

Report 6

Commercial Viability and Limitation

Vibhav Joshi

4th June, 2020

What do we need to Consider Before choosing any tool for Our Project especially in Deep Learning or Computer Vision?

- Framework
 - Model
 - Speed and Efficiency
 - Production level
 - Scalability
-
- Detectron2 is a ground-up rewrite of Detectron that started with [maskrcnn-benchmark](#). The platform is now implemented in [PyTorch](#)(**Framework**).
 - With a new, more modular design, Detectron2 is flexible and extensible, and able to provide fast training on single or multiple GPU servers. (**Speed and Efficiency**)
 - Detectron2 includes high-quality implementations of state-of-the-art object detection algorithms, including [DensePose](#), [panoptic feature pyramid networks](#), and numerous variants of the pioneering [Mask R-CNN](#) model family also developed by FAIR. Its extensible design makes it easy to implement cutting-edge research projects without having to fork the entire codebase. (**Model**)




Detectron 2

PyTorch: The original Detectron was implemented in Caffe2. PyTorch provides a more intuitive imperative programming model that allows researchers and practitioners to iterate more rapidly on model design and experiments. Because we've rewritten Detectron2 from scratch in PyTorch, users can now benefit from PyTorch's approach to deep learning as well as the large and active community that continually improves PyTorch

Modular, extensible design: In Detectron2, we've introduced a modular design that allows users to plug custom module implementations into almost any part of an object detection system. This means that many new research projects can be written in hundreds of lines of code with a clean separation between the core Detectron2 library and the novel research implementation. We continue to refine the modular, extensible design by implementing new models and discovering new ways in which we can make Detectron2 more flexible.

New models and features: Detectron2 includes all the models that were available in the original Detectron, such as Faster R-CNN, Mask R-CNN, RetinaNet, and DensePose. It also features several new models, including Cascade R-CNN, Panoptic FPN, and TensorMask, and we will continue to add more algorithms. We've also added features such as synchronous Batch Norm and support for new datasets like LVIS.

New tasks: Detectron2 supports a range of tasks related to object detection. Like the original Detectron, it supports object detection with boxes and instance segmentation masks, as well as human pose prediction. Beyond that, Detectron2 adds support for semantic segmentation and panoptic segmentation, a task that combines both semantic and instance segmentation.



Implementation quality: Rewriting Detectron2 from the ground up allowed us to revisit low-level design decisions and address several implementation issues in the original Detectron.

Speed and scalability: By moving the entire training pipeline to GPU, we were able to make Detectron2 faster than the original Detectron for a variety of standard models. Additionally, distributing training to multiple GPU servers is now easy, making it much simpler to scale training to very large data sets.

Detectron2go: Facebook AI's computer vision engineers have implemented an additional software layer, Detectron2go, to make it easier to deploy advanced new models to production. These features include standard training workflows with in-house data sets, network quantization, and model conversion to optimized formats for cloud and mobile deployment.









BenchMarks

Here we benchmark the training speed of a Mask R-CNN in detectron2, with some other popular open source Mask R-CNN implementations.

Settings

- Hardware: 8 NVIDIA V100s with NVLink.
- Software: Python 3.7, CUDA 10.1, cuDNN 7.6.5, PyTorch 1.5, TensorFlow 1.15.0rc2, Keras 2.2.5, MxNet 1.6.0b20190820.
- Model: an end-to-end R-50-FPN Mask-RCNN model, using the same hyperparameter as the [Detectron baseline config](#) (it does not have scale augmentation).
- Metrics: We use the average throughput in iterations 100-500 to skip GPU warmup time. Note that for R-CNN-style models, the throughput of a model typically changes during training, because it depends on the predictions of the model. Therefore this metric is not directly comparable with “train speed” in model zoo, which is the average speed of the entire training run.

Main Results

Implementation	Throughput (img/s)
 Detectron2 	62
mmdetection 	53
maskrcnn-benchmark 	53
tensorpack 	50
simplifiedet 	39
Detectron 	19
matterport/Mask_RCNN 	14

Details for each implementation:

Detectron2: with release v0.1.2, run:

```
python tools/train_net.py --config-file  
configs/Detectron1-Comparisons/mask_rcnn_R_50_FPN_noaug_1x.yaml --num-gpus 8
```

•

mmdetection: at commit [b0d845f](#), run

```
./tools/dist_train.sh configs/mask_rcnn/mask_rcnn_r50_caffe_fpn_1x_coco.py 8
```

•

maskrcnn-benchmark: use commit `0ce8f6f` with `sed -i 's/torch.uint8/torch.bool/g' **/*.py; sed -i 's/AT_CHECK/TORCH_CHECK/g' **/*.cu` to make it compatible with PyTorch

1.5. Then, run training with

```
python -m torch.distributed.launch --nproc_per_node=8 tools/train_net.py --config-file
configs/e2e_mask_rcnn_R_50_FPN_1x.yaml
```

•

The speed we observed is faster than its model zoo, likely due to different software versions.

tensorpack: at commit `caafda`, export `TF_CUDNN_USE_AUTOTUNE=0`, then run

```
mpirun -np 8 ./train.py --config DATA.BASEDIR=/data/coco TRAINER=horovod
BACKBONE.STRIDE_1X1=True TRAIN.STEPS_PER_EPOCH=50 --load ImageNet-R50-AlignPadding.npz
```

•

SimpleDet: at commit `9187a1`, run

```
python detection_train.py --config config/mask_r50v1_fpn_1x.py
```

•

Detectron: run

```
python tools/train_net.py --cfg
configs/12_2017_baselines/e2e_mask_rcnn_R-50-FPN_1x.yaml
```

•

Note that many of its ops run on CPUs, therefore the performance is limited.

matterport/Mask_RCNN: at commit `3deaec`, apply the following diff, export

`TF_CUDNN_USE_AUTOTUNE=0`, then run

```
python coco.py train --dataset=/data/coco/ --model=imagenet
```

Compatibility with Other Libraries

Compatibility with Detectron (and maskrcnn-benchmark)

Detectron2 addresses some legacy issues left in Detectron. As a result, their models are not compatible: running inference with the same model weights will produce different results in the two code bases.

The major differences regarding inference are:

- The height and width of a box with corners (x_1, y_1) and (x_2, y_2) is now computed more naturally as $\text{width} = x_2 - x_1$ and $\text{height} = y_2 - y_1$; In Detectron, a “+ 1” was added both height and width.

Note that the relevant ops in Caffe2 have [adopted this change of convention](#) with an extra option. So it is still possible to run inference with a Detectron2-trained model in Caffe2.

The change in height/width calculations most notably changes:

1. encoding/decoding in bounding box regression.
 2. non-maximum suppression. The effect here is very negligible, though.
- RPN now uses simpler anchors with fewer quantization artifacts.
In Detectron, the anchors were quantized and [do not have accurate areas](#). In Detectron2, the anchors are center-aligned to feature grid points and not quantized.
 - Classification layers have a different ordering of class labels.
This involves any trainable parameter with shape $(\dots, \text{num_categories} + 1, \dots)$. In Detectron2, integer labels $[0, K-1]$ correspond to the $K = \text{num_categories}$ object categories and the label “K” corresponds to the special “background” category. In Detectron, label “0” means background, and labels $[1, K]$ correspond to the K categories.

- ROIAlign is implemented differently. The new implementation is [available in Caffe2](#).
 1. All the ROIs are shifted by half a pixel compared to Detectron in order to create better image-feature-map alignment. See `layers/roi_align.py` for details. To enable the old behavior, use `ROIAlign(aligned=False)`, or `POOLER_TYPE=ROIAlign` instead of `ROIAlignV2` (the default).
 2. The ROIs are not required to have a minimum size of 1. This will lead to tiny differences in the output, but should be negligible.
- Mask inference function is different.

In Detectron2, the “paste_mask” function is different and should be more accurate than in Detectron. This change can improve mask AP on COCO by ~0.5% absolute.

There are some other differences in training as well, but they won't affect model-level compatibility. The major ones are:

- We fixed a [bug](#) in Detectron, by making `RPN.POST_NMS_TOPK_TRAIN` per-image, rather than per-batch. The fix may lead to a small accuracy drop for a few models (e.g. keypoint detection) and will require some parameter tuning to match the Detectron results.
- For simplicity, we change the default loss in bounding box regression to L1 loss, instead of smooth L1 loss. We have observed that this tends to slightly decrease box AP50 while improving box AP for higher overlap thresholds (and leading to a slight overall improvement in box AP).

- We interpret the coordinates in COCO bounding box and segmentation annotations as coordinates in range `[0, width]` or `[0, height]`. The coordinates in COCO keypoint annotations are interpreted as pixel indices in range `[0, width - 1]` or `[0, height - 1]`. Note that this affects how flip augmentation is implemented.

We will later share more details and rationale behind the above mentioned issues about pixels, coordinates, and “+1”s.

Compatibility with Caffe2

As mentioned above, despite the incompatibilities with Detectron, the relevant ops have been implemented in Caffe2. Therefore, models trained with detectron2 can be converted in Caffe2. See [Deployment](#) for the tutorial.

Compatibility with TensorFlow

Most ops are available in TensorFlow, although some tiny differences in the implementation of `resize` / `ROIAlign` / `padding` need to be addressed. A working conversion script is provided by [tensorpack FasterRCNN](#) to run a standard detectron2 model in TensorFlow

Deployment

Caffe2 Deployment

We currently support converting a detectron2 model to Caffe2 format through ONNX.

The converted Caffe2 model is able to run without detectron2 dependency in either Python or C++. It has a runtime optimized for CPU & mobile inference, but not for GPU inference.

Caffe2 conversion requires PyTorch ≥ 1.4 and ONNX ≥ 1.6 .

Coverage

It supports 3 most common meta architectures: `GeneralizedRCNN`, `RetinaNet`, `PanopticFPN`, and most official models under these 3 meta architectures.

Users' custom extensions under these architectures (added through registration) are supported as long as they do not contain control flow or operators not available in Caffe2 (e.g. deformable convolution). For example, custom backbones and heads are often supported out of the box.

1. The conversion needs valid sample inputs & weights to trace the model. That's why the script requires the dataset. You can modify the script to obtain sample inputs in other ways.
2. With the `--run-eval` flag, it will evaluate the converted models to verify its accuracy. The accuracy is typically slightly different (within 0.1 AP) from PyTorch due to numerical precisions between different implementations. It's

recommended to always verify the accuracy in case your custom model is not supported by the conversion.

The converted model is available at the specified `caffe2_model/` directory. Two files `model.pb` and `model_init.pb` that contain network structure and network parameters are necessary for deployment. These files can then be loaded in C++ or Python using Caffe2's APIs.

The script generates `model.svg` file which contains a visualization of the network. You can also load `model.pb` to tools such as [netron](#) to visualize it.

Use the model in C++/Python

The model can be loaded in C++. An example `caffe2_mask_rcnn.cpp` is given, which performs CPU/GPU inference using `COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x`.

The C++ example needs to be built with:

- PyTorch with caffe2 inside
- gflags, glog, opencv
- protobuf headers that match the version of your caffe2
- MKL headers if caffe2 is built with MKL

The following can compile the example inside [official detectron2 docker](#):

```
sudo apt update && sudo apt install libgflags-dev libgoogle-glog-dev libopencv-dev
pip install mkl-include

wget
https://github.com/protocolbuffers/protobuf/releases/download/v3.6.1/protobuf-cpp-3.6.1.tar.gz
```

```
tar xf protobuf-cpp-3.6.1.tar.gz

export CPATH=$(readlink -f ./protobuf-3.6.1/src/):$HOME/.local/include

export CMAKE_PREFIX_PATH=$HOME/.local/lib/python3.6/site-packages/torch/

mkdir build && cd build

cmake -DTORCH_CUDA_ARCH_LIST=$TORCH_CUDA_ARCH_LIST .. && make

# To run:

./caffe2_mask_rcnn --predict_net=./model.pb --init_net=./model_init.pb
--input=input.jpg
```

Note that:

- All converted models (the .pb files) take two input tensors: “data” is an NCHW image, and “im_info” is an Nx3 tensor consisting of (height, width, 1.0) for each image (the shape of “data” might be larger than that in “im_info” due to padding).
- The converted models do not contain post-processing operations that transform raw layer outputs into formatted predictions. The example only produces raw outputs (28x28 masks) from the final layers that are not post-processed, because in actual deployment, an application often needs its custom lightweight post-processing (e.g. full-image masks for every detected object is often not necessary).

We also provide a python wrapper around the converted model, in the `Caffe2Model.__call__` method. This method has an interface that’s identical to the [pytorch versions of models](#), and it internally applies pre/post-processing code to match

the formats. They can serve as a reference for pre/post-processing in actual deployment.

Limitation

- Not only Detectron2 but any mask-rcnn model will fail when the image in dataset is not of High resolution
- The dataset should be large and of well labelled mask should be generated
- If the mask are not properly defined or there is error in labelling the mask the error will increase in validation dataset
- The goal of project should be properly defined and thus a model from the model.zoo should be chosen

The categories are:

1. Instance Segmentation
2. Semantic Segmentation
3. Panoptic Segmentation
4. Denspose Detection
5. Keypoint Detection

And thus a model and its weight should be chosen

1. Mask R-CNN
2. RetinaNet
3. Faster R-CNN
4. RPN
5. Fast R-CNN

- 
6. TensorMask
 7. PointRend
 8. DensePose









- The Hyperparameters should be properly chosen by tuning them and finding best fit parameters
- If the parameters are not chosen properly then the model will fail and not detect the mask or will have low probability

1. **SOLVER.IMS_PER_BATCH**
2. **SOLVER.BASE_LR**
3. **SOLVER.MAX_ITER**
4. **MODEL.ROI_HEADS.NUM_CLASSES**
5. **MODEL.ROI_HEADS.SCORE_THRESH_TEST**



Conclusion

- Detectron2 is the state-of-art tool for computer vision and is the recent development for the multiple computer vision tasks specially instance segmentation
- As the Detectron2 is open source it has large community which leads to continuous improvement
- Commercially the Detectron2 is best in market as it is compatible with multiple frameworks and different deployment platforms
- The speed and Efficiency is best among all current instance segmentation implementations

Implementation	Throughput (img/s)
 Detectron2 	62
mmdetection 	53
maskrcnn-benchmark 	53
tensorpack 	50
simplifiedet 	39
Detectron 	19
matterport/Mask_RCNN 	14
