Report

Instance Segmentation on Custom dataset using Detectron 2

Vibhav Joshi

25th May, 2020

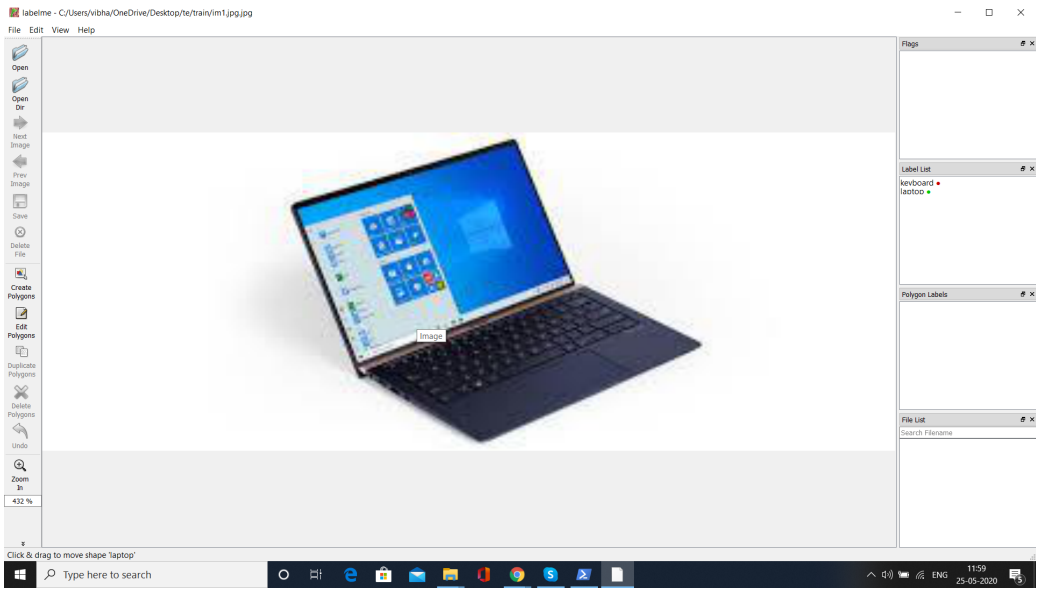
Link to google colab

File:https://colab.research.google.com/drive/16W9atfMwKjw2TUBywL9GpwL68YfTveC_?usp=sharing

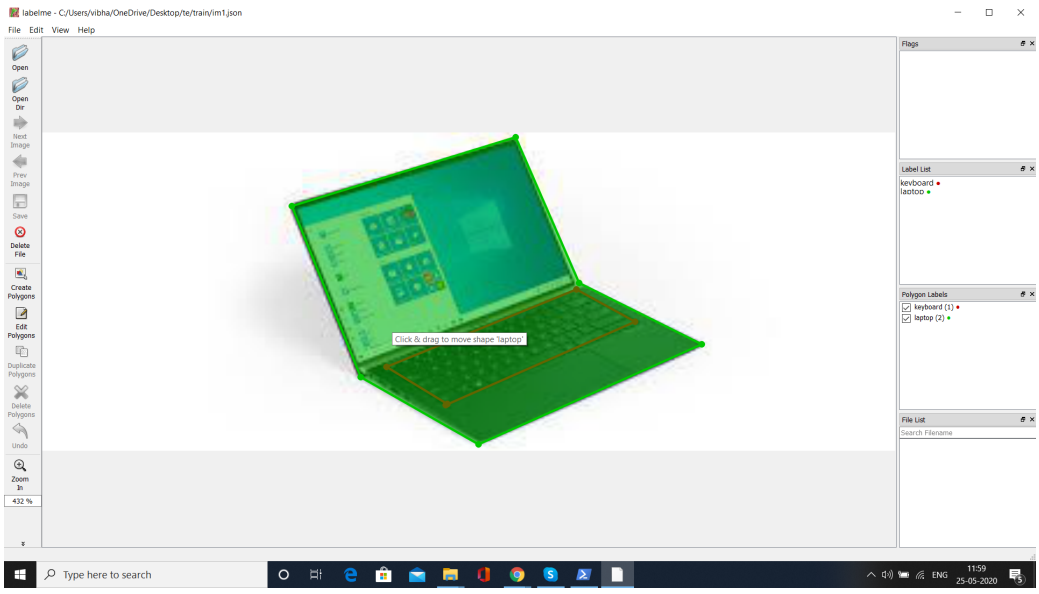
Introduction

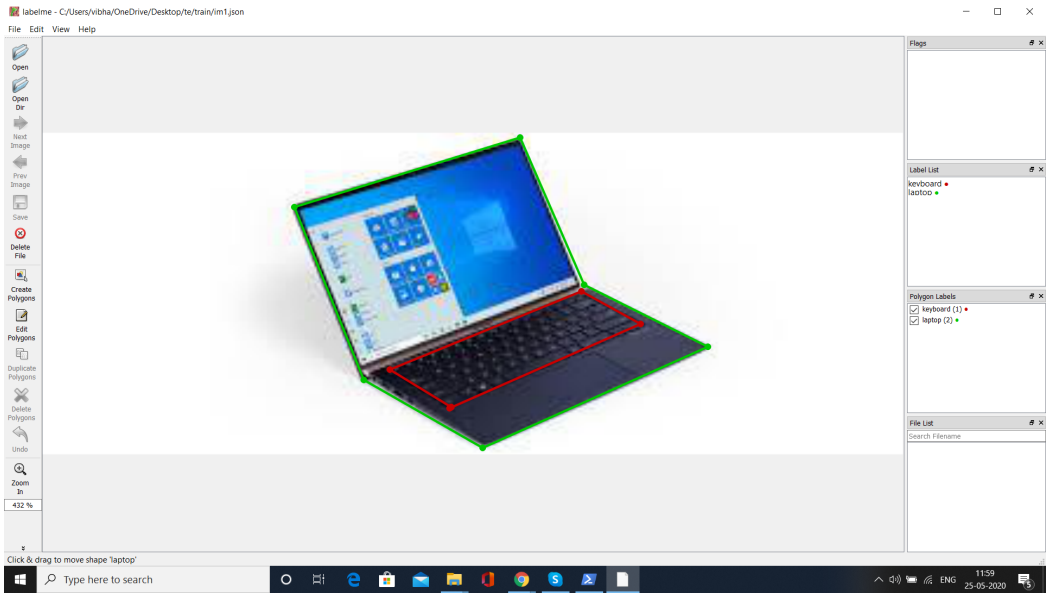
- Used custom dataset of laptops and labelled them and drew mask on them using labelling software
- converted them to json and split them in train and test dataset
- Converted them to COCO dataset format using script to convert them
- Using google colab registered them with COCO dataset
- Using Detectron 2(open source library)performed transfer learning to train model on custom dataset for mask of laptop and keyboard
- Using test dataset we tested our model to draw mask and it performed very well for mask on laptop and keyboard
- I am giving the link of google colab of the task I performed
- https://colab.research.google.com/drive/16W9atfMwKjw2TUBywL9GpwL68YfTveC_?usp=sharing

Preparing The Dataset



- Used labelme to draw mask on the image with two labels laptop and Keyboard
- The following images show the mask





- Then for each image we get a json file in the following format with different annotations


```
{
 "version": "4.2.10",
 "flags": {},
 "shapes": [
  {
   "label": "keyboard",
   "points": [
    [
     148.14814814814815,
     100.89814814814815
    ],
    [
     229.8611111111111,
     67.56481481481481
```

```
      ],
      [
        255.09259259259258,
        81.4537037037037
      ],
      [
        173.6111111111111,
        117.33333333333331
      ],
      [
        174.30555555555554,
        116.87037037037035
      ],
      [
        174.30555555555554,
        116.87037037037035
      ],
      [
        174.30555555555554,
        116.87037037037035
      ]
    ],
    "group_id": 1,
    "shape_type": "polygon",
```

```
    "flags": {}
  },
  {
    "label": "laptop",
    "points": [
      [
        107.4074074074074,
        31.685185185185176
      ],
      [
        203.7037037037037,
        2.05555555555555
      ],
      [
        231.0185185185185,
        64.78703703703702
      ],
      [
        283.7962962962963,
        91.17592592592592
      ],
      [
        187.73148148148147,
        134.23148148148147
```

```
        ],
        [
          137.03703703703704,
          105.29629629629628
        ]
      ],
      "group_id": 2,
      "shape_type": "polygon",
      "flags": {}
    }
  ],
  "imagePath": "..\\dataset\\1.jpg",
  "imageData":
"/9j/+AqzRSbb3CwmKMUtFIYUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQ
B//9k=",
  "imageHeight": 137,
  "imageWidth": 367
}
```


- Following this We convert it using python script from labelme to coco format(Working on integrating it in google colab)

Getting All dependencies

- We get all dependencies for Our Project i.e `torch, torchvision,detectron 2etc`

```
# install dependencies: (use cu100 because colab is on CUDA 10.0)

!pip install -U torch==1.4+cu100 torchvision==0.5+cu100 -f
https://download.pytorch.org/whl/torch_stable.html

!pip install cython pyyaml==5.1

!pip install -U
'git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI'

import torch, torchvision

torch.__version__

!gcc --version

# opencv is pre-installed on colab

# install detectron2:

!pip install detectron2 -f
https://dl.fbaipublicfiles.com/detectron2/wheels/cu100/index.html
```

Introduction Of detectron 2

Detectron2 is a ground-up rewrite of Detectron that started with maskrcnn-benchmark. The platform is now implemented in PyTorch. With a new, more modular design, Detectron2 is flexible and extensible, and able to provide fast training on single or multiple GPU servers. 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-CNNmodel family also developed by FAIR. Its extensible design makes it easy to implement cutting-edge research projects without having to fork the entire codebase.

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

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.

Register dataset with coco dataset

```
from detectron2.data.datasets import register_coco_instances


for d in ["train", "test"]:

    register_coco_instances(f"keyboard_{d}", {}, f"final/{d}.json",
f"final/{d}")

import random

from detectron2.data import DatasetCatalog, MetadataCatalog


dataset_dicts = DatasetCatalog.get("keyboard_train")

lapkeyboard_metadata = MetadataCatalog.get("keyboard_train")


for d in random.sample(dataset_dicts, 3):

    img = cv2.imread(d["file_name"])

    print(d["file_name"])

    v = Visualizer(img[:, :, ::-1], metadata=lapkeyboard_metadata,
scale=0.5)

    v = v.draw_dataset_dict(d)

    plt.figure(figsize = (14, 10))

    plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))

    plt.show()
```

Standard Dataset Dicts

For standard tasks (instance detection, instance/semantic/panoptic segmentation, keypoint detection), we load the original dataset into `list[dict]` with a specification similar to COCO's json annotations. This is our standard representation for a dataset.

Each dict contains information about one image. The dict may have the following fields, and the required fields vary based on what the dataloader or the task needs (see more below).

- `file_name`: the full path to the image file. Will apply rotation and flipping if the image has such exif information.
- `height`, `width`: integer. The shape of image.
- `image_id` (str or int): a unique id that identifies this image. Used during evaluation to identify the images, but a dataset may use it for different purposes.
- `annotations` (list[dict]): each dict corresponds to annotations of one instance in this image. Required by instance detection/segmentation or keypoint detection tasks.
  Images with empty `annotations` will by default be removed from training, but can be included using `DATALOADER.FILTER_EMPTY_ANNOTATIONS`.
  Each dict contains the following keys, of which `bbox`,`bbox_mode` and `category_id` are required:
  - `bbox` (list[float]): list of 4 numbers representing the bounding box of the instance.
  - `bbox_mode` (int): the format of bbox. It must be a member of structures.BoxMode. Currently supports: `BoxMode.XYXY_ABS`, `BoxMode.XYWH_ABS`.

- `category_id` (int): an integer in the range [0, num_categories) representing the category label. The value num_categories is reserved to represent the “background” category, if applicable.
- `segmentation` (list[list[float]] or dict): the segmentation mask of the instance.
    - If `list[list[float]]`, it represents a list of polygons, one for each connected component of the object. Each `list[float]` is one simple polygon in the format of `[x1, y1, ..., xn, yn]`. The Xs and Ys are either relative coordinates in [0, 1], or absolute coordinates, depend on whether “bbox_mode” is relative.
    - If `dict`, it represents the per-pixel segmentation mask in COCO’s RLE format. The dict should have keys “size” and “counts”. You can convert a uint8 segmentation mask of 0s and 1s into RLE format by `pycocotools.mask.encode(np.asarray(mask, order="F"))`.
- `keypoints` (list[float]): in the format of [x1, y1, v1,…, xn, yn, vn]. v[i] means the visibility of this keypoint. `n` must be equal to the number of keypoint categories. The Xs and Ys are either relative coordinates in [0, 1], or absolute coordinates, depend on whether “bbox_mode” is relative.

  Note that the coordinate annotations in COCO format are integers in range [0, H-1 or W-1]. By default, detectron2 adds 0.5 to absolute keypoint coordinates to convert them from discrete pixel indices to floating point coordinates.
- `iscrowd`: 0 (default) or 1. Whether this instance is labeled as COCO’s “crowd region”. Don’t include this field if you don’t know what it means.

- `sem_seg_file_name`: the full path to the ground truth semantic segmentation file. Required by semantic segmentation task. It should be an image whose pixel values are integer labels.

If your dataset is already a json file in the COCO format, the dataset and its associated metadata can be registered easily with:

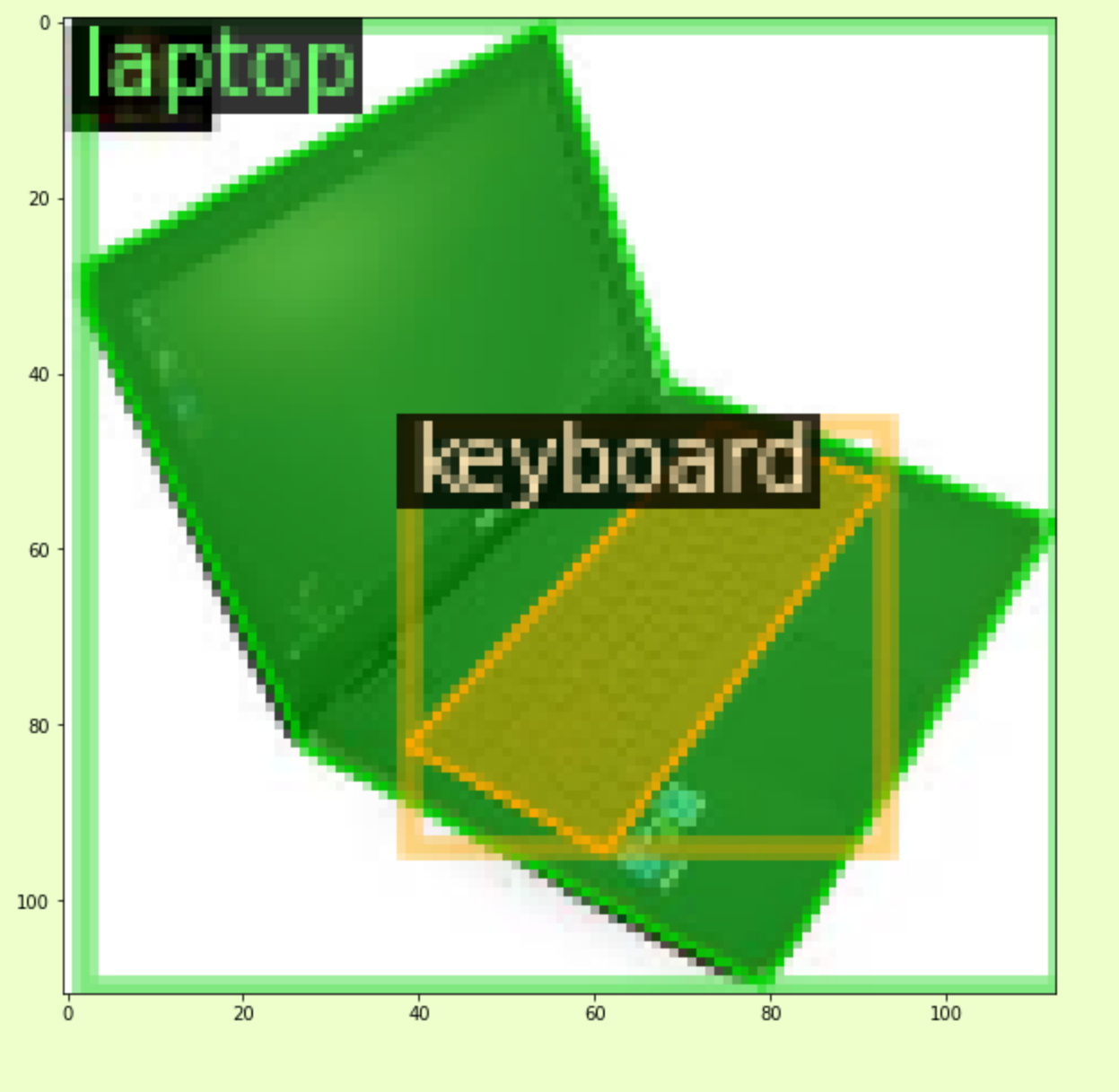
```
from detectron2.data.datasets import register_coco_instances


register_coco_instances("my_dataset", {}, "json_annotation.json", "path/to/image/dir")
```

“Metadata” for Datasets

Each dataset is associated with some metadata, accessible through `MetadataCatalog.get(dataset_name).some_metadata`. Metadata is a key-value mapping that contains information that’s shared among the entire dataset, and usually is used to interpret what’s in the dataset, e.g., names of classes, colors of classes, root of files, etc. This information will be useful for augmentation, evaluation, visualization, logging, etc. The structure of metadata depends on the what is needed from the corresponding downstream code.

If you register a new dataset through `DatasetCatalog.register`, you may also want to add its corresponding metadata through `MetadataCatalog.get(dataset_name).some_key = some_value`, to enable any features that need the metadata. You can do it like this (using the metadata key “thing_classes” as an example):

```
from detectron2.data import MetadataCatalog
```


laptop
keyboard
0
20
40
60
80
100
0
20
40
60
80
100

Build Model on Custom dataset with 2 classes

```
from detectron2.engine import DefaultTrainer

from detectron2.config import get_cfg

import os

cfg = get_cfg()

cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))

cfg.DATASETS.TRAIN = ("keyboard_train",)

cfg.DATASETS.TEST = ()

cfg.DATALOADER.NUM_WORKERS = 2

cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")

cfg.SOLVER.IMS_PER_BATCH = 2

cfg.SOLVER.BASE_LR = 0.00025

cfg.SOLVER.MAX_ITER = 1000
```

```
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 2
```

```
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

- **Use model and weights from model.zoo from detectron 2**
- **Set parameters for training**

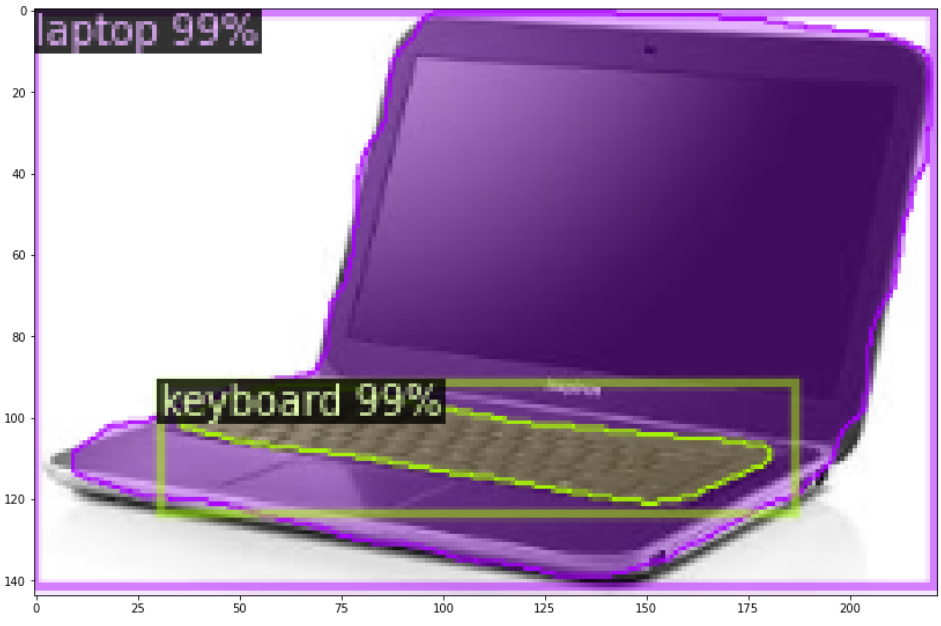
- **cfg.MODEL.ROI_HEADS.NUM_CLASSES = 2 set As We Have two classes laptop and keyboard**

Build Model on Custom dataset with 2 classes

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")

cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5

cfg.DATASETS.TEST = ("keyboard_test", )

predictor = DefaultPredictor(cfg)
```

- Save The model and Use predictor on Test dataset for making prediction


laptop 100%
keyboard 100%


laptop 99%
keyboard 99%