TIDE: A General Toolbox for Identifying Object Detection Errors

Supplemental Material

Table of Contents

1 Additional Discussion

There are some minor details left out of the main submission due to space constraints. We discuss those details here.

1.1 Oddities of ΔAP

While ΔAP_a correctly weights the importance of error type a, it has some potentially unintuitive properties that we list here.

First, consider Tab. 2 in the main submission. It would be nice if the improvement (or negative improvement) for each error type when summed equaled the overall improvement in AP. For instance, take the improvement row for MS R-CNN. The improvement in AP_{50} is $+0.2$, while the sum of changes in main errors $(-(0.4 - 1.5 + 0.7 + 0.3))$ is $+0.1$. Using the special errors (FP and FN, $-(0.0 + 0.3)$) would even predict worse AP_{50}: -0.3. That is, in general

$$\Delta AP \neq -\sum_{o \in \mathcal{O}} \Delta E_o \tag{1}$$

This isn't a huge issue, as the relative magnitudes of each error type can still be compared. However, it's something that needs to be kept in mind while analyzing performance.

Second, and a related issue is that summing the ΔAP_a for each error type a does not result in $100 - AP$. For instance, consider the special error types (FP and FN), which should account for all the error in the model. If we use the numbers from the same Tab. 2 for Mask R-CNN (first row), adding the AP_{50} with ΔAP_{FP} and ΔAP_{FN} $(58.1 + 15.9 + 17.8)$ yields 91.8, not 100. Similarly, for YOLACT++ we have $(51.8 + 10.7 + 27.7)$ 90.2, which is again not 100. More concretely, for $\mathcal{O} = \{o_1, \ldots, o_n\}$ this means in general

$$AP + \Delta AP_{o_1} + \ldots + \Delta AP_{o_n} \neq 100 \tag{2}$$

This is a direct result of not computing errors progressively (Fig. 2 in the main submission), where the errors sum to 100, and in fact is an odd property of AP explained in Sec. 2.3 of the main submission: fixing multiple errors at once gives a bigger boost in mAP than fixing each error on their own.

Both of these issues have an underlying cause that we can see if we write out the same expression as in Eq. 2 but with progressive error:

$$AP + \Delta AP_{o_1,\ldots,o_n} = 100 \tag{3}$$

which begs the question, how can we relate $\Delta AP_a + \Delta AP_b$ to $\Delta AP_{a,b}$? It turns out that they differ by $(\Delta AP_a - \Delta AP_{a|b})$.

To show this, we first split each term into its definition:

$$\Delta AP_{a,b} = AP_{a,b} - AP \qquad \Delta AP_a + \Delta AP_b = AP_a + AP_b - 2AP \tag{4}$$

Table 1: **Errors over thresholds.** Evaluating the error types at different foreground IoU thresholds (t_f) using Mask R-CNN detections on COCO.

t_f	AP	E_{cls}	E_{loc}	E_{both}	E_{dupe}	E_{bkg}	E_{miss}	E_{FP}	E_{FN}
0.5	61.7	3.3	6.2	1.2	0.2	4.1	7.0	16.6	15.3
0.6	57.1	2.7	10.6	1.2	0.0	3.5	7.3	16.5	18.3
0.7	49.7	2.1	18.1	0.9	0.0	2.7	7.0	15.0	23.9
0.8	36.1	1.3	31.6	0.6	0.0	1.4	6.9	12.9	32.1
0.9	12.0	0.2	55.1	0.1	0.0	0.3	4.9	9.7	33.2

Then we rearrange the terms for the left equation to get it in terms of AP:

$$AP = AP_{a,b} - \Delta AP_{a,b} \tag{5}$$

Then, substitute 1 AP into the right equation in Eq. 4 to get

$$\Delta AP_a + \Delta AP_b = AP_a + AP_b - AP - AP_{a,b} + \Delta AP_{a,b} \tag{6}$$

We can then group $AP_a - AP$ and $-(AP_{a,b} - AP_b)$ and substitute them with the definitions for ΔAP_a and $-\Delta AP_{a|b}$ respectively (if collecting the terms a different way we could swap a and b here). This leaves us with the following:

$$\Delta AP_a + \Delta AP_b = \Delta AP_{a,b} + (\Delta AP_a - \Delta AP_{a|b}) \tag{7}$$

Since the $\Delta AP_{a|b} > \Delta AP_a$ in most cases (following the reasoning given in Sec. 2.3), this means $\Delta AP_{a,b} > \Delta AP_a + \Delta AP_b$ in most cases.

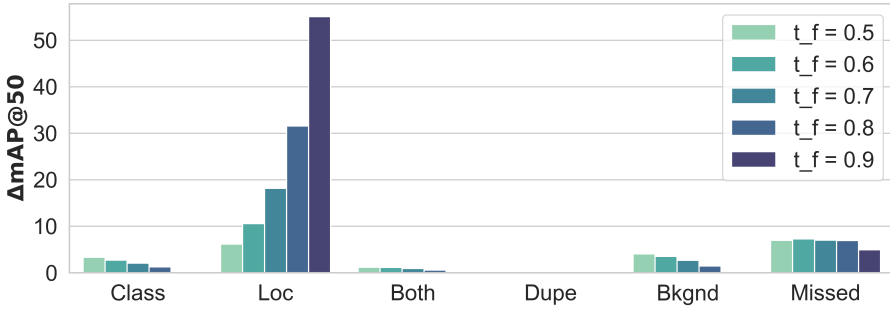
Oddities like this are why such great care needs to be taken when working with AP, since the properties it has are not intuitive.

1.2 $AP^{0.5:0.95}$

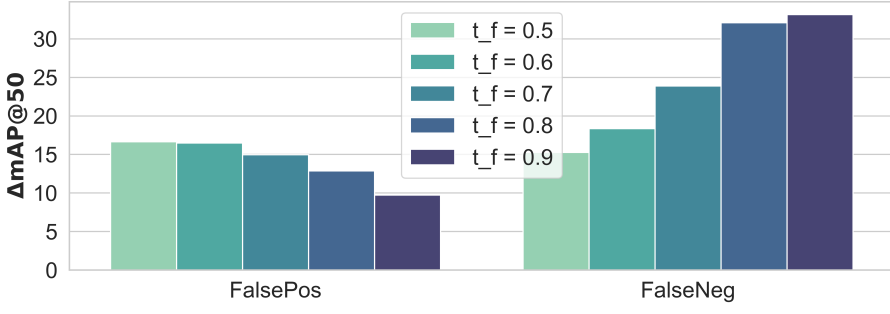
The primary metric used in the COCO and CityScapes challenges is $AP^{0.5:0.95}$, or the average of mAP across 10 IoU thresholds starting from 0.5 to 0.95 with increments of 0.05. All our analysis in our main submission is done with an IoU threshold (t_f) of 0.5, but it's worth looking at higher thresholds because of this metric.

In Tab. 1 and Fig. 1 we evaluate the error types over the IoU thresholds 0.5, 0.6, 0.7, 0.8, and 0.9 using Mask R-CNN detections on COCO. As expected, the error type that responds most strongly to higher IoU thresholds is localization error (Fig. 1a), since increasing the threshold just makes it harder to localize to the ground truth. We also see that false negatives start mattering more than false positives at higher IoU thresholds (Fig. 1b).

Thus, COCO and CityScape's average over IoU threshold metric is biased heavily toward localization errors and to a lesser extent false negatives. This is why Mask Scoring R-CNN, which rescores its masks in a way that better



(a) Main Error Types



(b) Special Error Types

Fig. 1: **Errors over thresholds.** The values for these plots are reproduced in Tab. 1. This is using Mask R-CNN detections on COCO.

calibrates localization at the expense of other error types (see Tab. 2 in the main submission), is so effective. Their approach makes no significant difference at AP_{50}, but at the higher thresholds that are biased more toward localization, they get a huge boost, leading to a big improvement in $AP^{0.5:0.95}$. Moreover, many aspects of YOLACT / YOLACT++ are much worse than other methods, which we can see by its detector performance in Fig. 4, but it localizes masks on par with other instance segmenters which give it a boost in performance in $AP^{0.5:0.95}$ for instance segmentation. Whether this is a desirable trait for a metric is up to the dataset maintainers, but designers need to take this into consideration when prioritizing areas of improvement.

2 Implementation Details

Here we discuss design choices and implementation details that weren't able to fully explain in the main submission.

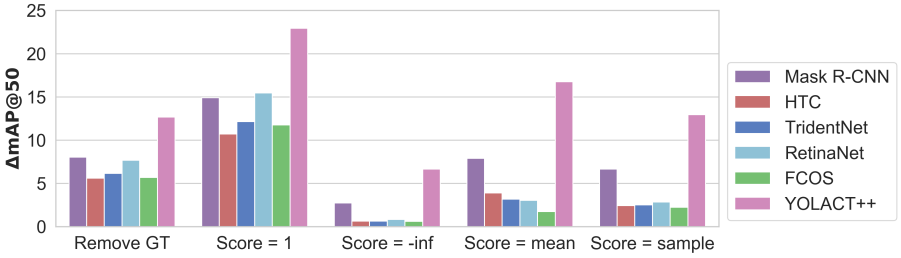


Fig. 2: **Possible Definitions for Missed GT.** Defining the oracle for missing GT is difficult, so it's important to choose a good definition. Here we compare ways of choosing the score for a new true positive versus just removing GT (what we use in our main submission).

2.1 Defining the Missed GT Oracle

As we noted in the main submission, creating a definition for "fixing" false negatives is a tricky subject. We outlined two strategies to do so: remove true negatives (i.e., lower N_{GT}) or add true positives (i.e., add a detection). We chose the former because the latter required us to choose a score for this new detection. In this section, we elaborate why choosing the right score is difficult and may lead to false conclusions. We do this by evaluating several reasonable techniques for choosing this score.

First, we could set the score to some fixed value. Two obvious choices are 1 (to put all the new detections at beginning of the sorted list) and $-\infty$ (to put all new detections at the end of the list). As evident in Fig. 2, setting the score to 1 produces very high values for the missing GT and likely overweights their contribution. In effect, setting the score to 1 assumes that whatever predictions the model add to catch this missed GT will be perfectly calibrated. Since the predictions for the other error types aren't perfectly calibrated, this results in the relative weight for missed GT being too high.

On the other hand, setting the score to $-\infty$ essentially uses the lowest score output by the model. This doesn't just assume the model will have poor calibration for this GT, but it also depends heavily on how many low scoring detections the model produced. In order to boost AP, many detectors (HTC, FCOS, TridentNet, RetinaNet) produce a lot of low-scoring detections (since COCO allows 100 detections per image). This results in this version of missed GT being disproportionately small for these models as compared to the rest.

Another, and perhaps more reasonable, method for determining the score samples from the existing predictions' scores. In Fig. 2 we test setting the score to the mean of the predicted scores and setting the score to that of an existing prediction sampled uniformly at random. However, as we see in Fig. 2, both methods produce the same skewed results as simply setting the score to $-\infty$ does (e.g., Mask R-CNN and RetinaNet for removing GT and setting the score to 1 are nearly identical, but wildly different for the other definitions).

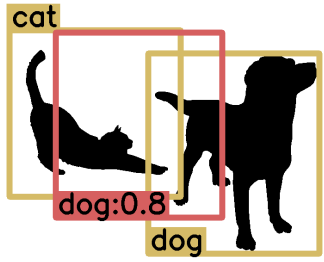


Fig. 3: **Ties in Error Assignment.** A possible tie in error assignment is illustrated here (■ = false positive detection; ■ = ground truth). The prediction is a classification error for the cat and a localization error for the dog. We break ties like this by assigning a localization error with the dog. Note that this is different to the both cls+loc error, as these are errors with two entirely separate ground truths.

In general, we can't trust the distribution of scores given by the detector as accurate, since some detectors like to use all of the available bandwidth of detections (100 per image for COCO) by flooding the predictions with low scoring detections that have some chance of being correct. This produces skewed results when defining the score as anything that depends on these low scoring detections ($-\infty$, mean, sample). Thus, we can't tell what the score for the new prediction should be, leaving us with the only option of defining the missed GT oracle as removing true negatives.

2.2 Breaking Ties in Error Assignment

Some situations can cause a predicted box to have two separate errors with two different ground truth. This is because when computing $IoU_{\max}$ for classification error, we use GT of a *different* class, while for localization we use the GT of the *same* class. Thus, it's possible for a given prediction to have a localization error w.r.t. one GT and a classification error w.r.t. another GT (as illustrated in Fig. 3).

While these ties don't happen often ($\sim 0.78\%$ of Mask R-CNN predictions on COCO), it is important to deal with them in a defined way. In the case presented in Fig. 3, we prioritize localization error over classification error, choosing to trust the classification of the model more than its localization. This choice is largely arbitrary (there are arguments for both) but needs to be made. For all other tie breakers, we follow the order we define the error types.

Note that this tie breaking is not to be confused with computing errors progressively. In our implementation, we first assign an error type to each false

positive and false negative, and then only after all positives and negatives are accounted for do we compute ΔAP.

2.3 Models and Sources Used

We used off-the-shelf models for each method tested and didn't train any new models. Some methods directly provided a COCO evaluation JSON file which we could use directly with our toolbox, while others required us to run the code ourselves. For each model, we list the method name, the model description (as describes the relevant weights file), a link to the code we used, and whether or not the method provided a COCO JSON (i.e., whether we didn't need to evaluate the model ourselves or not) in Tab. 2. Note that in general we use the Resnet101 version of each model without any bells and whistles. The one exception is YOLACT++, since it uses deformable convolutions while the rest of the models don't, so we use its Resnet50 model to compensate.

Table 2: **Model Sources.** The sources for the models we used in our analysis.

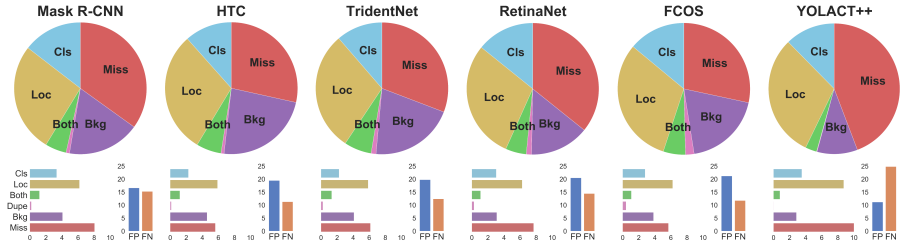
Method	Model Description	Implementation	JSON?
Mask R-CNN	R-101-FPN, 2x (35861858)	detectron	✔
MS R-CNN	ResNet-101 FPN	maskscoring_rcnn	✗
HTC	R-101-FPN	mmdetection	✗
TridentNet	TridentNet, 1x (ResNet-101)	simpledet	✗
RetinaNet	R-101-FPN, 2x (36768840)	detectron	✔
FCOS	FCOS_R_101_FPN_2x	FCOS	✗
YOLACT++	Resnet50-FPN	yolact	✗

3 COCO Instance Segmentation Summary

In Fig. 4 we show the summary plots for COCO instance segmentation that didn't make it into the main submission. For convenience, we also reproduce the detection results from the main submission.

4 API Example

In order to express the ease of use of our toolkit, we include here an example of how one can generate a summary figure as in Fig. 4. The relevant python code is included below.



(a) Detection Results

(b) Instance Segmentation Results

Fig. 4: **Summary of errors on COCO.** The same as Fig. 3 in the main submission but with instance segmentation included.

```python
from toolkit.quantify import MetricsEval
from toolkit.datasets import COCO

# Load the dataset and the model's predictions. The format is
# the same as COCOEval, so this is a drop-in replacement.
dataset     = COCO()
predictions = dataset.load_predictions('path/to/json')

# All hyperparemeters are variables that can be changed.
metrics = MetricsEval(pos_threshold=0.5, mode=MetricsEval.MASK)

# Run the evaluation of error types. Multiple runs can be done
# with the same metrics object so that they can be compared.
run = metrics.evaluate(predictions, name='Model Name')

# Plot the summary figure described above.
# If we had multiple runs, this would plot them side-by-side.
metrics.plot_summary()
```

There are a couple of important differences that separate the design of our API from other toolkits. First, our API is very modular. The dataset and prediction loading format can easily be swapped out so that we can support future datasets as well. Second, we don't output all possible figures at once. The typical use is to first run the metric evaluation, then plot the summary of errors, then dive

deeper in specific deficiencies noticed in the summary. Not only does this save the user from being overloaded with redundant and mostly useless information, but it also allows us and the broader community to implemenent new modes of analysis that can be easily slotted in to the existing code. Finally, we structure the API around comparison. One metric eval holds multiple runs, whether that be multiple different models, multiple ablations of the same model, or the same model on multiple datasets. The main goal of this toolkit is to make comparison easy and meaningful, which is something that is mostly absent in other toolkits.

5 More Misannotated COCO Ground Truth

As discussed in the main section, we find that a surprising number of the most confident errors are due to misannotated ground truth. In our study, we looked at the top 100 most confident errors in each error type (using a uniform random sample for Missed GT since those errors don't have an associated prediction) and pick out all the missannotated and ambiguously annotated examples from them. An important point to note is that COCO doesn't require their annotators to annotate all instances in an image. Where they don't annotate something, they're supposed to mark the whole area as a "crowd" annotation (i.e., for each crowd, they need only annotate a few instances and then anything else in the crowd annotation during evaluation will be ignored). We find that issues with crowd annotations are very common in COCO, such as not including a crowd annotation when there should be one or drawing a crowd annotation but then not marking the flag for crowd annotations.

This leads us to seperate all the misannotations into 3 categories: missing crowd label (i.e., the crowd annotation existed, but it wasn't flagged as a crowd annotation and thus got treated like a regular annotation), bad annotation (e.g., wrong class, box drawn incorrectly, or the GT didn't exist when it should have), and ambiguous (otherwise questionable annotations such as action figures not annotated as people, reflections of object not annotated, etc.).

We include all the misannotations we found for Mask R-CNN this way in the `misannotated_gt` folder provided alongside this document sorted by error type (except for duplicate detections, where this isn't applicable) and misannotation type. If applicable (everything but missed) the false positive prediction is in blue, if applicable (cls, loc, missed) the false negative is in red, the rest of the GT is in green, and the boxes labeled as crowds are in gray. We summarize our findings in Tab. 3. Localization, both, and background errors all have a worrying number of misannotated GT, with the both error type having a whole two thirds of the 100 most confident as misannotated! Furthermore, simply forgetting to mark a crowd box as a crowd is a surprisingly common mistake that causes localization errors. This suggests that very simple steps can be taken to improve the quality of these annotations (just fix this mislabeled crowds and draw crowds around those that don't have them). This might be a good idea for future work to persue.

Another crucial note is that these missannotations exist in the training set too. This means for instance that all the boxes that should be marked as crowds

Table 3: **Distribution of Misannotations on COCO.** We sample the top 100 errors from each error type (and randomly for missed) and bin the misannotations we found into one of three categories. Because these are the most confident examples, they have a very large effect on overall mAP.

	Cls	Loc	Both	Bkgnd	Missed
Crowd Flag	1	22	1	3	0
Bad Annotation	1	8	36	34	8
Ambiguous	5	2	29	20	3
Total	7	32	66	57	11

but aren't are being used in our models as training examples. Qualitatively, a common error detectors make is when they lump two or more objects into the same detection. This type of error isn't exclusive to one method (we've observed it in FCIS, Mask R-CNN, and YOLACT, which all are vastly different architectures). Perhaps this type of error is caused by bad training data. Certainly this type of misannotation seems very common, so we can't really confidently pin those types of errors as our detectors' fault.

Thus as a meta point, it's really important that we be careful about how much we trust a dataset. Many errors could actually be a dataset's fault and not a model's fault, but it's not very common to really explore the dataset when designing new architectures and testing on those datasets. We urge researchers in machine learning to not treat their datasets as black box benchmarks, since in many aspects the dataset matters as much as the method.