ResNet-101. To speed up our ResNet-101 model while maintaining its performance boost, we explore using less deformable convolutions. Specifically, we try having deformable convolutions in four different configurations: (1) in the last 10 ResNet blocks, (2) in the last 13 ResNet blocks, (3) in the last 3 ResNet stages with an interval of 3 (i.e., skipping two ResNet blocks in between; total 11 deformable layers), and (4) in the last 3 ResNet stages with an interval of 4 (total 8 deformable layers). Given the results in Table 7, the *DCN (interval=3)* setting is chosen as the final configuration in YOLACT++, which cuts down the speed overhead by $5.2$ ms to $2.8$ ms and only has a $0.2$ mAP drop compared to not having an interval.

| Method | Backbone | FPS | Time | AP | $AP_{50}$ | $AP_{75}$ | $AP_S$ | $AP_M$ | $AP_L$ |
|---|---|---|---|---|---|---|---|---|---|
| PA-Net [15] | R-50-FPN | 4.7 | 212.8 | 36.6 | 58.0 | 39.3 | 16.3 | 38.1 | 53.1 |
| RetinaMask [52] | R-101-FPN | 6.0 | 166.7 | 34.7 | 55.4 | 36.9 | 14.3 | 36.7 | 50.5 |
| FCIS [3] | R-101-C5 | 6.6 | 151.5 | 29.5 | 51.5 | 30.2 | 8.0 | 31.0 | 49.7 |
| Mask R-CNN [2] | R-101-FPN | 8.6 | 116.3 | 35.7 | 58.0 | 37.8 | 15.5 | 38.1 | 52.4 |
| MS R-CNN [16] | 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 |
| YOLACT-550++ | R-50-FPN | 33.5 | 29.9 | 34.1 | 53.3 | 36.2 | 11.7 | 36.1 | 53.6 |
| YOLACT-550++ | R-101-FPN | 27.3 | 36.7 | 34.6 | 53.8 | 36.9 | 11.9 | 36.8 | 55.1 |

TABLE 1: **MS COCO [11] 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 [10] and DarkNet [1], respectively. Our base model, YOLACT-550 with ResNet-101, is 3.9x faster than the previous fastest approach with competitive mask mAP. Our YOLACT++-550 model with ResNet-50 has the same speed while improving the performance of the base model by 4.3 mAP. Compared to Mask R-CNN, YOLACT++-R-50 is 3.9x faster and falls behind by only 1.6 mAP.

| Method | NMS | AP | FPS | Time |
|---|---|---|---|---|
| YOLACT | Standard | **30.0** | 24.0 | 41.6 |
| | Fast | 29.9 | **33.5** | **29.8** |
| 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 |
|---|---|---|---|
| 8 | 26.8 | **33.0** | **30.4** |
| 16 | 27.1 | 32.8 | 30.5 |
| *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 $k$. We choose 32 for its mix of performance and speed.

| 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 | **33.5** | **29.8** |

(c) **Accelerated Baselines** We compare to other baseline methods by tuning their speed-accuracy trade-offs. *fc*-mask is our model but with $16 \times 16$ 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.

### 6.3 Optimized Prediction Head

Finally, as YOLACT is based off of an anchor-based backbone detector, choosing the right hyper-parameters for the anchors, such as their scales and aspect ratios, is very important. We therefore revisit our anchor choice and compare with the anchor design of RetinaNet [27] and RetinaMask [52]. We try two variations: (1) keeping the scales unchanged while increasing the anchor aspect ratios from $[1, 1/2, 2]$ to $[1, 1/2, 2, 1/3, 3]$, and (2) keeping the aspect ratios unchanged while increasing the scales per FPN level by threefold ([1x, $2^{\frac{1}{3}}$x, $2^{\frac{2}{3}}$x]). The former and latter increases the number of anchors compared to the original configuration of YOLACT by $\frac{5}{3}$x and 3x, respectively. As shown in Table 3, using multi-scale anchors per FPN level (config 2) produces the best speed vs. performance trade off.

## 7 Results

We report instance segmentation results on MS COCO [11] and Pascal 2012 SBD [37] using the standard metrics. For MS COCO, we train on `train2017` and evaluate on `val2017` and `test-dev`. We also report box detection results on MS COCO.

### 7.1 Implementation Details

We train all models with batch size 8 *on one GPU* using ImageNet [53] 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 $5\times10^{-4}$, a momentum of 0.9, and all data augmentations used in SSD [6]. 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.

### 7.2 Mask Results

We first compare YOLACT to state-of-the art methods on COCO's `test-dev` set in Table 1. Because our main goal is speed, we compare against other single model results with no test-time augmentations. We report all speeds computed on a single Titan Xp, so some listed speeds may be faster than in the original paper.

YOLACT-550 offers competitive instance segmentation performance while at 3.8x the speed of the previous fastest instance segmentation method on COCO. We also note an interesting difference in where the performance of our method lies compared to others. Supporting our qualitative findings in Figure 8, the gap between YOLACT-550 and Mask R-CNN at the 50% overlap threshold is 9.5 AP, while it's 6.6 at the 75% IoU threshold. This is different from the performance of FCIS, for instance, compared to Mask R-CNN where the gap is consistent (AP values of 7.5 and 7.6 respectively). Furthermore, at the highest (95%) IoU threshold, we outperform Mask R-CNN with 1.6 vs. 1.3 AP.

Fig. 7: **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 | $AP_{mask}$ | $AP_{bbox}$ | FPS | Time |
|---|---|---|---|---|
| YOLACT-550 | 27.7 | 29.8 | 32.4 | 30.9 |
| w/ 5 Aspect Ratios | 28.0 | 30.1 | **33.2** | **30.1** |
| w/ 3 Scales | **30.2** | **32.5** | 31.2 | 32.1 |

TABLE 3: **Different Anchor Choices of Prediction Head** We compare different anchor aspect ratios and scales. All models were trained for 400k iterations. Results on MS COCO `val2017`.

| Method | Backbone | FPS | Time | $mAP^r_{50}$ | $mAP^r_{70}$ |
|---|---|---|---|---|---|
| MNC [54] | VGG-16 | 2.8 | 360 | 63.5 | 41.5 |
| FCIS [3] | R-101-C5 | 9.6 | 104 | 65.7 | 52.1 |
| YOLACT-550 | R-50-FPN | **47.6** | **21.0** | **72.3** | **56.2** |

TABLE 4: **Pascal 2012 SBD [37] Results** Timing for FCIS redone on a Titan Xp for fairness. Since Pascal has fewer and easier detections than COCO, YOLACT does much better than previous methods. Note that COCO and Pascal FPS are not comparable because Pascal has fewer classes.

We also report numbers for alternate model configurations in Table 1. In addition to our base $550 \times 550$ image size model, we train $400 \times 400$ (YOLACT-400) and $700 \times 700$ (YOLACT-700) models, adjusting the anchor scales accordingly ($s_x = s_{550}/550 * x$). Lowering the image size results in a large decrease in performance, demonstrating that instance segmentation naturally demands larger images. Then, raising the image size decreases speed significantly but also increases performance, as expected. In addition to our base backbone of ResNet-101 [10], we also test ResNet-50 and DarkNet-53 [1] to obtain even faster results. If higher speeds are preferable we suggest using ResNet-50 or DarkNet-53 instead of lowering the image size, as these configurations perform much better than YOLACT-400, while only being slightly slower.

The bottom two rows in Table 1 show the results of our YOLACT++ model with ResNet-50 and ResNet-101 backbones. With the proposed enhancements, YOLACT++ obtains a huge performance boost over YOLACT (5.9 mAP for the ResNet-50 model and 4.8 mAP for the ResNet-101 model) while maintaining high speed. In particular, our YOLACT++-ResNet-50 model runs at a real-time speed of 33.5 fps, which is 3.9x faster than Mask R-CNN, while its instance segmentation accuracy only falls behind by 1.6 mAP.

Finally, we also train and evaluate our YOLACT ResNet-50 model on Pascal 2012 SBD in Table 4. YOLACT clearly outperforms popular approaches that report SBD performance, while also being significantly faster.

### 7.3 Mask Quality

Because we produce a final mask of size $138 \times 138$, and because we create masks directly from the original features (with no repooling to transform and potentially misalign the features), our masks for large objects are noticeably higher quality than those of Mask R-CNN [2] and FCIS [3]. For instance, in Figure 8, YOLACT produces a mask that cleanly follows the boundary of the arm, whereas both FCIS and Mask R-CNN have more noise. Moreover, despite being 5.9 mAP worse overall, at the 95% IoU threshold, our base model achieves 1.6 AP while Mask R-CNN obtains 1.3. This indicates that repooling does result in a quantifiable decrease in mask quality.

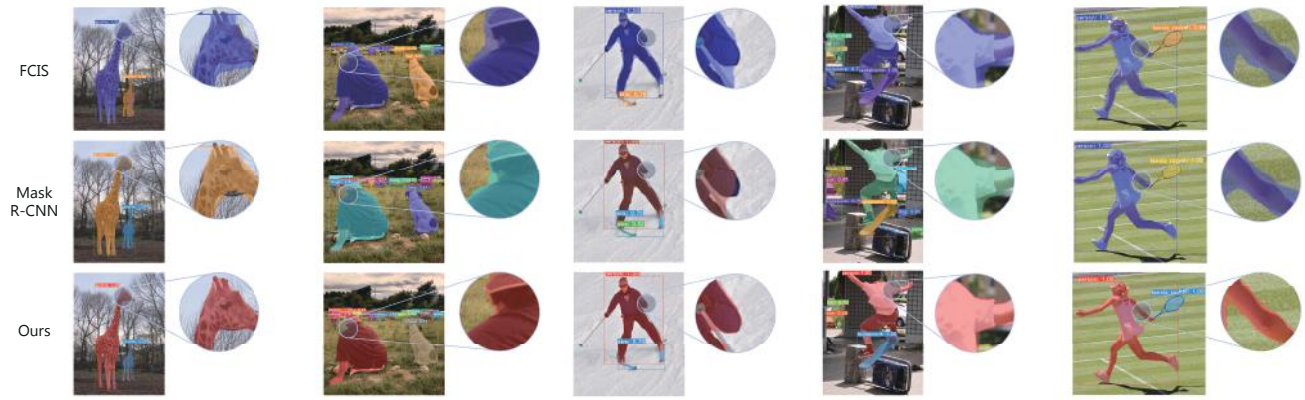



Fig. 8: **Mask Quality** Our masks are typically higher quality than those of Mask R-CNN [2] and FCIS [3] because of the larger mask size and lack of feature repooling.

| Method | Backbone | FPS | Time | AP | $AP_{50}$ | $AP_{75}$ | $AP_S$ | $AP_M$ | $AP_L$ |
|---|---|---|---|---|---|---|---|---|---|
| YOLOv3-320 [1] | D-53 | **71.17** | **14.05** | 28.2 | 51.5 | · | · | · | · |
| YOLACT-400 | R-101-FPN | 55.43 | 18.04 | **28.4** | 48.6 | 29.5 | 10.7 | 28.9 | 43.1 |
| YOLACT-550 | R-50-FPN | **59.30** | **16.86** | 30.3 | 50.8 | 31.9 | 14.0 | 31.2 | 43.0 |
| YOLOv3-416 [1] | D-53 | 54.47 | 18.36 | **31.0** | 55.3 | · | · | · | · |
| YOLACT-550 | D-53-FPN | 52.99 | 18.87 | **31.0** | 51.1 | 32.9 | 14.4 | 31.8 | 43.7 |
| YOLACT-550 | R-101-FPN | **41.14** | **24.31** | 32.3 | 53.0 | 34.3 | 14.9 | 33.8 | 45.6 |
| YOLOv3-608 [1] | D-53 | 30.54 | 32.74 | 33.0 | 57.9 | 34.4 | 18.3 | 35.4 | 41.9 |
| YOLACT-700 | R-101-FPN | 29.61 | 33.77 | **33.7** | 54.3 | 35.9 | 16.8 | 35.6 | 45.7 |

TABLE 5: **Box Performance** on COCO's `test-dev` set. For our method, timing is done without evaluating the mask branch. Both methods were timed on the same machine (using one Titan Xp). In each subgroup, we compare similar performing versions of our model to a corresponding YOLOv3 model. YOLOv3 doesn't report all metrics for the 320 and 416 versions.

### 7.4 Temporal Stability

Although we only train using static images and do not apply any temporal smoothing, we find that our model produces more temporally stable masks on videos than Mask R-CNN, whose masks jitter across frames even when objects are stationary. We believe our masks are more stable in part because they are higher quality (thus there is less room for error between frames), but mostly because our model is one-stage. Masks produced in two-stage methods are highly dependent on their region proposals in the first stage. In contrast for our method, even if the model predicts different boxes across frames, the prototypes are not affected, yielding much more temporally stable masks. See https://www.youtube.com/watch?v=Im-bqiWQ5nE for a comparison between YOLACT Base and Mask R-CNN.

### 7.5 More Qualitative Results

Figure 7 shows many examples of adjacent people and vehicles, but not many for other classes. To further support that YOLACT is not just doing semantic segmentation, we include many more qualitative results for images with adjacent instances of the same class in Figure 9.

For instance, in an image with two elephants (Figure 9 row 2, col 2), despite the fact that two instance boxes are overlapping with each other, their masks are clearly separating the instances. This is also clearly manifested in the examples of zebras (row 4, col 2) and birds (row 5, col 1).

Note that for some of these images, the box doesn't exactly crop off the mask. This is because for speed reasons (and because the model was trained in this way), we crop the mask at the prototype resolution (so one fourth the image resolution) with 1px of padding in each direction. On the other hand, the corresponding box is displayed at the original image resolution with no padding.

### 7.6 Box Results

Since YOLACT produces boxes in addition to masks, we can also compare its object detection performance to other real-time object detection methods. Moreover, while our *mask performance* is real-time, we don't need to produce masks to run YOLACT as an object detector. Thus, YOLACT is faster when run to produce boxes than when run to produce instance segmentations.

In Table 5, we compare our performance and speed to various skews of YOLOv3 [1]. We are able to achieve similar detection results to YOLOv3 at similar speeds, while not employing any of the additional improvements in YOLOv2 and YOLOv3 like multi-scale training, optimized anchor boxes, cell-based regression encoding, and objectness score. Because the improvements to our detection performance in our observation come mostly from using FPN and training with masks (both of which are orthogonal to the improvements that YOLO makes), it is likely that we can combine YOLO and YOLACT to create an even better detector.

Moreover, these detection results show that our mask branch takes *only 6 ms* in total to evaluate, which demonstrates how minimal our mask computation is.

### 7.7 YOLACT++ Improvements

Table 6 shows the contribution of each new component in our YOLACT++ model. The optimized anchor choice directly improves the recall of box prediction and boosts our backbone

Fig. 9: **More YOLACT** evaluation results on COCO's `test-dev` set with the same parameters as before. To further support that YOLACT implicitly localizes instances, we select examples with adjacent instances of the same class.

| Method | FPS | Time | AP |
|---|---|---|---|
| YOLACT-550 (R-101-FPN) | **33.5** | **29.8** | 29.9 |
| + more anchors | 30.8 | 32.5 | 31.7 |
| + deform convs (interval=3) | 28.3 | 35.3 | 33.3 |
| + fast mask re-scoring | 27.3 | 36.7 | **34.4** |
| YOLACT-550 (R-50-FPN) | **45.0** | **22.2** | 28.5 |
| + more anchors | 40.2 | 24.9 | 29.9 |
| + deform convs | 34.7 | 28.8 | 32.7 |
| + fast mask re-scoring | 33.5 | 29.9 | **33.7** |

TABLE 6: **YOLACT++ Improvements** Contribution to instance segmentation accuracy and speed overhead of each component of YOLACT++. Results on MS COCO `val2017`.

| Method | FPS | Time | AP |
|---|---|---|---|
| w/o DCN | **30.8** | **32.5** | 31.7 |
| w/ DCN | 24.7 | 40.5 | **33.5** |
| w/ DCN (interval=3) | 28.3 | 35.3 | 33.3 |
| w/ DCN (interval=4) | 29.2 | 34.3 | 33.0 |
| w/ DCN (last 10 layers) | 29.0 | 34.5 | 33.0 |
| w/ DCN (last 13 layers) | 28.0 | 35.8 | 33.0 |

TABLE 7: **Different Choices of Using Deformable Convolution Layers** The speed vs. performance trade off of different design choices when applying deformable convolutions [14] in YOLACT. Results on MS COCO `val2017`. Note that in these results, the backbone is ResNet-101 with the 3-scale anchor choice in the prediction head.

detector. The deformable convolutions help with better feature sampling by aligning the sampling positions with the instances of interest and better handles changes in scale, rotation, and aspect ratio. Importantly, with our exploration of using less deformable convolution layers, we can cut down their speed overhead significantly (from 8 ms to 2.8 ms) while keeping the performance almost the same (only 0.2 mAP drop) as compared to the original configuration proposed in [14]; see Table 7. With these two upgrades for object detection, YOLACT++ suffers less from localization failure and has finer mask predictions, as shown in Figure 10b, c, which together result in 3.4 mAP and 4.2 mAP boost for ResNet-101 and ResNet-50, respectively. In addition, the proposed fast mask re-scoring network re-ranks the mask predictions with the IoU based mask scores instead of solely relying on classification confidence. As a result, the under-estimated masks (masks with good quality but with low classification confidence) and over-estimated masks (masks with bad quality but with high classification confidence) are put into a more proper ranking as shown in Figure 10a. Our mask re-scoring method is also fast. Compared to incorporating MS R-CNN into YOLACT, it is 26.8 ms faster yet can still improve YOLACT by 1 mAP.

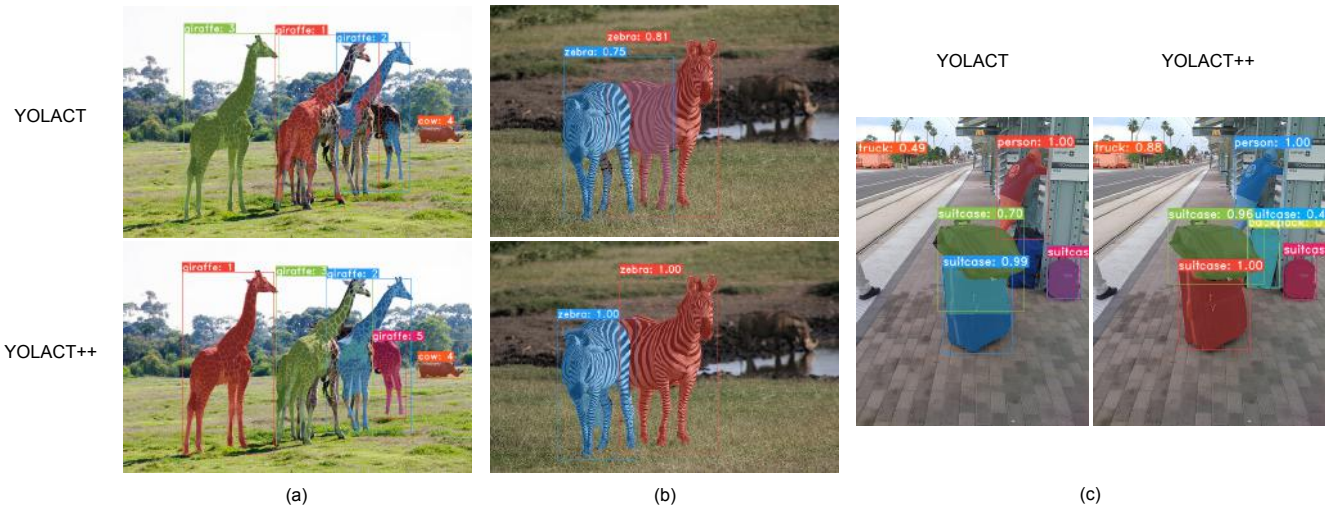



Fig. 10: **YOLACT vs. YOLACT++** (a) shows the rank of each detection in the image. As YOLACT++ has a fast mask re-scoring branch, its detections with better masks are ranked higher than those of YOLACT (see the leftmost giraffe). Since YOLACT++ is equipped with deformable convolutions in the backbone and has a better anchor design, the box recall, mask quality, and classification confidence are all increased. Specifically, (b) shows that both the box prediction and instance segmentation mask of the left zebra is more precise. (c) shows increased detection recall and improved class confidence scores.

| Model | YOLACT Base | | YOLACT Resnet50 | | YOLACT++ Base | | YOLACT++ Resnet50 | |
|---|---|---|---|---|---|---|---|---|
| Component | ms | % | ms | % | ms | % | ms | % |
| Backbone | 64.66 | 63.43% | 32.85 | 47.53% | 70.37 | 63.62% | 44.07 | 52.37% |
| FPN | 4.39 | 4.31% | 5.20 | 7.53% | 4.12 | 3.72% | 4.12 | 4.89% |
| Protonet | 5.08 | 4.98% | 5.03 | 7.28% | 4.63 | 4.19% | 4.58 | 5.44% |
| Prediction Heads | 10.72 | 10.52% | 10.58 | 15.3% | 11.94 | 10.8% | 11.81 | 14.03% |
| Fast NMS | 6.21 | 6.09% | 5.87 | 8.5% | 7.12 | 6.44% | 6.96 | 8.28% |
| Postprocessing | 7.47 | 7.33% | 6.16 | 8.91% | 5.97 | 5.40% | 6.14 | 7.29% |
| Mask Rescoring | . | . | . | . | 2.69 | 2.43% | 2.72 | 3.23% |
| Other | 3.41 | 3.35% | 3.42 | 4.95% | 3.77 | 3.40% | 3.76 | 4.47% |
| Total | 101.95 | 100% | 69.11 | 100% | 110.60 | 100% | 84.15 | 100% |

TABLE 8: **Timing Breakdown** The time taken for each stage of the model. Note that in order to properly time each portion of the model, we have to disable GPU parallelization (i.e., with `CUDA_LAUNCH_BLOCKING=1`), which means that the times shown here are much higher than what is typical of the model. The fact that this sequential execution of our model is 3 times slower than normal also shows how well our method exploits parallelization.

### 7.8 Timing Breakdown

In Table 8, we demonstrate the time taken for each part of our method with asynchronous GPU execution disabled (i.e., `CUDA_LAUNCH_BLOCKING=1`) in order to properly time each part. Note that because this is timed with parallelism turned off, the total time is much higher than the original model, and thus the time of each component has to be considered individually. The fact that our model is 3 times faster with parallelism turned on also demonstrates how effective our method is at exploiting parallel computation.

## 8 Discussion

Despite our masks being higher quality and having nice properties like temporal stability, we fall a bit behind state-of-the-art instance segmentation methods in overall performance, albeit while being much faster. Most errors are caused by mistakes in the detector: misclassification, box misalignment, etc. However, we have identified two typical errors caused by YOLACT's mask generation algorithm.

**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, it 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 7 (row 1 column 1), the blue truck under the red airplane is not properly localized.

Our YOLACT++ model addresses this problem to some degree by introducing more anchors covering more scales and applying deformable convolutions in the backbone for better feature sampling. For example, there are higher confidence and more accurate box detections in Figure 10c using YOLACT++.

**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" from outside the cropped region. This can also happen when two instances are far away from each other, because the network has learned that it doesn't need to localize far away instances—the cropping will take care of it. However, if the predicted bounding box is too big, the mask will

include some of the far away instance's mask as well. For instance, Figure 7 (row 2 column 4) exhibits this leakage because the mask branch deems the three skiers to be far enough away to not have to separate them.

Our YOLACT++ model partially mitigates these issues with a light-weight mask error down-weighting scheme, where masks exhibiting these errors will be ignored or ranked lower than higher quality masks. In Figure 10a, the leftmost giraffe's mask has the best quality and with mask re-scoring, it is ranked highest with YOLACT++ whereas with YOLACT it is ranked 3rd among all detections in the image.

**Understanding the AP Gap** However, localization failure and leakage alone are not enough to fully explain the gap between YOLACT's base model and, say, Mask R-CNN. Indeed, if we ignore all mask-related errors and replace the predicted masks with the ground-truth, our mask mAP only improves from 33.7 to 35.1 (given 34.9 box mAP) using a YOLACT++ R-50 model. Moreover, Mask R-CNN in fact has a slightly larger 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.

**Quality of mask coefficients** As YOLACT produces masks by combining prototypes with mask coefficients, it would be nice to inspect the quality of those predicted coefficients. In order to do this, after training a YOLACT++ R-50 model, we freeze everything but the mask coefficient branch, and fine-tune only the coefficient predictor on the *evaluation set*. With this setting, we only improve mask mAP from 33.7 to 33.9 even though we essentially have access to the "optimal coefficients" (i.e., fitting coefficients to test data given fixed prototypes). This shows that our predicted coefficients are very close to "optimal coefficients" and it is therefore a more promising direction to improve prototypes in order to minimize the gap between box and mask mAP.

## 9 Conclusion

We presented the first competitive single-stage real-time instance segmentation method. The key idea is to predict mask prototypes and per-instance mask coefficients in parallel, and linearly combine them to form the final instance masks. Extensive experiments on MS COCO and Pascal VOC demonstrated the effectiveness of our approach and contribution of each component. We also analyzed the emergent behavior of our prototypes to explain how YOLACT, even as an FCN, introduces translation variance for instance segmentation. Finally, with improvements to the backbone network, a better anchor design, and a fast mask re-scoring network, our YOLACT++ showed a significant boost compared to the original framework while still running at real-time.

## Acknowledgments

This work was supported in part by ARO YIP W911NF17-1-0410, NSF CAREER IIS-1751206, NSF IIS-1812850, AWS ML Research Award, Google Cloud Platform research credits, and XSEDE IRI180001.

## References

[1] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv:1804.02767 [cs]*, Apr. 2018, arXiv: 1804.02767. [Online]. Available: http://arxiv.org/abs/1804.02767

[2] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *ICCV*, 2017.

[3] Y. Li, H. Qi, J. Dai, X. Ji, and Y. Wei, "Fully convolutional instance-aware semantic segmentation," in *CVPR*, 2017.

[4] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NeurIPS*, 2015.

[5] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in *NeurIPS*, 2016.

[6] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. Berg, "Ssd: Single shot multibox detector," in *ECCV*, 2016.

[7] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *CVPR*, 2017.

[8] E. Xie, P. Sun, X. Song, W. Wang, X. Liu, D. Liang, C. Shen, and P. Luo, "PolarMask: Single Shot Instance Segmentation With Polar Representation," 2020, pp. 12 193–12 202. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2020/html/Xie_PolarMask_Single_Shot_Instance_Segmentation_With_Polar_Representation_CVPR_2020_paper.html

[9] N. Benbarka, H. u. M. Riaz, and A. Zell, "FourierNet: Compact mask representation for instance segmentation using differentiable shape decoders," *arXiv:2002.02709 [cs, eess]*, Feb. 2020, arXiv: 2002.02709. [Online]. Available: http://arxiv.org/abs/2002.02709

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[11] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and L. Zitnick, "Microsoft coco: Common objects in context," in *ECCV*, 2014.

[12] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, "Yolact: Real-time instance segmentation," in *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[13] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 764–773.

[14] X. Zhu, H. Hu, S. Lin, and J. Dai, "Deformable convnets v2: More deformable, better results," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9308–9316.

[15] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *CVPR*, 2018.

[16] Z. Huang, L. Huang, Y. Gong, C. Huang, and X. Wang, "Mask scoring r-cnn," in *CVPR*, 2019.

[17] J. Dai, K. He, Y. Li, S. Ren, and J. Sun, "Instance-sensitive fully convolutional networks," in *ECCV*, 2016.

[18] L.-C. Chen, A. Hermans, G. Papandreou, F. Schroff, P. Wang, and H. Adam, "Masklab: Instance segmentation by refining object detection with semantic and direction features," in *CVPR*, 2018.

[19] A. Kirillov, E. Levinkov, B. Andres, B. Savchynskyy, and C. Rother, "Instancecut: from edges to instances with multicut," in *CVPR*, 2017.

[20] M. Bai and R. Urtasun, "Deep watershed transform for instance segmentation," in *CVPR*, 2017.

[21] X. Liang, L. Lin, Y. Wei, X. Shen, J. Yang, and S. Yan, "Proposal-free network for instance-level object segmentation," *TPAMI*, 2018.

[22] A. Arnab and P. H. Torr, "Pixelwise instance segmentation with a dynamically instantiated network," in *CVPR*, 2017.

[23] A. Newell, Z. Huang, and J. Deng, "Associative embedding: End-to-end learning for joint detection and grouping," in *NeurIPS*, 2017.

[24] A. Harley, K. Derpanis, and I. Kokkinos, "Segmentation-aware convolutional networks using local attention masks," in *ICCV*, 2017.

[25] B. De Brabandere, D. Neven, and L. Van Gool, "Semantic Instance Segmentation with a Discriminative Loss Function," *arXiv:1708.02551 [cs]*, Aug. 2017, arXiv: 1708.02551. [Online]. Available: http://arxiv.org/abs/1708.02551

[26] A. Fathi, Z. Wojna, V. Rathod, P. Wang, H. O. Song, S. Guadarrama, and K. P. Murphy, "Semantic Instance Segmentation via Deep Metric Learning," *arXiv:1703.10277 [cs]*, Mar. 2017, arXiv: 1703.10277. [Online]. Available: http://arxiv.org/abs/1703.10277

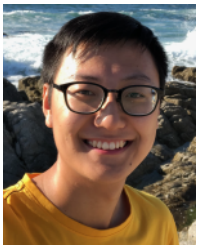
[27] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," 2017, pp. 2980–2988. [Online]. Available: https://openaccess.thecvf.com/content_iccv_2017/html/Lin_Focal_Loss_for_ICCV_2017_paper.html

[28] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *CVPR*, 2016.

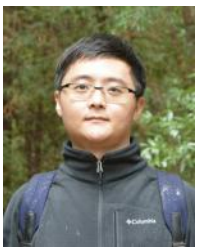
[29] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, Dec. 2017, conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[30] M. Treml, J. Arjona-Medina, T. Unterthiner, R. Durgesh, F. Friedmann, P. Schuberth, A. Mayr, M. Heusel, M. Hofmarcher, M. Widrich *et al.*, "Speeding up semantic segmentation for autonomous driving," in *MLITS, NIPS Workshop*, vol. 2, no. 7, 2016.
[31] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation," *arXiv:1606.02147 [cs]*, Jun. 2016, arXiv: 1606.02147. [Online]. Available: http://arxiv.org/abs/1606.02147
[32] N. Dvornik, K. Shmelkov, J. Mairal, and C. Schmid, "Blitznet: A real-time deep network for scene understanding," in *ICCV*, 2017.
[33] H. Zhao, X. Qi, X. Shen, J. Shi, and J. Jia, "Icnet for real-time semantic segmentation on high-resolution images," in *ECCV*, 2018.
[34] S. Jetley, M. Sapienza, S. Golodetz, and P. Torr, "Straight to shapes: real-time detection of encoded shapes," in *CVPR*, 2017.
[35] J. Uhrig, E. Rehder, B. Fröhlich, U. Franke, and T. Brox, "Box2pix: Single-shot instance segmentation by assigning pixels to object boxes," in *IEEE Intelligent Vehicles Symposium*, 2018.
[36] M. Everingham, L. Van Gool, C. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010.
[37] B. Hariharan, P. Arbeláez, L. Bourdev, S. Maji, and J. Malik, "Semantic contours from inverse detectors," in *ICCV*, 2011.
[38] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *CVPR*, 2016.
[39] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *CVPR*, 2012.
[40] D. Neven, B. D. Brabandere, M. Proesmans, and L. V. Gool, "Instance segmentation by jointly optimizing spatial embeddings and clustering bandwidth," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8837–8845.
[41] T. Leung and J. Malik, "Representing and recognizing the visual appearance of materials using three-dimensional textons," *IJCV*, 2001.
[42] J. Sivic and A. Zisserman, "Video google: A text retrieval approach to object matching in videos," in *ICCV*, 2003.
[43] J. Yang, J. Wright, T. Huang, and Y. Ma, "Image super-resolution via sparse representation," *IEEE Transactions on Image Processing*, 2010.
[44] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong, "Locality-constrained linear coding for image classification," in *CVPR*, 2010.
[45] T. Zhang, B. Ghanem, S. Liu, C. Xu, and N. Ahuja, "Low-rank sparse coding for image classification," in *ICCV*, 2013.
[46] S. Agarwal and D. Roth, "Learning a sparse representation for object detection," in *ECCV*, 2002.
[47] X. Ren and D. Ramanan, "Histograms of sparse codes for object detection," in *CVPR*, 2013.
[48] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015.
[49] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *CVPR*, 2017.
[50] A. Araujo, W. Norris, and J. Sim, "Computing receptive fields of convolutional neural networks," *Distill*, 2019.
[51] A. Shrivastava, A. Gupta, and R. Girshick, "Training region-based object detectors with online hard example mining," in *CVPR*, 2016.
[52] C.-Y. Fu, M. Shvets, and A. C. Berg, "RetinaMask: Learning to predict masks improves state-of-the-art single-shot detection for free," *arXiv:1901.03353 [cs]*, Jan. 2019, arXiv: 1901.03353. [Online]. Available: http://arxiv.org/abs/1901.03353
[53] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.
[54] J. Dai, K. He, and J. Sun, "Instance-aware semantic segmentation via multi-task network cascades," in *CVPR*, 2016.

**Daniel Bolya** graduated with a Bachelors of Science with Honors in 2019 from University of California, Davis. Daniel did most of his work for YOLACT while as an undergraduate and is currently pursuing a Ph.D. in Machine Learning at Georgia Institute of Technology. His interests lie broadly in the space of addressing core deficiencies in Machine Learning and Computer Vision, including speed, data usage, and generalizability.



**Chong Zhou** is a Master's student from the Department of Computer Science at the University of California, Davis. Prior to that, he completed his Bachelor's degree in Software Engineering from Nankai University, Tianjin, China. Chong is interested in Computer Vision and related problems in Machine Learning.



**Fanyi Xiao** is a PhD candidate working in the Computer Vision Lab at the University of California, Davis. Before this, he obtained his Master's degree in Robotics from Carnegie Mellon University (Pittsburgh, USA) and his Bachelor's degree in Computer Science from Central South University (Changsha, China). He is broadly interested in deep learning for computer vision, and video understanding in particular.

**Yong Jae Lee** is an Assistant Professor in the Department of Computer Science at the University of California, Davis. He received his Ph.D. from the University of Texas at Austin in 2012, and was a post-doc at Carnegie Mellon University (2012-2013) and UC Berkeley (2013-2014). He received his B.S. in Electrical Engineering from the University of Illinois at Urbana-Champaign in 2006. He is a recipient of the Army Research Office (ARO) Young Investigator Program (YIP) award, National Science Foundation (NSF) CAREER award, and UC Davis College of Engineering Outstanding Junior Faculty award. His main research interests are in computer vision and machine learning.